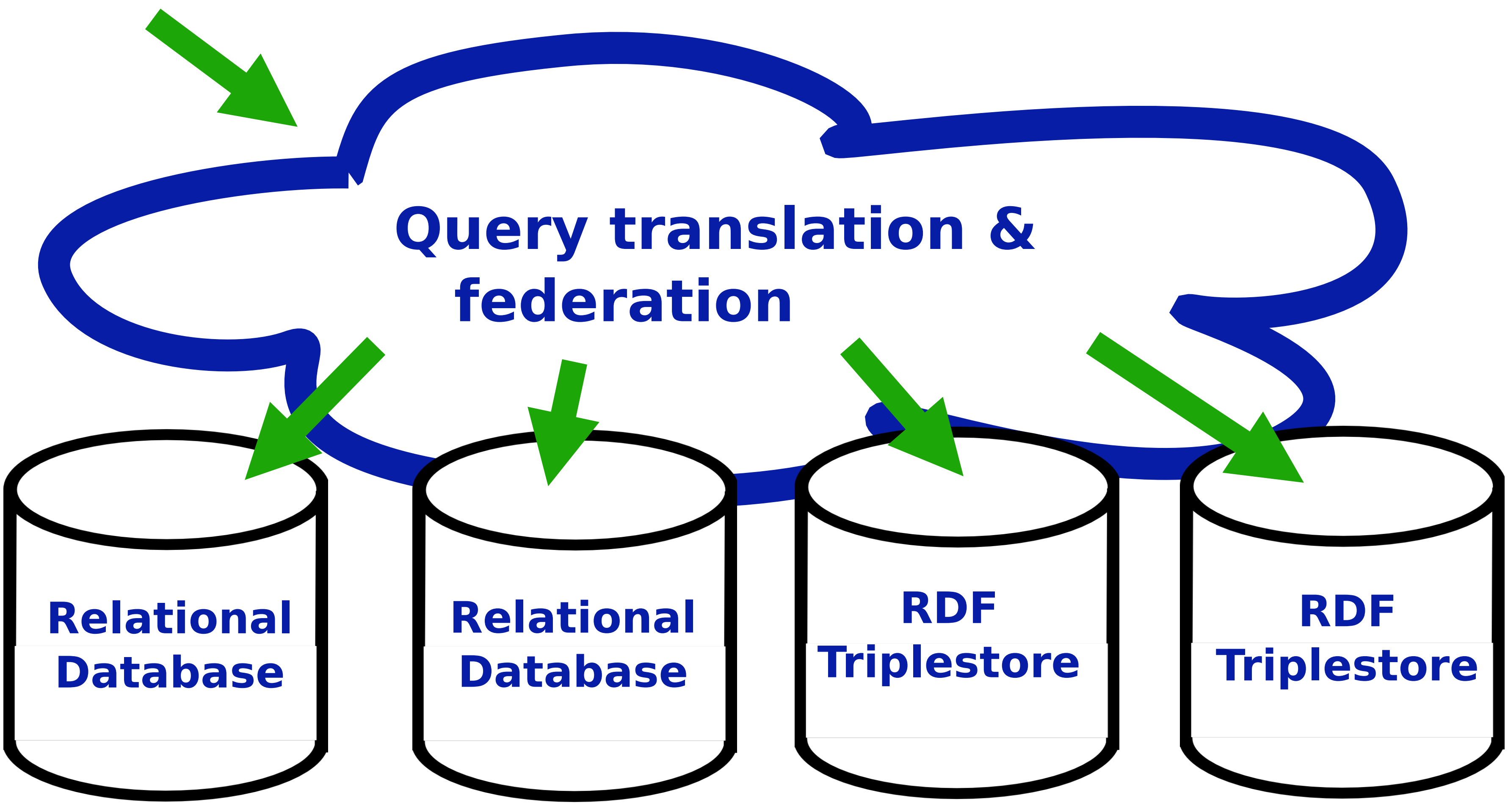
Concurrent Stream Processing

David McNeil
Revelytix, Inc.
November 2011

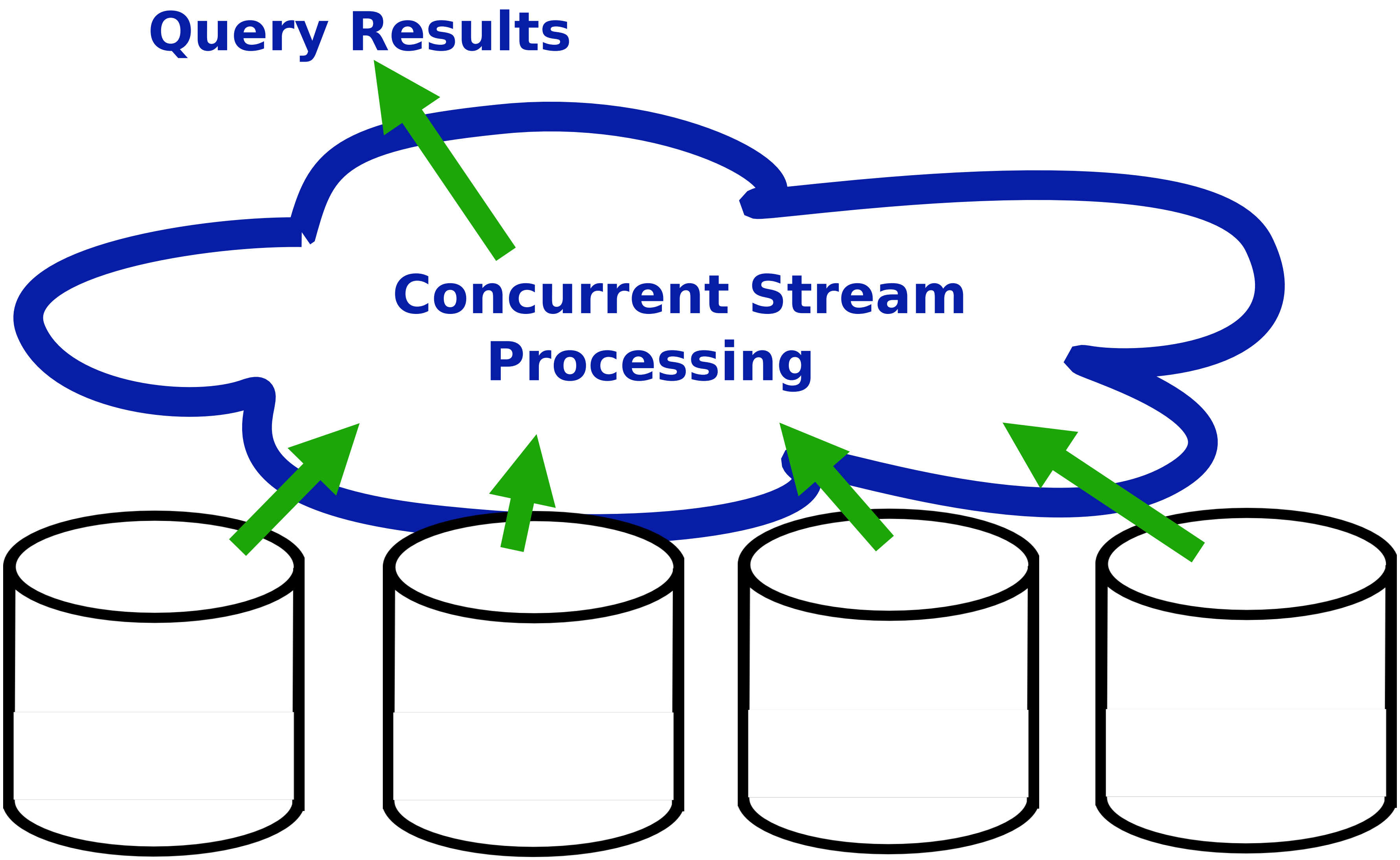
Our products translate semantic queries to SQL and execute federated queries across data sources.

SPARQL Query

```
SELECT ...
WHERE { ?x ?y ?z }
```



One of our core problems is to process many large streams of data asynchronously and in parallel.



Needed Stream Operations

The operations we need are largely analogous to Clojure sequence operations.

- combine streams
- filter streams
- compute expressions
- sort
- remove duplicates
- limit results

Other Requirements

We need to build a hybrid SPARQL/SQL database engine in Clojure
... where source data arrives in streams.

Efficient

- *Parallel execution*
- *Non-blocking on source I/O*

Robust

- *Exception handling*
- *Don't blow the heap*

Manageable

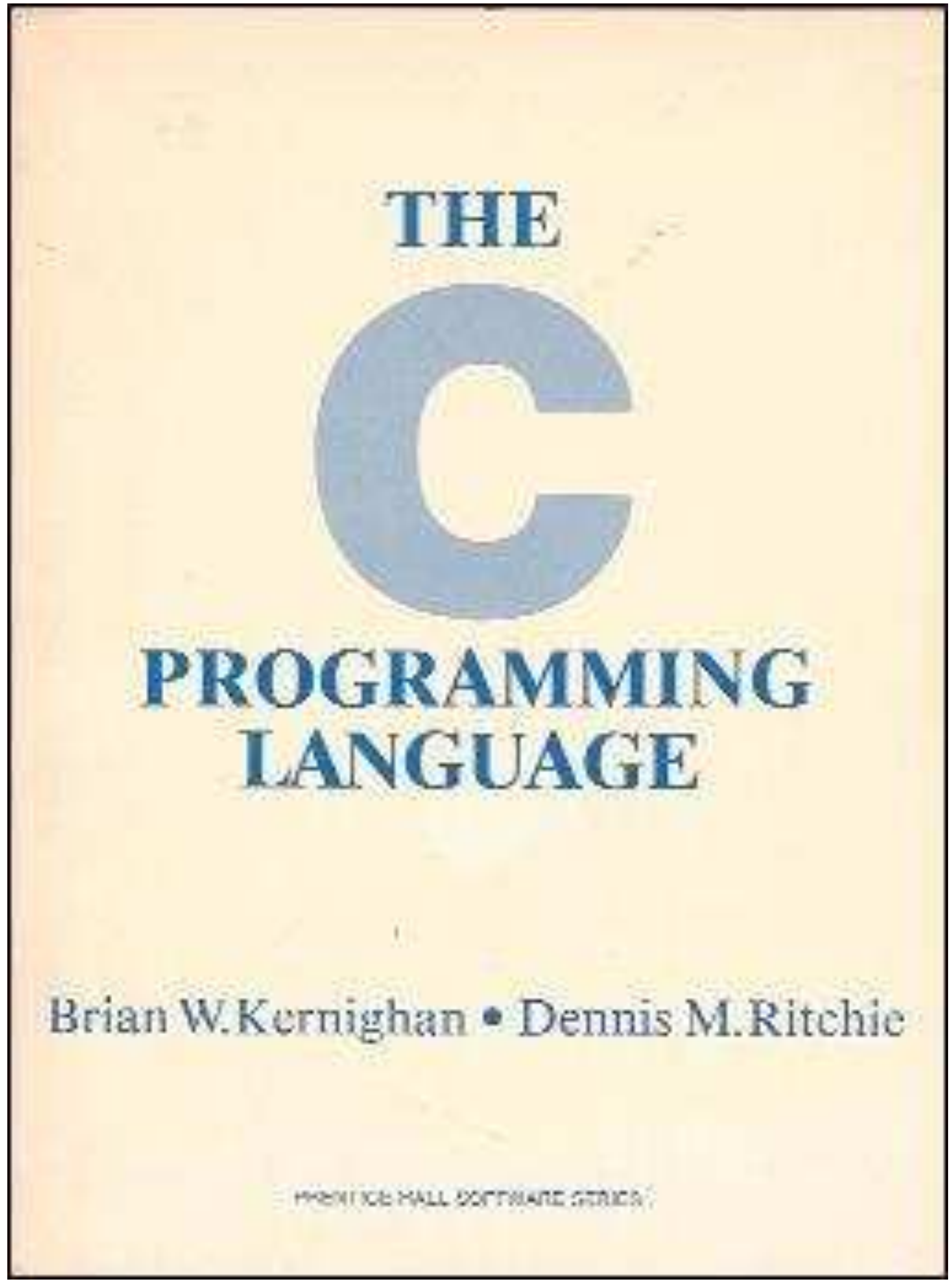
- *Visible workings - what is running?*
- *Query cancellations*
- *Query timeouts*

Shoulders of Giants

What past work can we leverage to solve this problem?

Dennis Ritchie

1941 - 2011



"some ways of coupling programs like garden hoses"

Summary--what's most important.

To put my strongest concerns in a nutshell:

1. We should have some ways of coupling programs like garden hose--screw in another segment when it becomes when it becomes necessary to massage data in another way. This is the way of IO also.

2. Our loader should be able to do link-loading and controlled establishment.

3. Our library filing scheme should allow for rather general indexing, responsibility, generations, data path switching.

4. It should be possible to get private system components (all routines are system components) for buggering around with.

M. D. McIlroy
Oct. 11, 1964

http://doc.cat-v.org/unix/pipes/pipe.png

Below is an example of a pipeline that implements a kind of spell checker for the web resource indicated by a URL. An explanation of what it does follows.

```
curl "http://en.wikipedia.org/wiki/Pipeline_(Unix)" 2>/dev/null |
sed 's/[^a-zA-Z ]/ /g' |
tr 'A-Z ' 'a-z\n' |
grep '[a-z]' |
sort -u |
comm -23 - <(sort /usr/share/dict/words) |
less
```

```
curl "http://en.wikipedia.org/wiki/Pipeline_(Unix)" 2>/dev/null |
sed 's/[^a-zA-Z ]/ /g' |
tr 'A-Z ' 'a-z\n' |
grep '[a-z]' |
sort -u |
comm -23 - <(sort /usr/share/dict/words) |
less
```

Processes

- *concurrent*
- *separate address spaces*
- *multi-threaded*
- *stdin, stdout, stderr*

```
curl "http://en.wikipedia.org/wiki/Pipeline_(Unix)" 2>/dev/null |
sed 's/[^a-zA-Z ]/ /g' |
tr 'A-Z ' 'a-z\n' |
grep '[a-z]' |
sort -u |
comm -23 - <(sort /usr/share/dict/words) |
less
```

Processes
- *concurrent*
- *separate address spaces*
- *multi-threaded*
- *stdin, stdout, stderr*

Pipes
- *asynchronous*
- *buffered*
- *EOF*

Syntax
- compact

```
curl "http://en.wikipedia.org/wiki/Pipeline_(Unix)" 2>/dev/null |
sed 's/[^a-zA-Z ]/ /g' |
tr 'A-Z ' 'a-z\n' |
grep '[a-z]' |
sort -u |
comm -23 - <(sort /usr/share/dict/words) |
less
```

Processes
- concurrent
- separate address spaces
- multi-threaded
- stdin, stdout, stderr

Pipes
- asynchronous
- buffered
- EOF

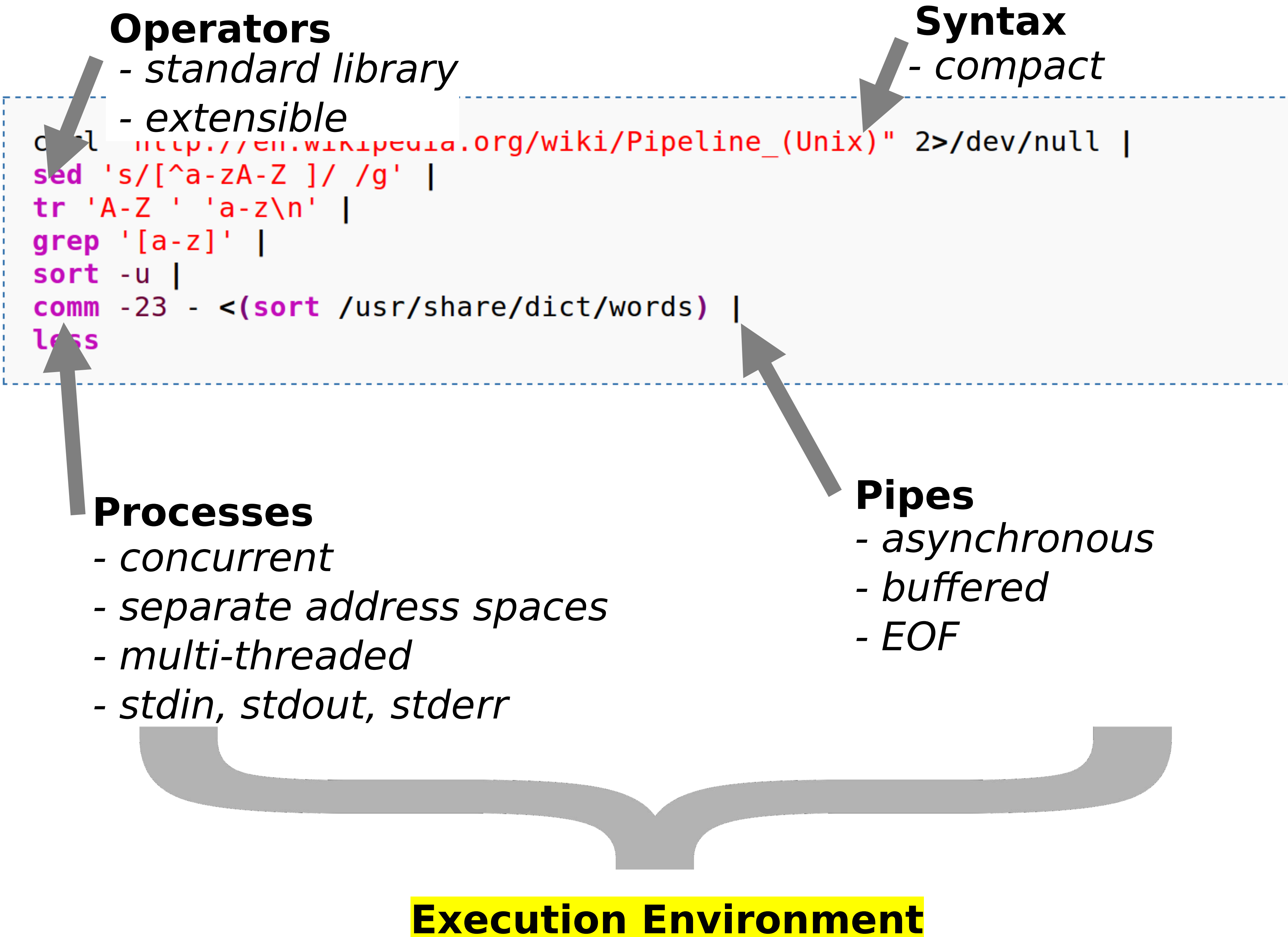
Operators
- *standard "library"*
- *extensible*

Syntax
- *compact*

```
curl "http://en.wikipedia.org/wiki/Pipeline_(Unix)" 2>/dev/null |
sed 's/[^a-zA-Z ]/ /g' |
tr 'A-Z ' 'a-z\n' |
grep '[a-z]' |
sort -u |
comm -23 - <(sort /usr/share/dict/words) |
less
```

Processes
- *concurrent*
- *separate address spaces*
- *multi-threaded*
- *stdin, stdout, stderr*

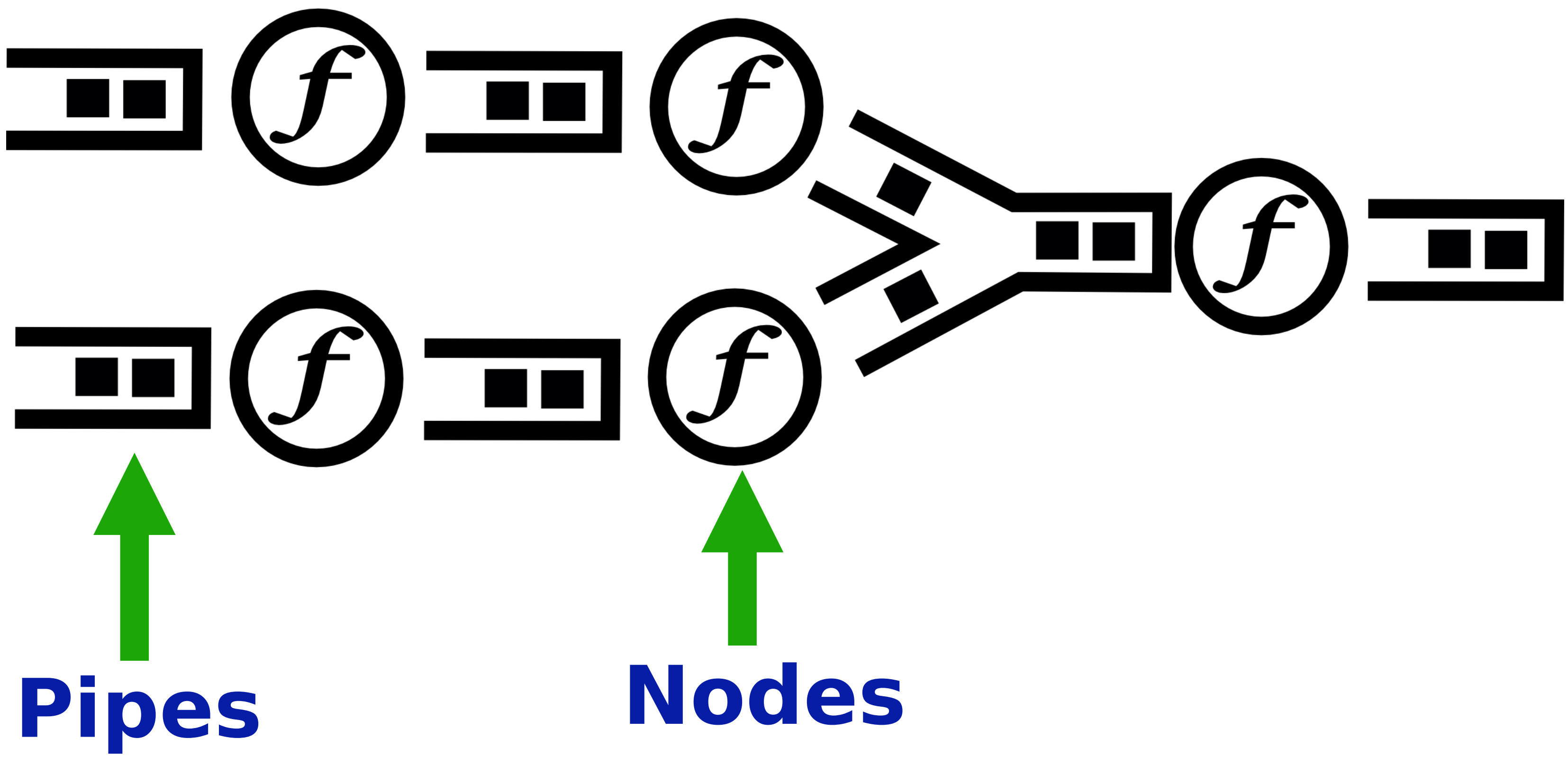
Pipes
- *asynchronous*
- *buffered*
- *EOF*

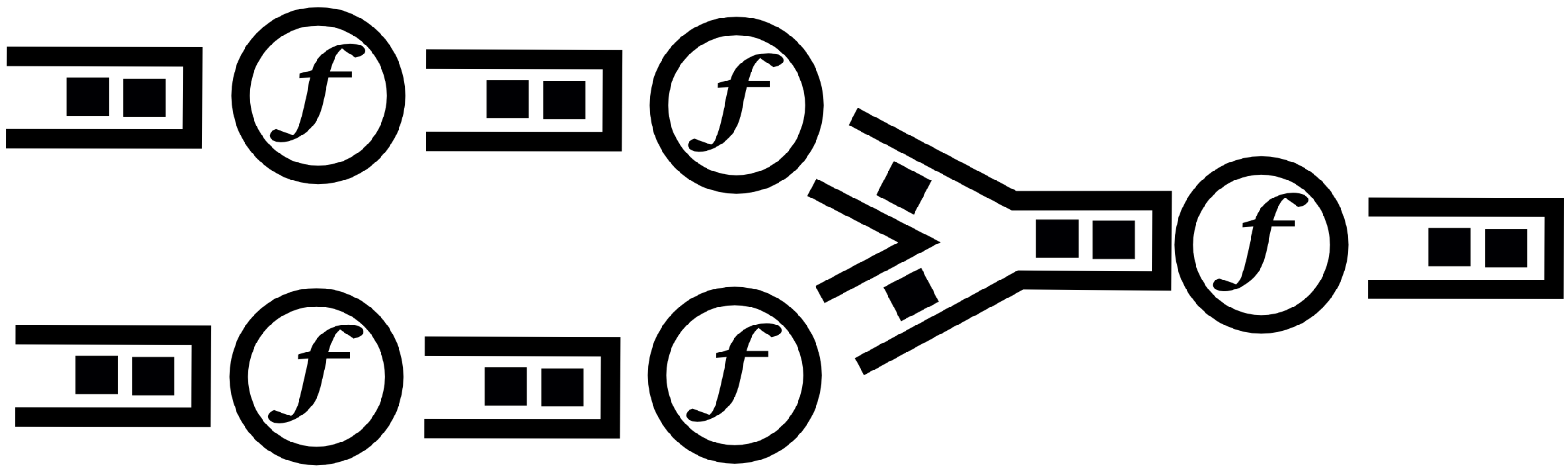


Operators
- standard library
- extensible
Syntax
- compact
.org/wiki/Pipeline_(Unix)" 2>/dev/null |
sed 's/[^a-zA-Z]/ /g' |
tr 'A-Z ' 'a-z\n' |
grep '[a-z]' |
sort -u |
comm -23 - <(sort /usr/share/dict/words) |
less
Processes
- concurrent
- separate address spaces
- multi-threaded
- stdin, stdout, stderr
Pipes
- asynchronous
- buffered
- EOF
Execution Environment

Pipes & Nodes

We have built a Clojure library that provides communication pipes and processing nodes for representing stream processing.





What syntax?

http://lib.store.yahoo.net/lib/paulgraham/jmc.lisp

```
; The Lisp defined in McCarthy's 1960 paper, translated into CL.
; Assumes only quote, atom, eq, cons, car, cdr, cond.
; Bug reports to lispcode@paulgraham.com.

(defun null. (x)
  (eq x '()))

(defun and. (x y)
  (cond (x (cond (y 't) ('t '())))
        ('t '())))

(defun not. (x)
  (cond (x '())
        ('t 't)))

(defun append. (x y)
  (cond ((null. x) y)
        ('t (cons (car x) (append. (cdr x) y)))))

(defun list. (x y)
  (cons x (cons y '())))

(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
        ((and. (not. (atom x)) (not. (atom y)))
         (cons (list. (car x) (car y))
               (pair. (cdr x) (cdr y))))))

(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))

(defun eval. (e a)
  (cond
    ((atom e) (assoc. e a))
    ((atom (car e))
     (cond
       ((eq (car e) 'quote) (cadr e))
       ((eq (car e) 'atom)  (atom   (eval. (cadr e) a)))
       ((eq (car e) 'eq)    (eq     (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'car)   (car    (eval. (cadr e) a)))
       ((eq (car e) 'cdr)   (cdr    (eval. (cadr e) a)))
       ((eq (car e) 'cons)  (cons   (eval. (cadr e) a)
                                    (eval. (caddr e) a)))
       ((eq (car e) 'cond)  (evcon. (cdr e) a))
       ('t (eval. (cons (assoc. (car e) a)
                        (cdr e))
                  a))))
    ((eq (caar e) 'label)
     (eval. (cons (caddar e) (cdr e))
            (cons (list. (cadar e) (car e)) a)))
    ((eq (caar e) 'lambda)
     (eval. (caddar e)
            (append. (pair. (cadar e) (evlis. (cdr e) a))
                     a)))))

(defun evcon. (c a)
  (cond ((eval. (caar c) a)
         (eval. (cadar c) a))
        ('t (evcon. (cdr c) a))))

(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval.  (car m) a)
                  (evlis. (cdr m) a)))))
```

John McCarthy

1927 - 2011

Stream Processing: Syntax and Operations

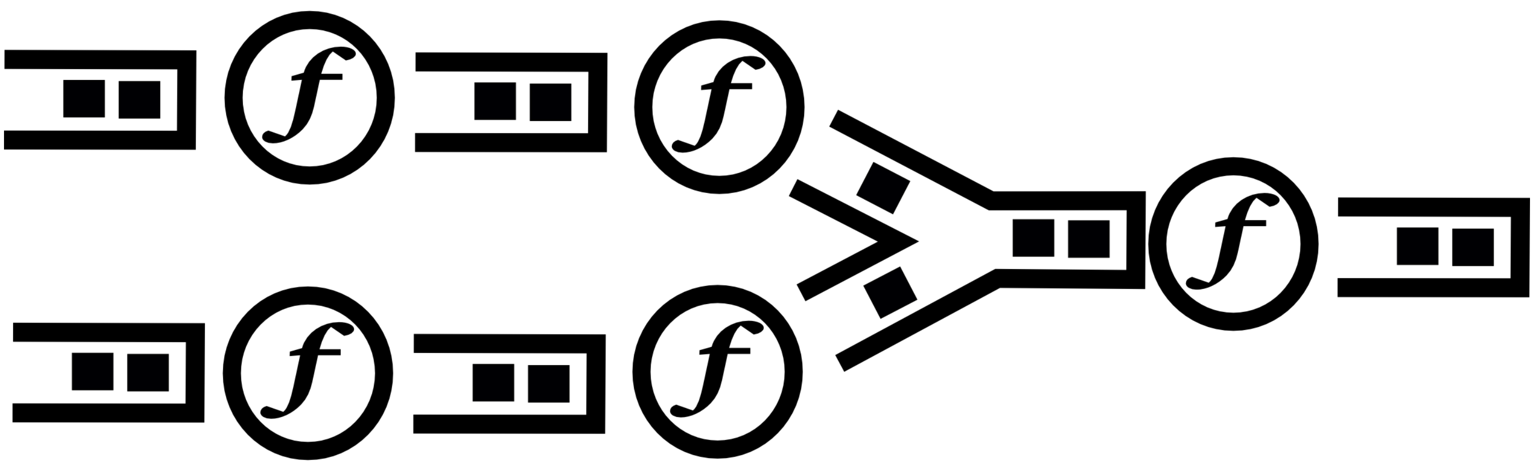
Stream processing expressed as s-expressions using variants of the standard Clojure sequence operations.

```
(preduce+ + 0 +
          (pmap+ (comp count #(split % #" "))
                 (source-data+ [["hello"
                                 "a simple test"]])))
```

Stream Processing: Syntax to Execution Model

Stream processing expressions compile into pipe/node structures.

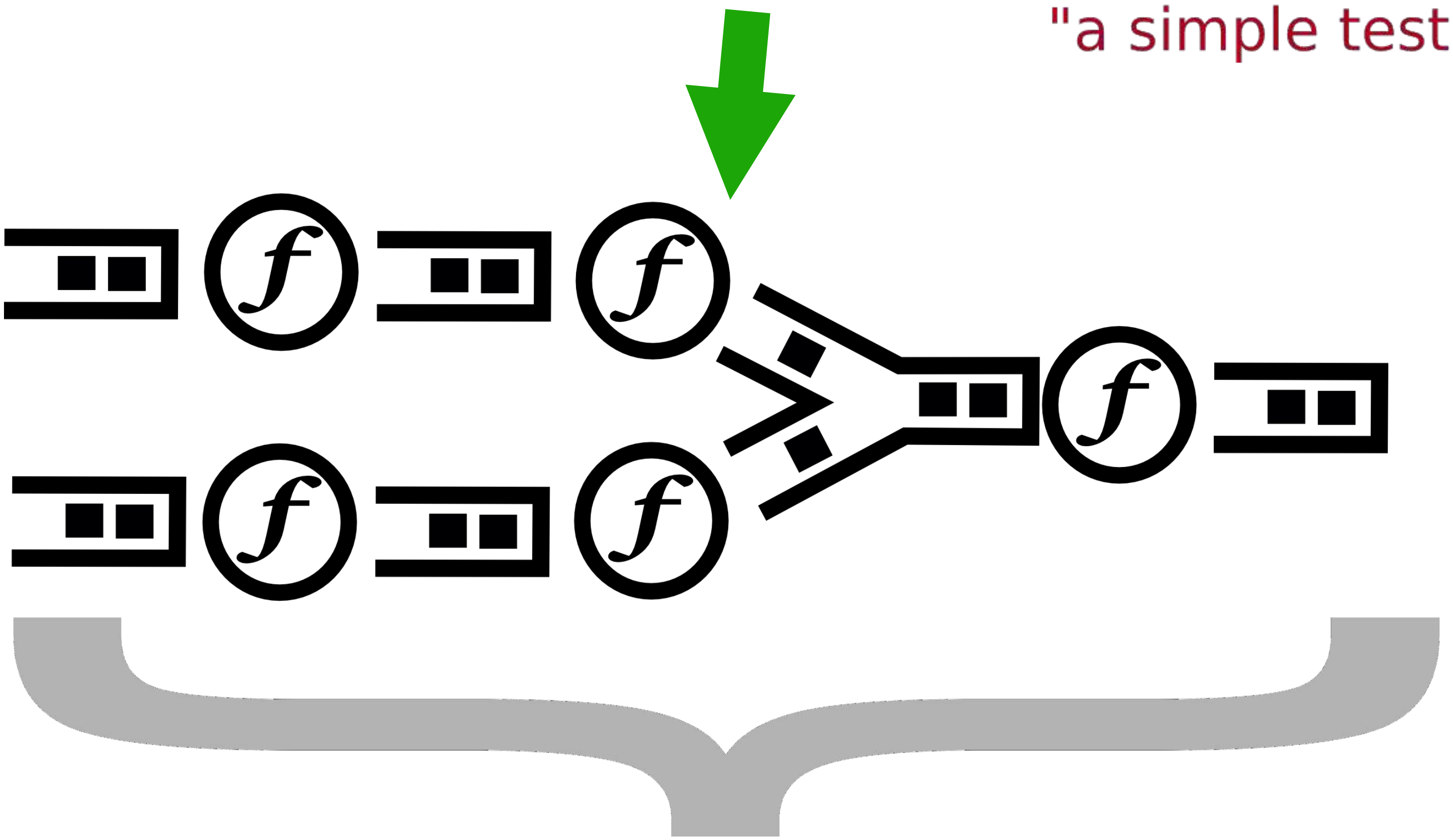
```
(preduce+ + 0 +
              (pmap+ (comp count #(split % #" "))
                     (source-data+ [["hello"
                                     "a simple test"]])))
```



Stream Processing: Execution Environment

Stream processing expressions are executed via Fork/Join.

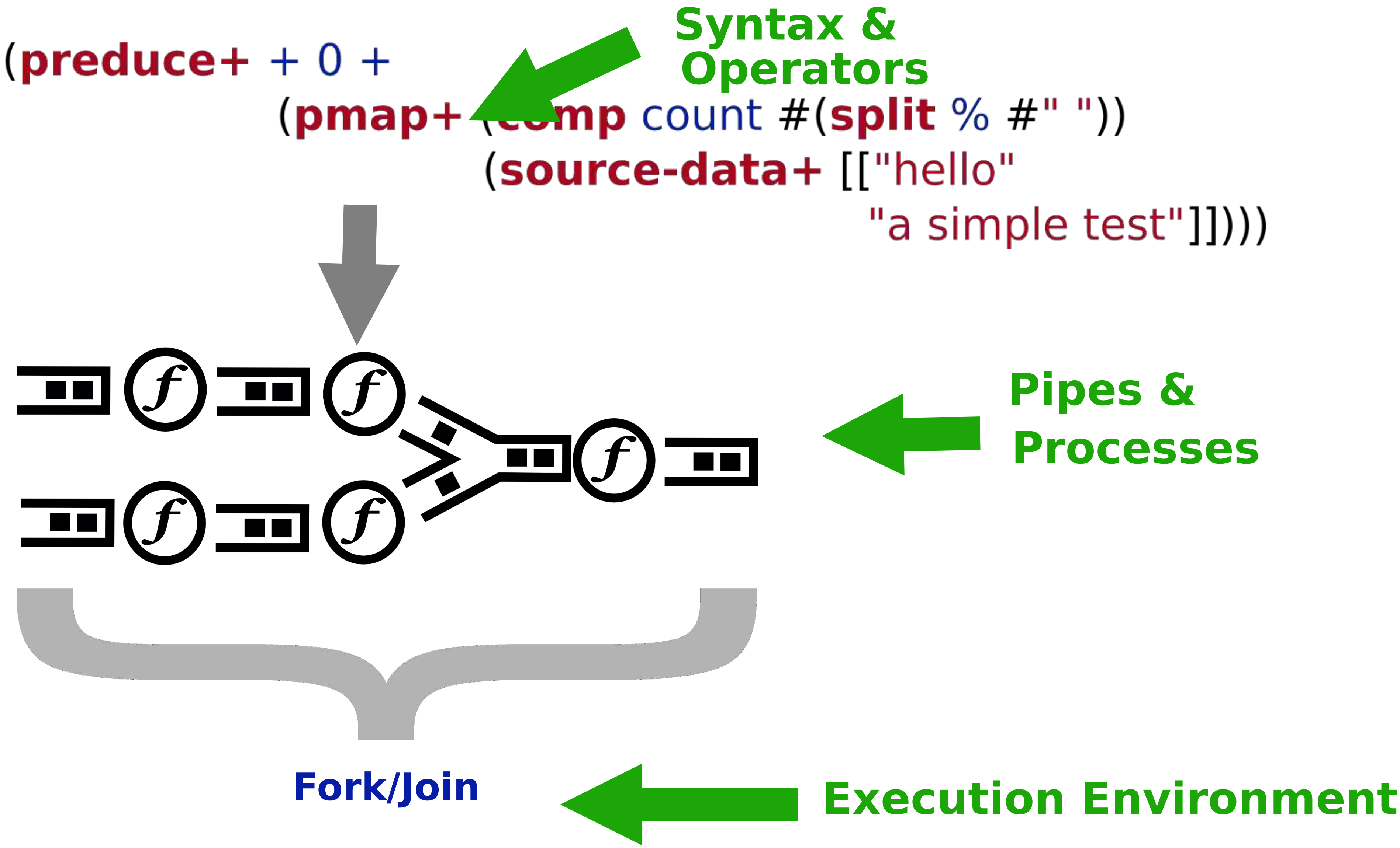
```
(preduce+ + 0 +
                (pmap+ (comp count #(split % #" "))
                       (source-data+ [["hello"
                                       "a simple test"]])))
```



Fork/Join

Stream Processing

The same elements from the Unix pipe example are present here.



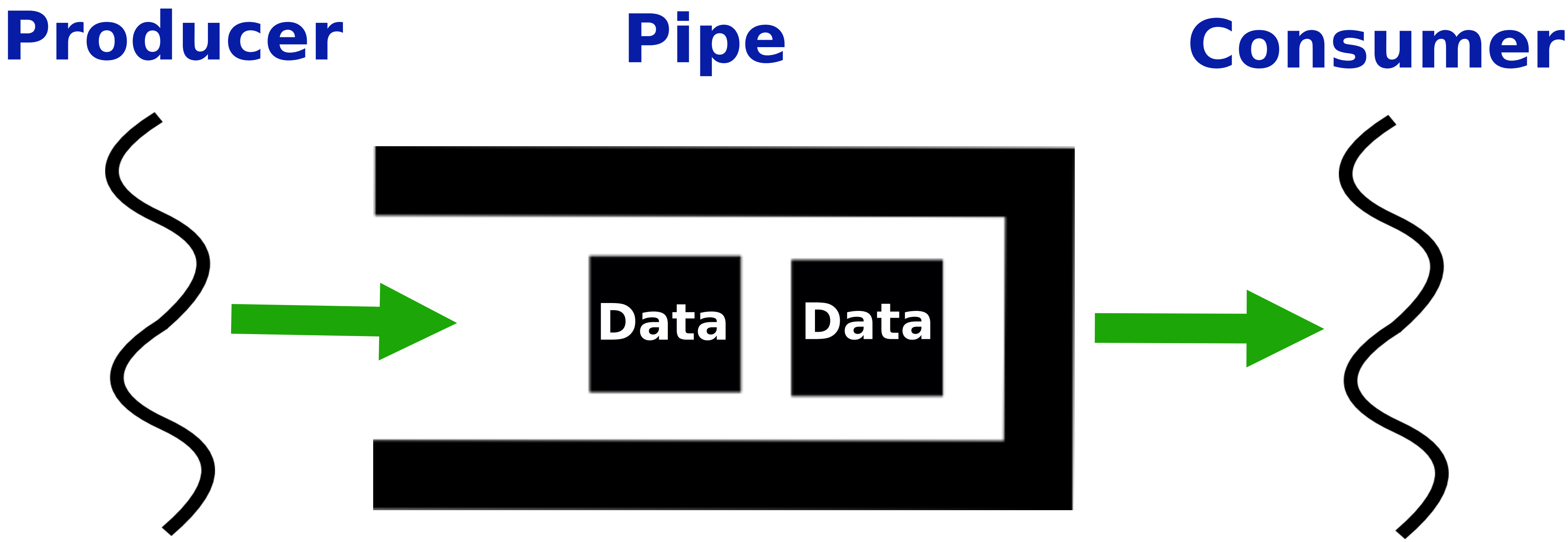
Demo

From the bottom-up,

how would you build this?

Pipes

"Pipes" represent streams of data.



Pipe Operations

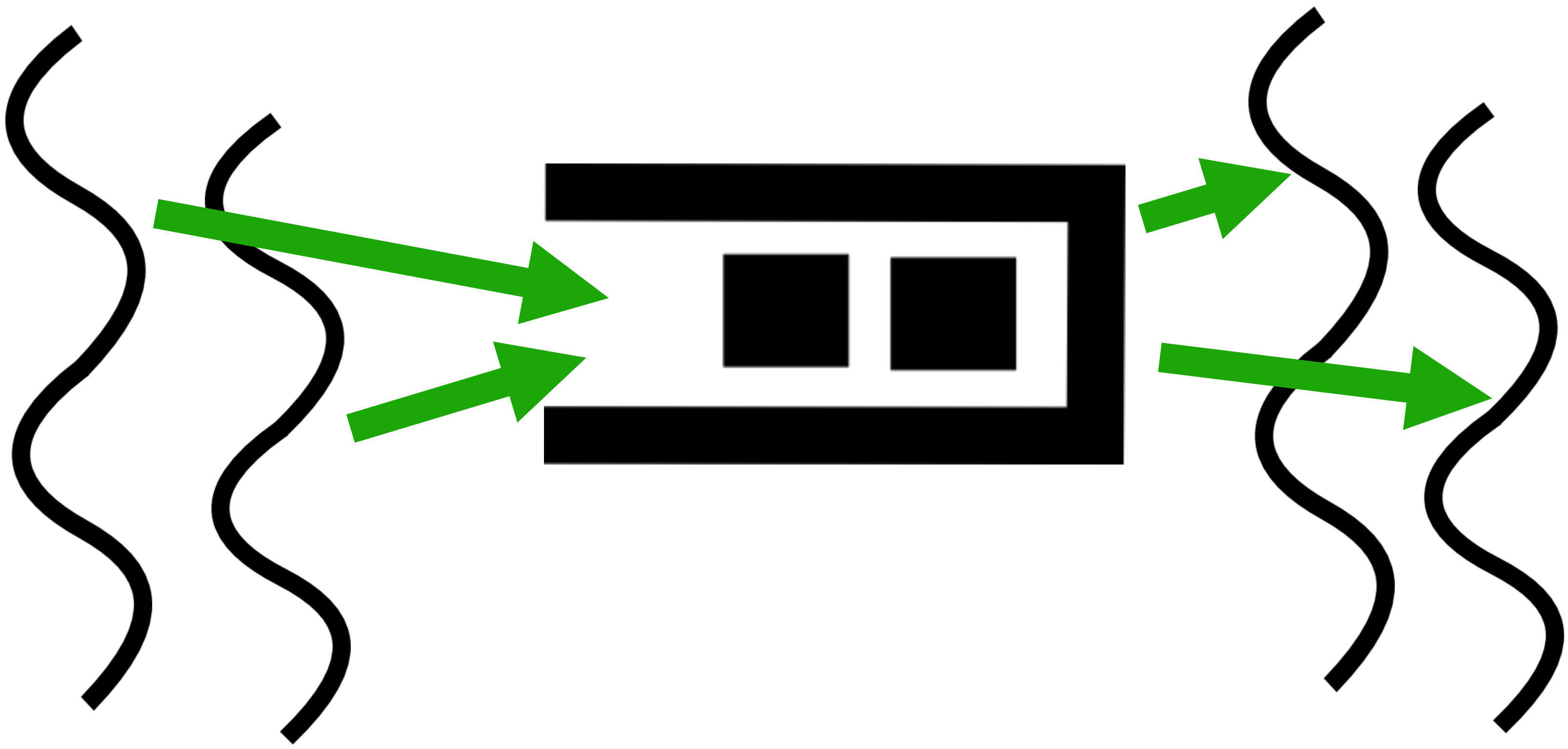
Pipe operations allow data to be "sent" and "received".
Pipes can also be closed, indicating the "end" of the stream.

```
(enqueue pipe item)
(enqueue-all pipe items)
(close pipe)
(error pipe exception)
```

```
(dequeue pipe)
(dequeue-all pipe)
(closed? pipe)
(error? pipe)
```

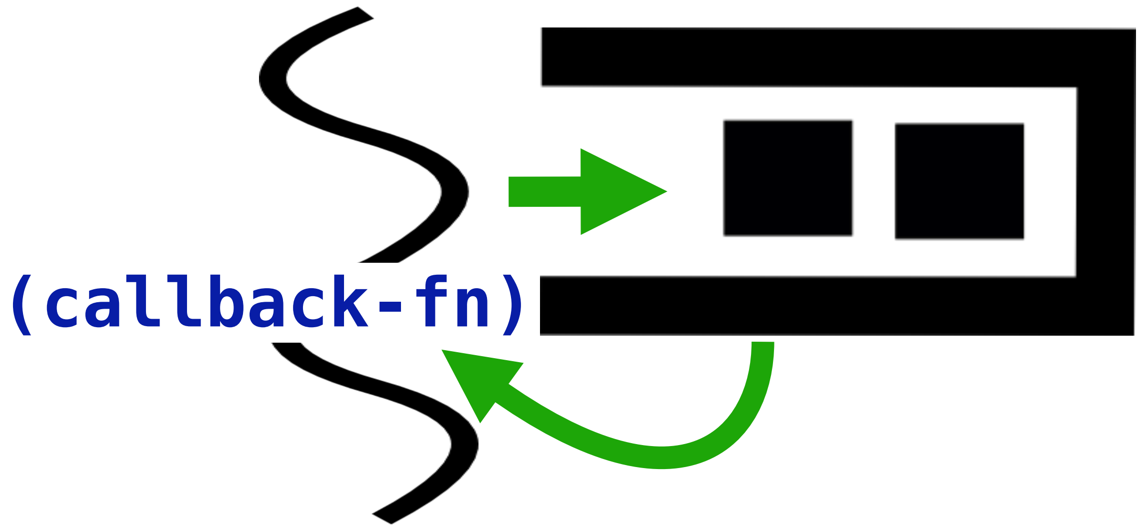
Pipes - Threadsafe

Pipes support multiple producers and consumers.
Items are reliably delivered exactly once.



Pipe Callbacks

Pipes can have associated callback functions.
Callbacks are executed each time an item is enqueued.




```
(add-callback pipe callback-fn)
(clear-callbacks pipe)
```

Pipe Protocol

Pipes are abstracted as a Clojure protocol.

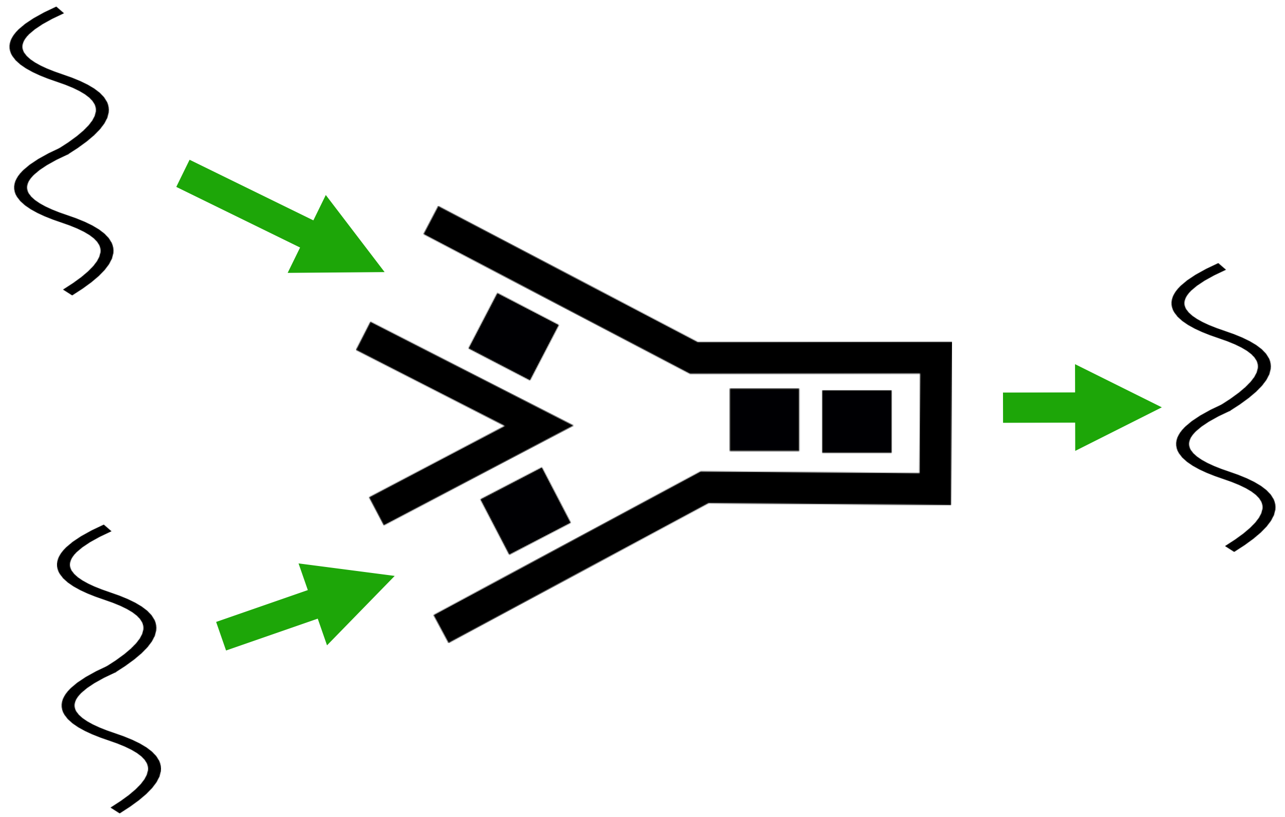
```
(defprotocol Pipe
      (enqueue [pipe item])
      (enqueue-all [pipe items])
      (close [pipe])
      (error [pipe exception])

      (dequeue [pipe])
      (dequeue-all [pipe])
      (closed? [pipe])
      (error? [pipe])

      (add-callback [pipe callback])
      (clear-callbacks [pipe])
      ...)
```

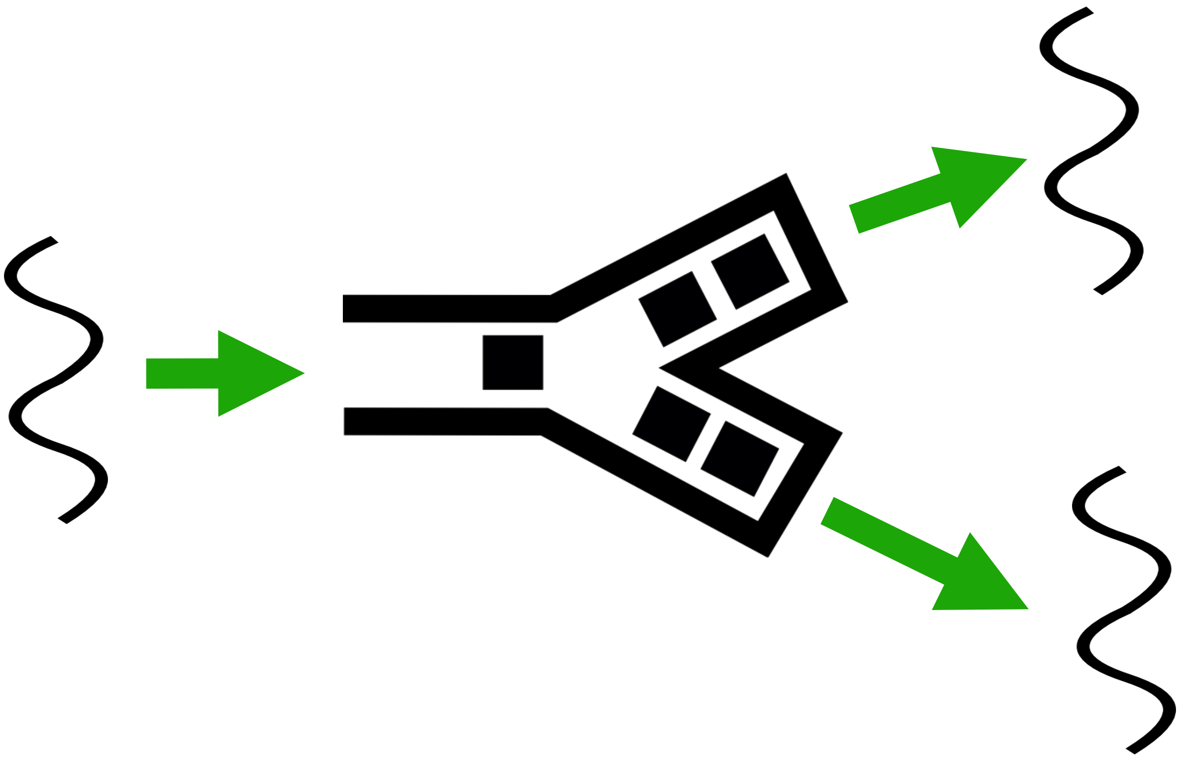
Pipe Multiplexer

Behind the "pipe" abstraction we can implement alternate behavior.
Two input pipes combined into a single underlying pipe.



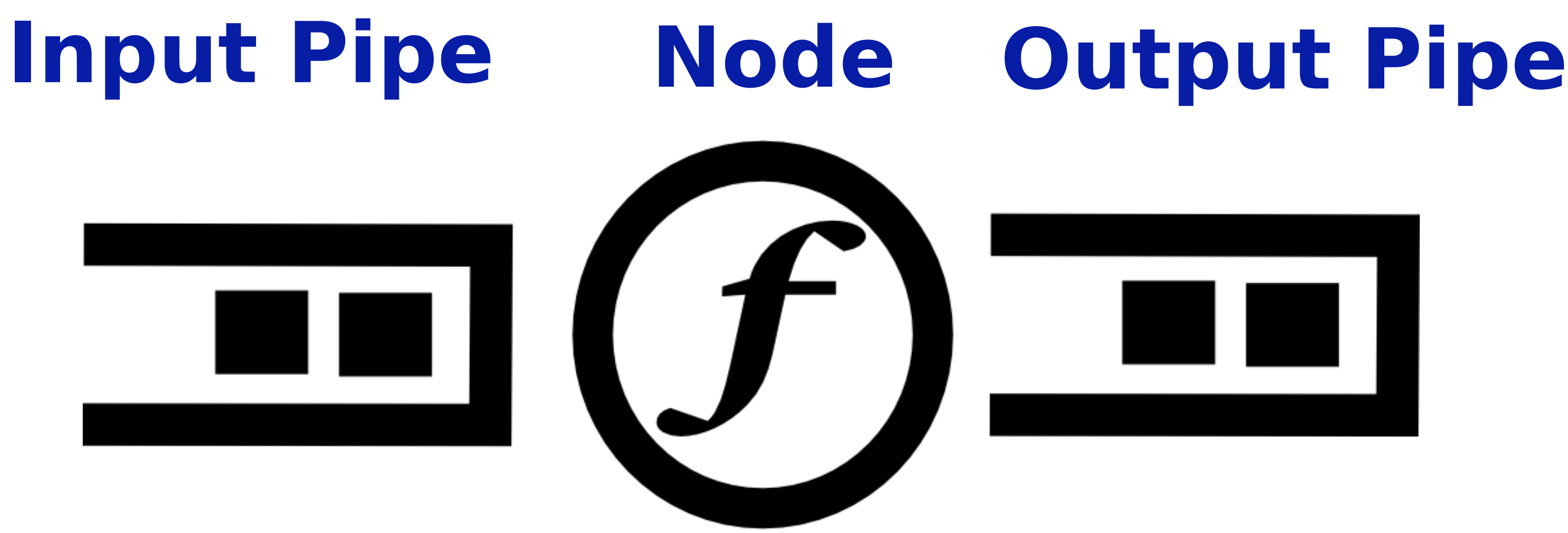
Pipe "tee"

... or a single input pipe that is copied to two destinations.



Processing Nodes

"Pipes" represent streams of data.
"Nodes" represent processing of the data flowing through streams.



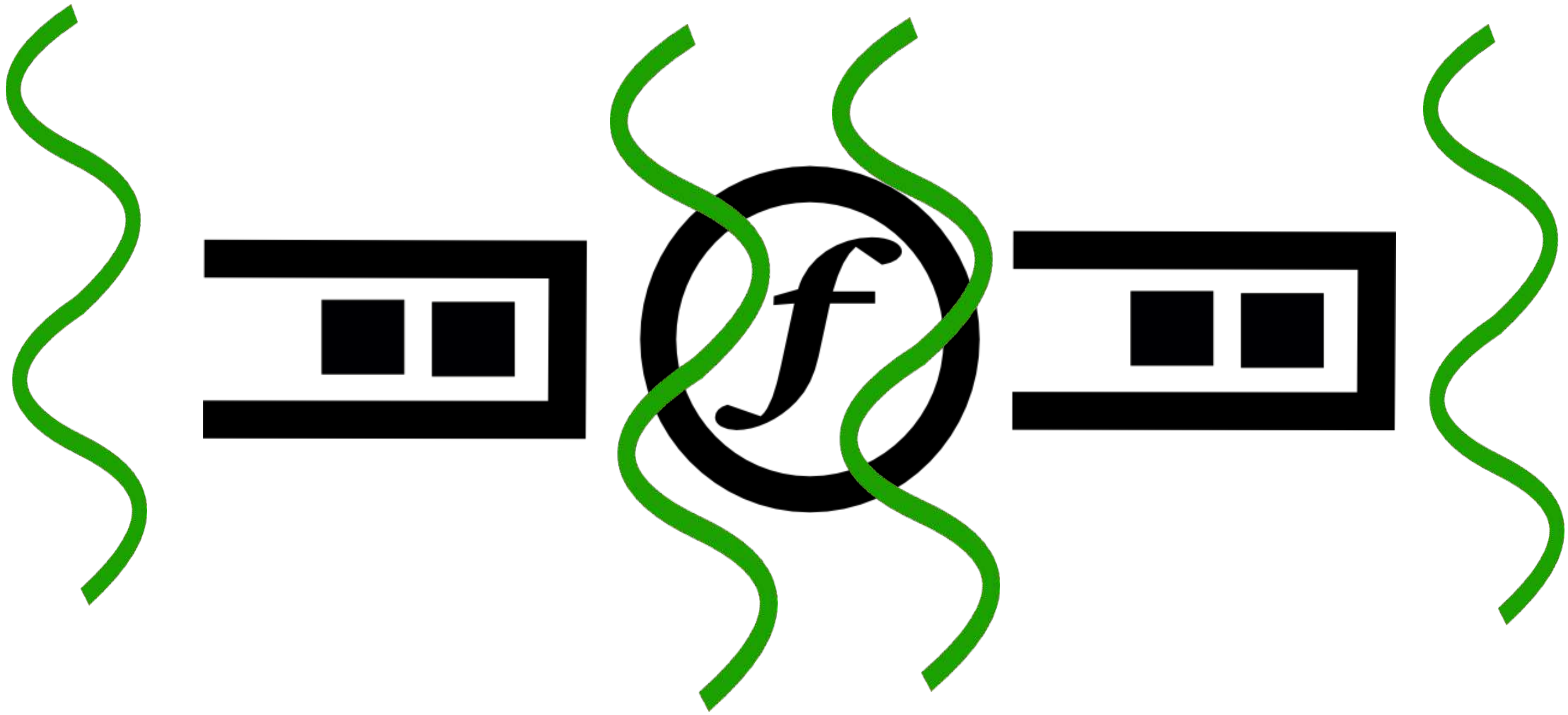
Processing Nodes

Nodes consist of several fields.

```
{:input-pipe ...
 :output-pipe ...
 :task-fn ...
 :state ...
 :concurrency n}
```

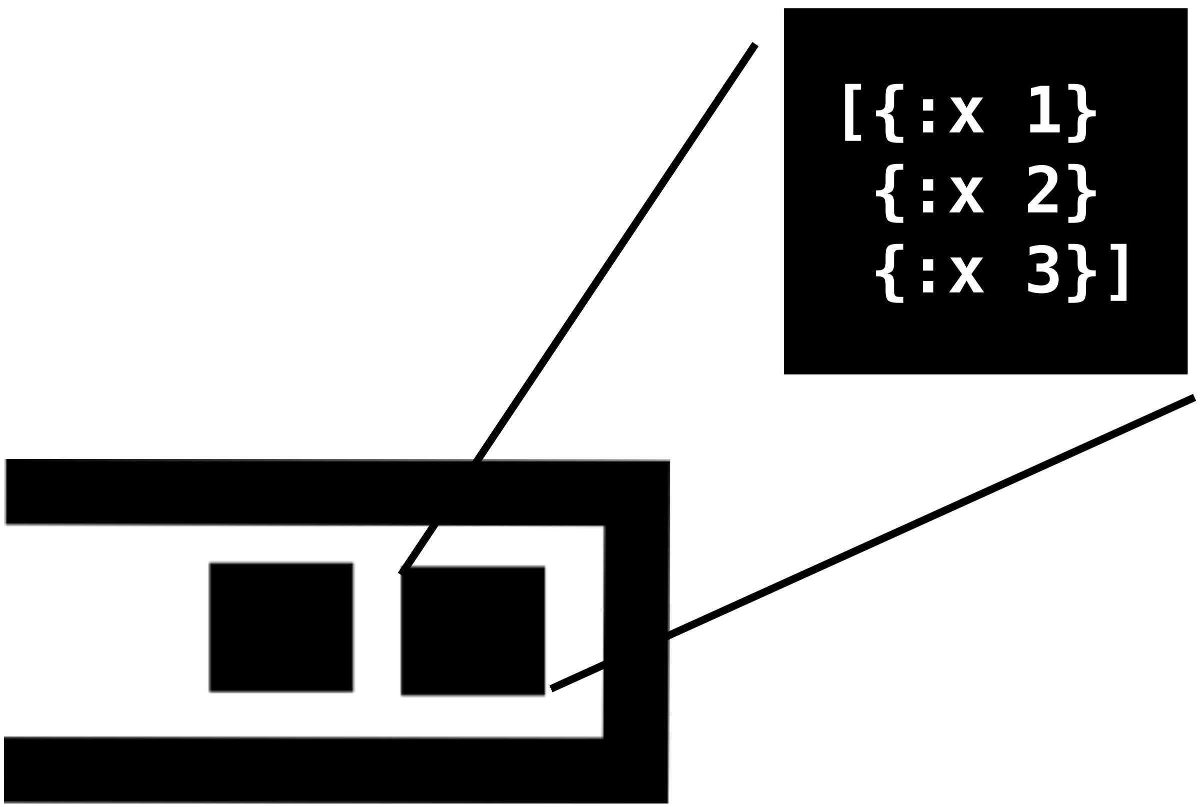
Nodes: Parallel Processing

Nodes are the mechanism for parallel stream processing.



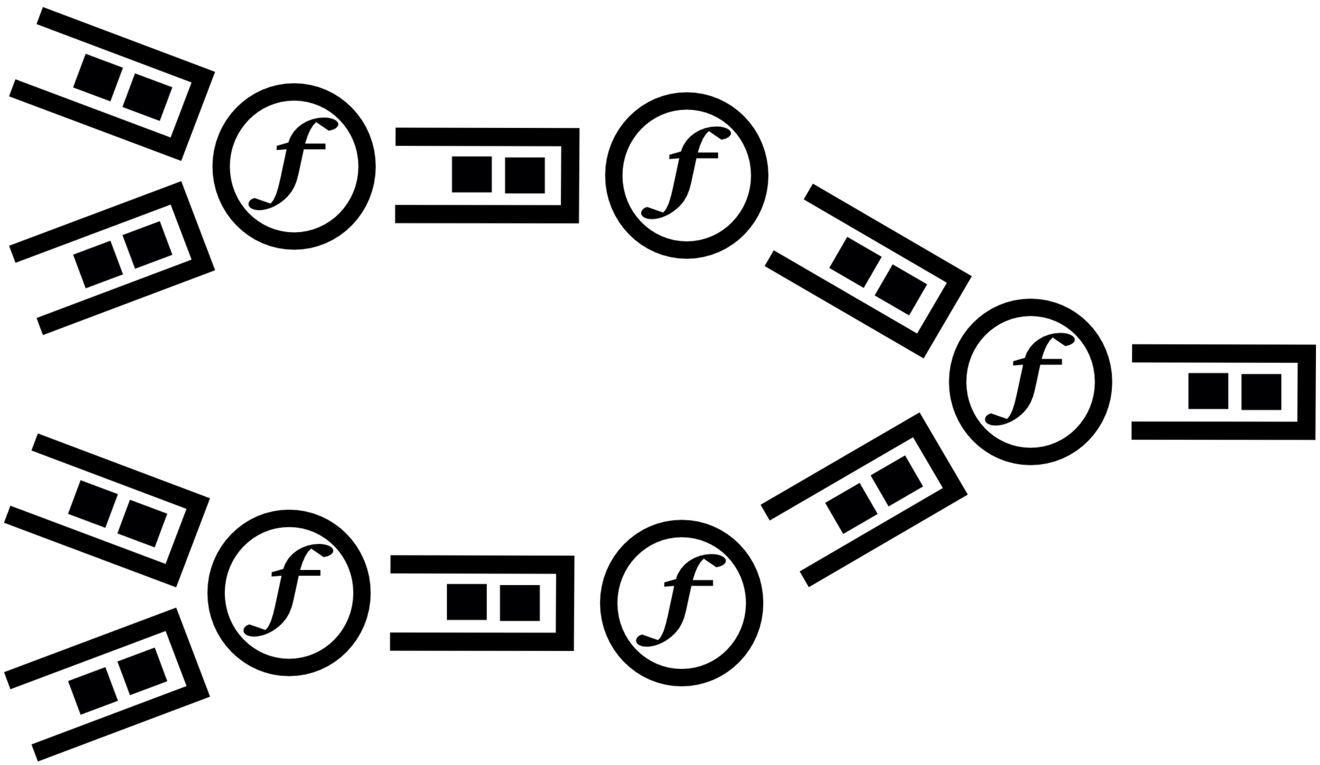
Chunks in Pipes

To reduce overhead, nodes use pipes to transfer "chunks" of items rather than individual items. This can usually be ignored.



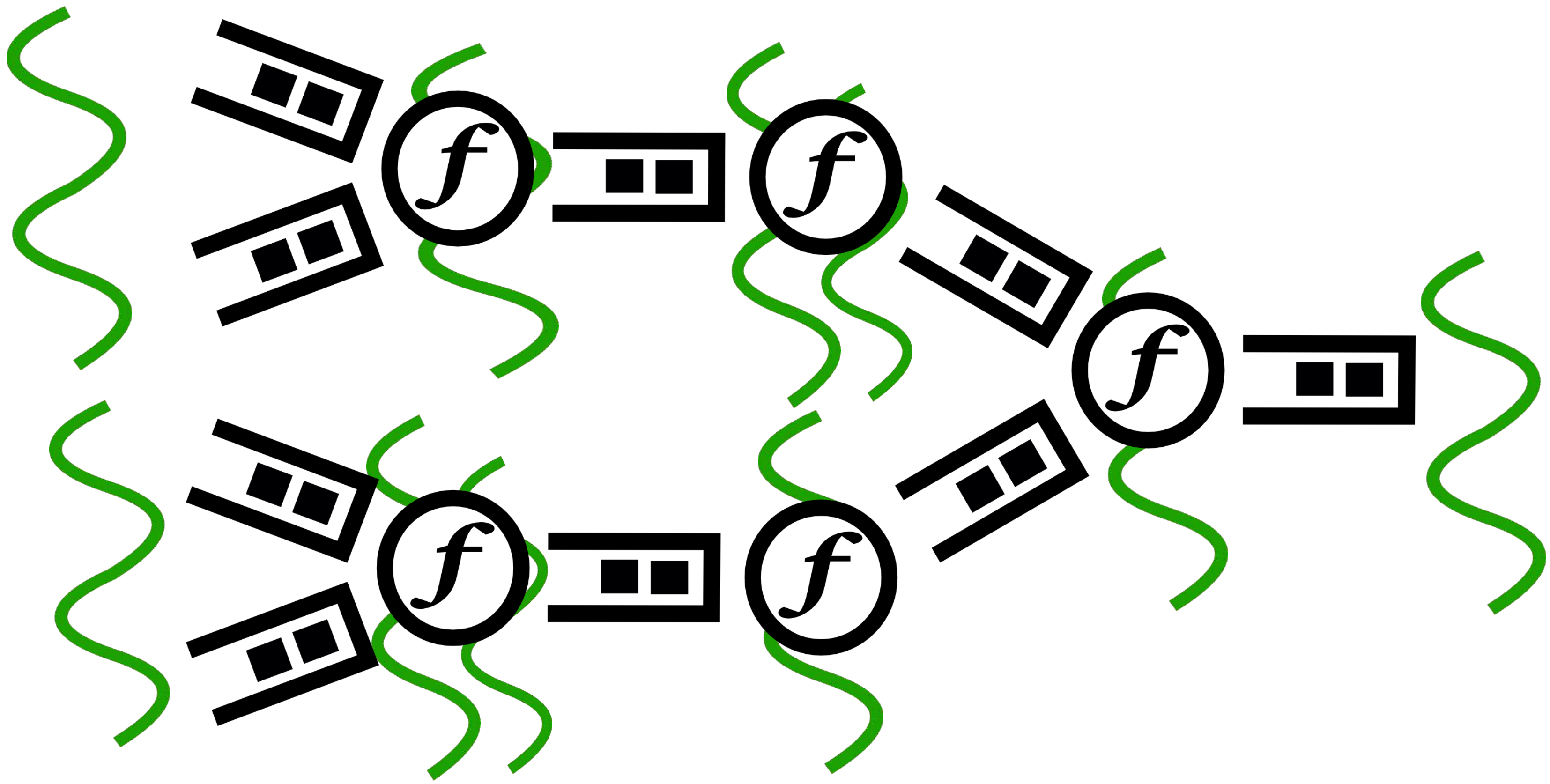
Processing Trees

Nodes and pipes are combined to make trees representing concurrent stream processing.



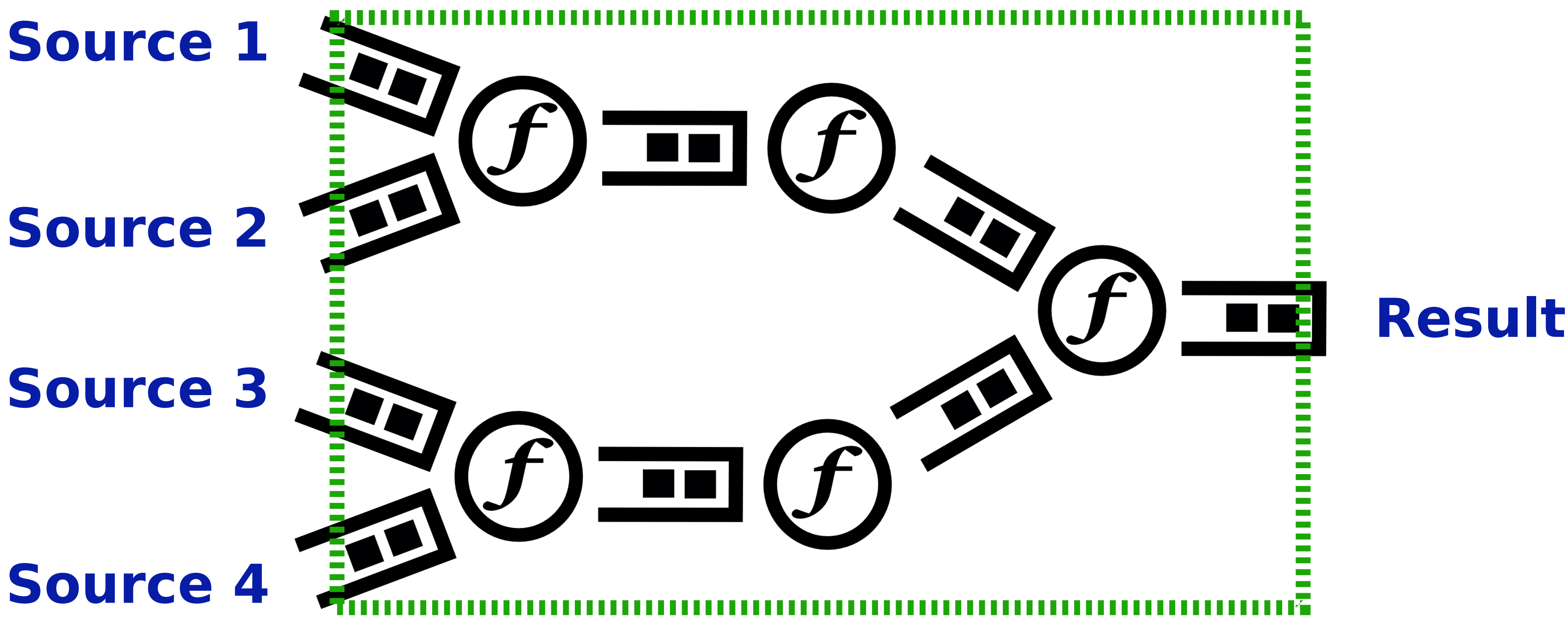
Lots 'o Threads

Nodes and pipes are relatively simple primitives, but in combination can define complex parallel computations.



Data Sources

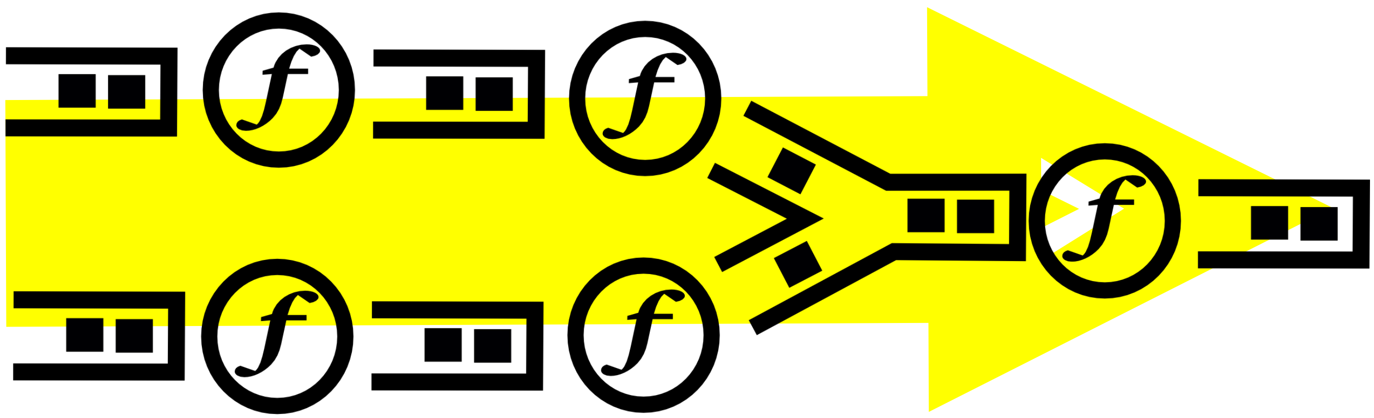
The external view of a processing tree is a box with several input data streams and an output stream.



Is the data pushed or pulled through the tree?

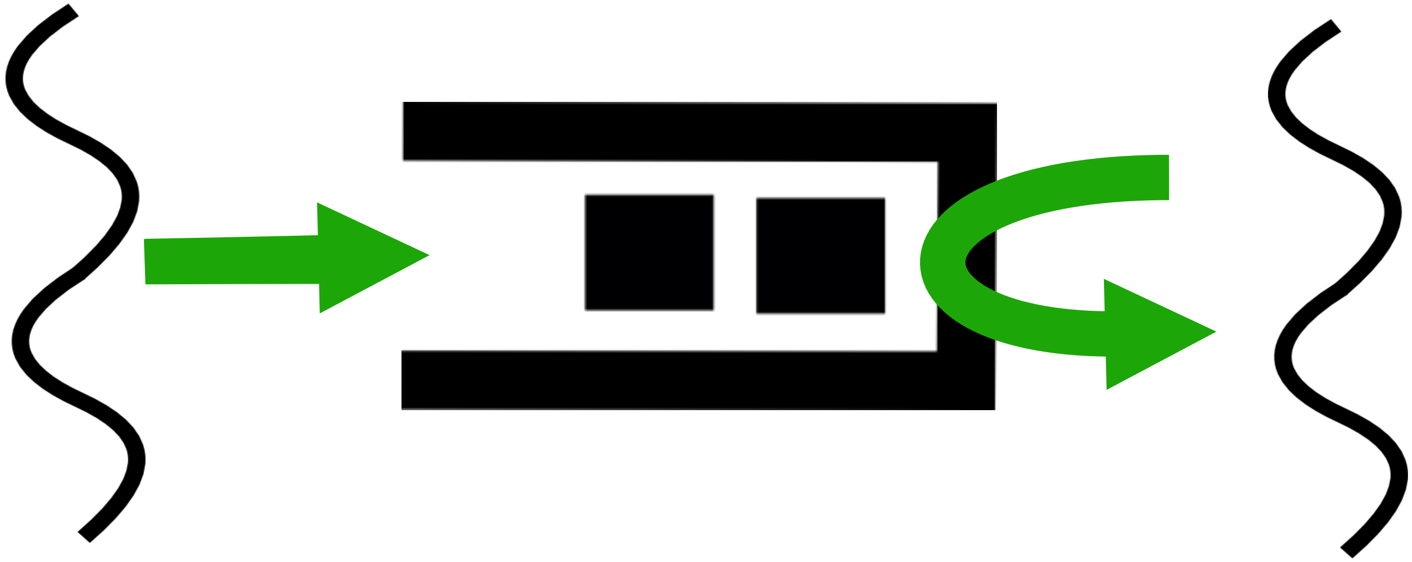
Direction of Data Flow

The data flows left-to-right, but is it pushed or pulled?



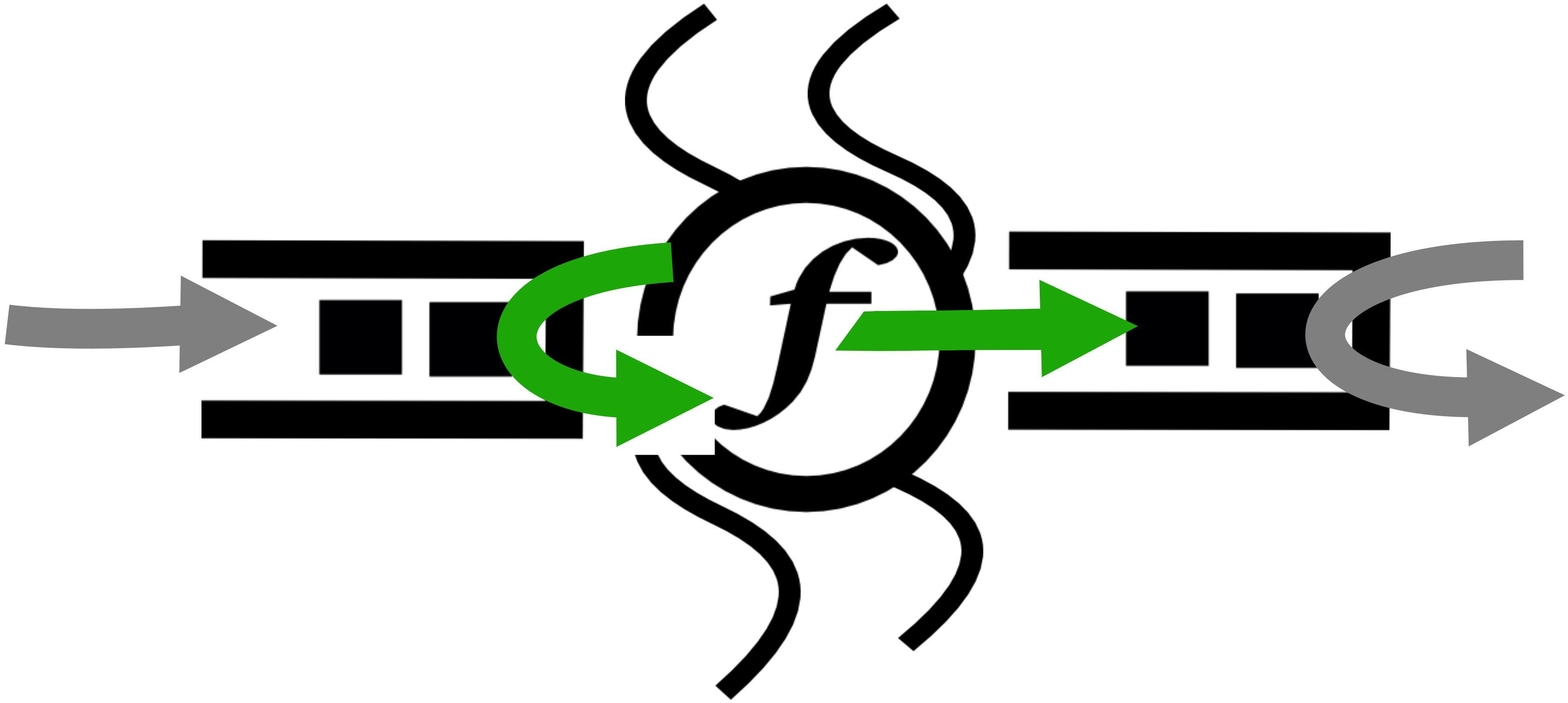
Pipes - Push & Pull

Input to a pipe is provided via push, output is consumed via pull.



Nodes - Pull & Push

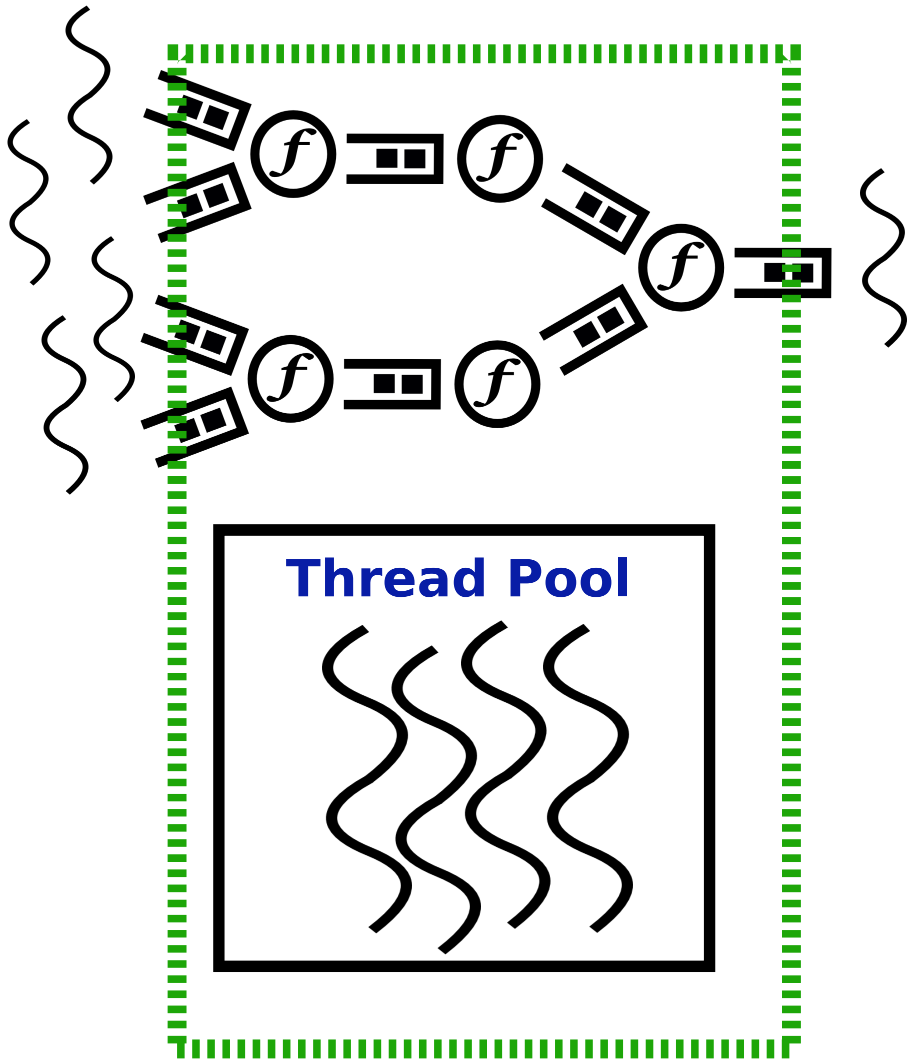
Each node pulls data from its input pipe and pushes data to its output pipe.



How do we get threads to run the node tasks?

Efficient Use of Worker Threads

Worker threads are not bothered until there is data "in-hand" to be processed.

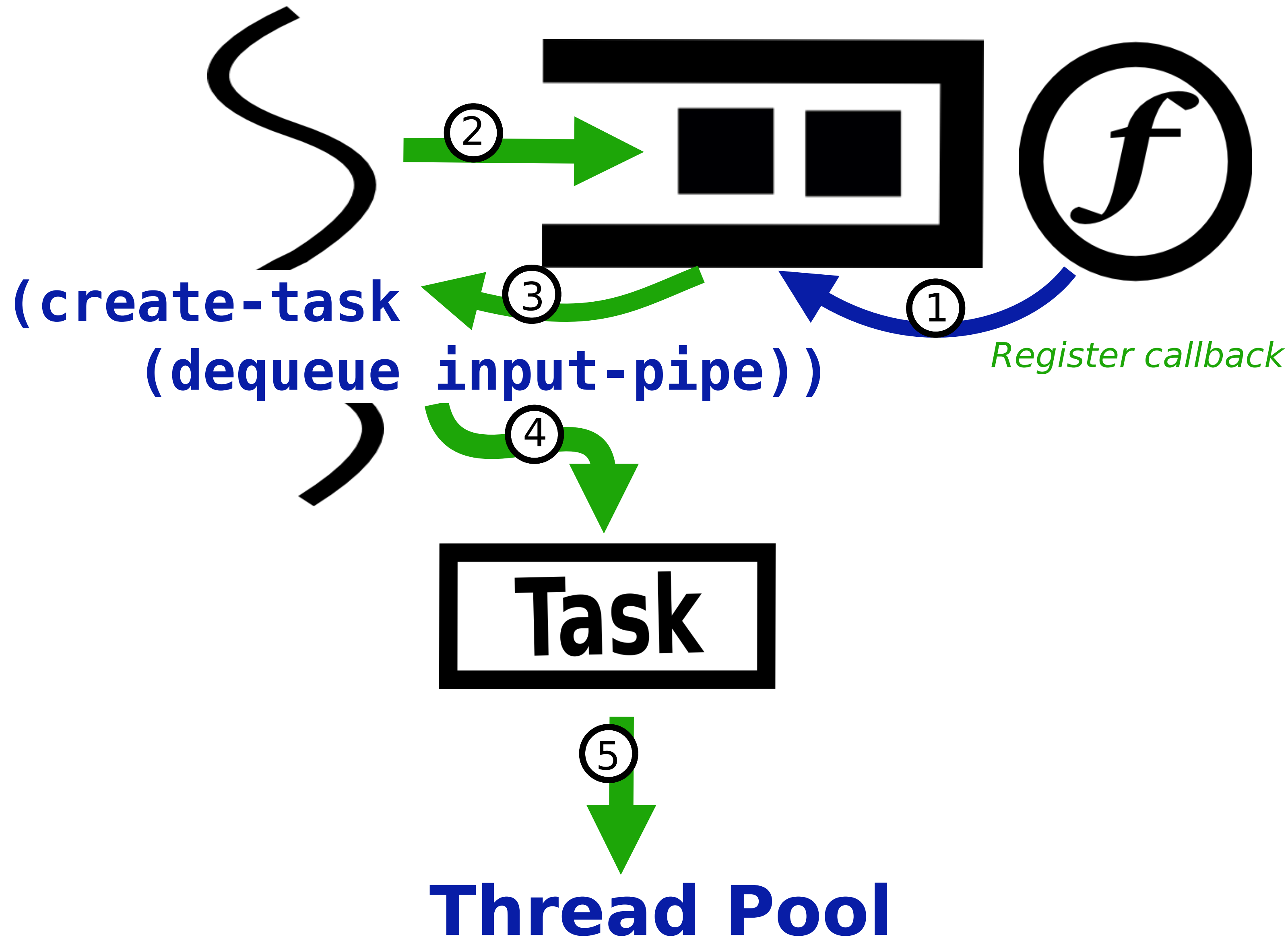


Assume:
- source threads are running
- final consumer thread running
- we have a worker thread pool to use

Constraints:
- worker threads don't block waiting for source data
- worker threads don't poll looking for work

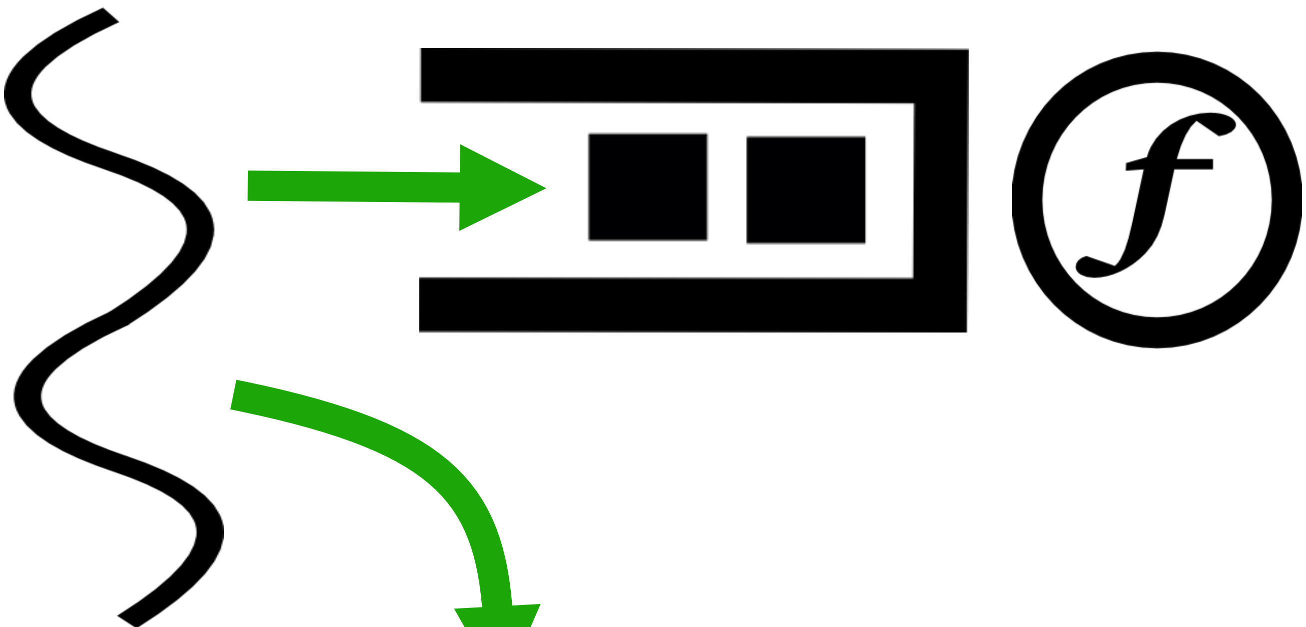
Generating Tasks

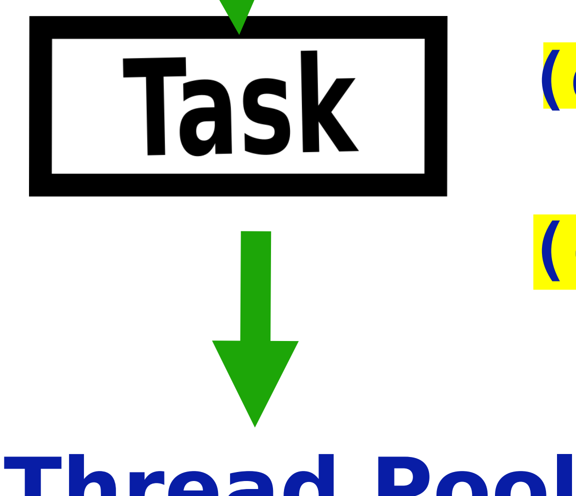
When pipes are wired to nodes, a callback fn is added to the pipe.
The callback fn is run in the enqueuing thread.



Generating Tasks

Tasks run the nodes' task-fn, enqueue the results, and on completion can schedule a new task for the node.

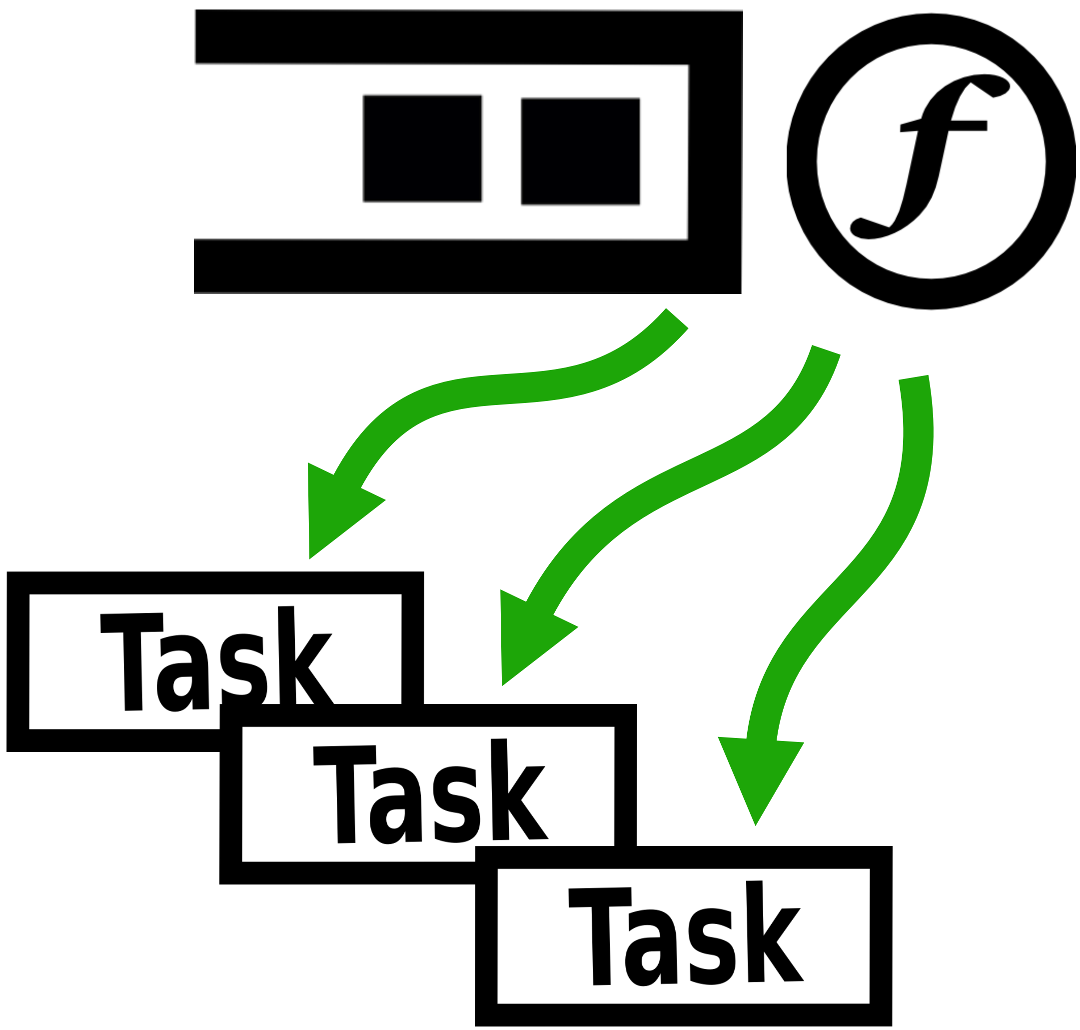





```
(enqueue output-pipe
   (task-fn item))
(create-task
   (dequeue input-pipe))
```

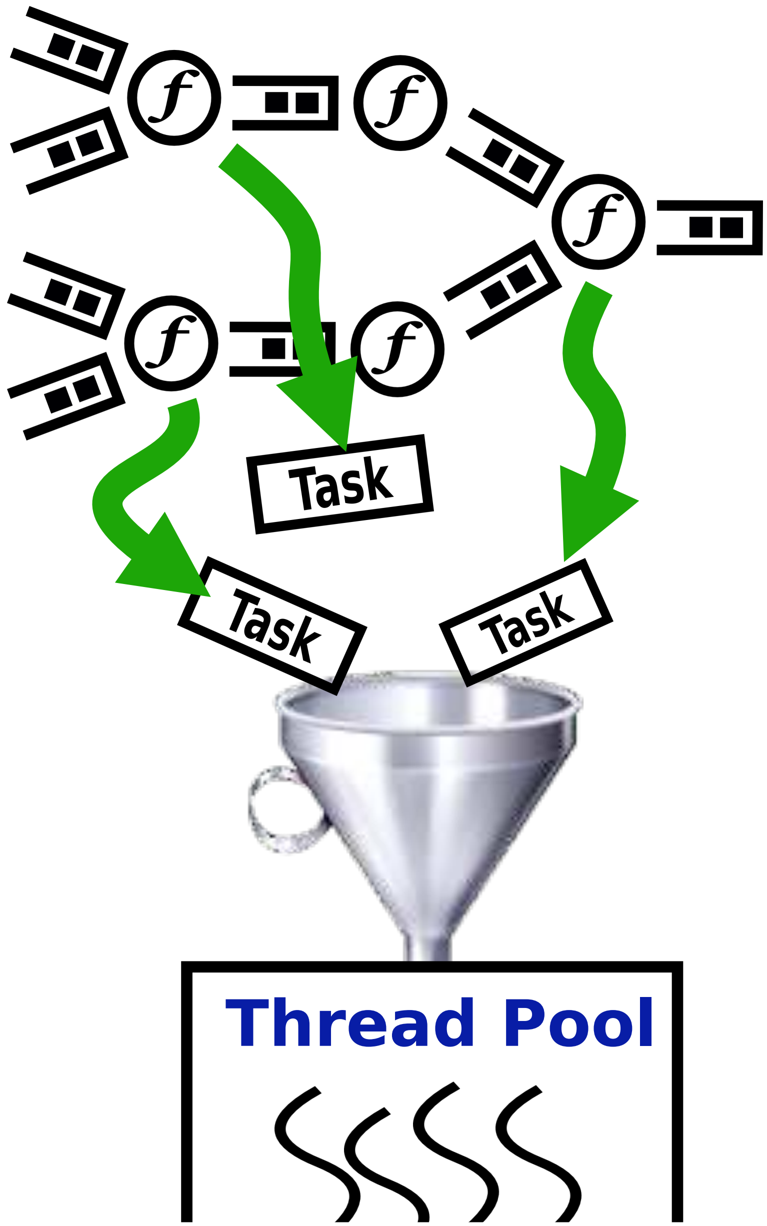
Generating Tasks

Number of concurrent tasks per node limited by the :concurrency of the node.



Generating Tasks

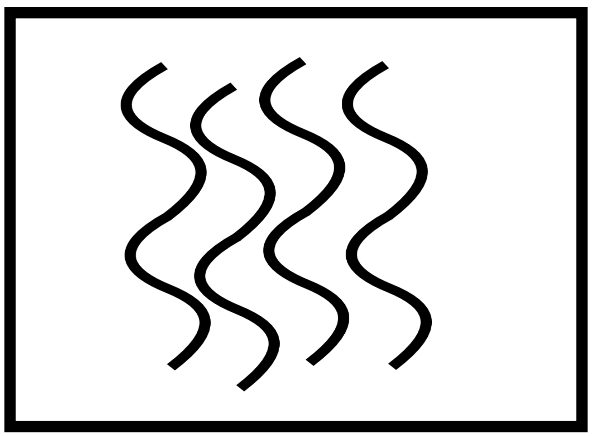
Tasks are generated per node as data is available.
All these tasks are fed into the worker thread pool.



Java Fork/Join

The "Thread Pool" that we use is a ForkJoinPool.

`jsr166y.ForkJoinPool`



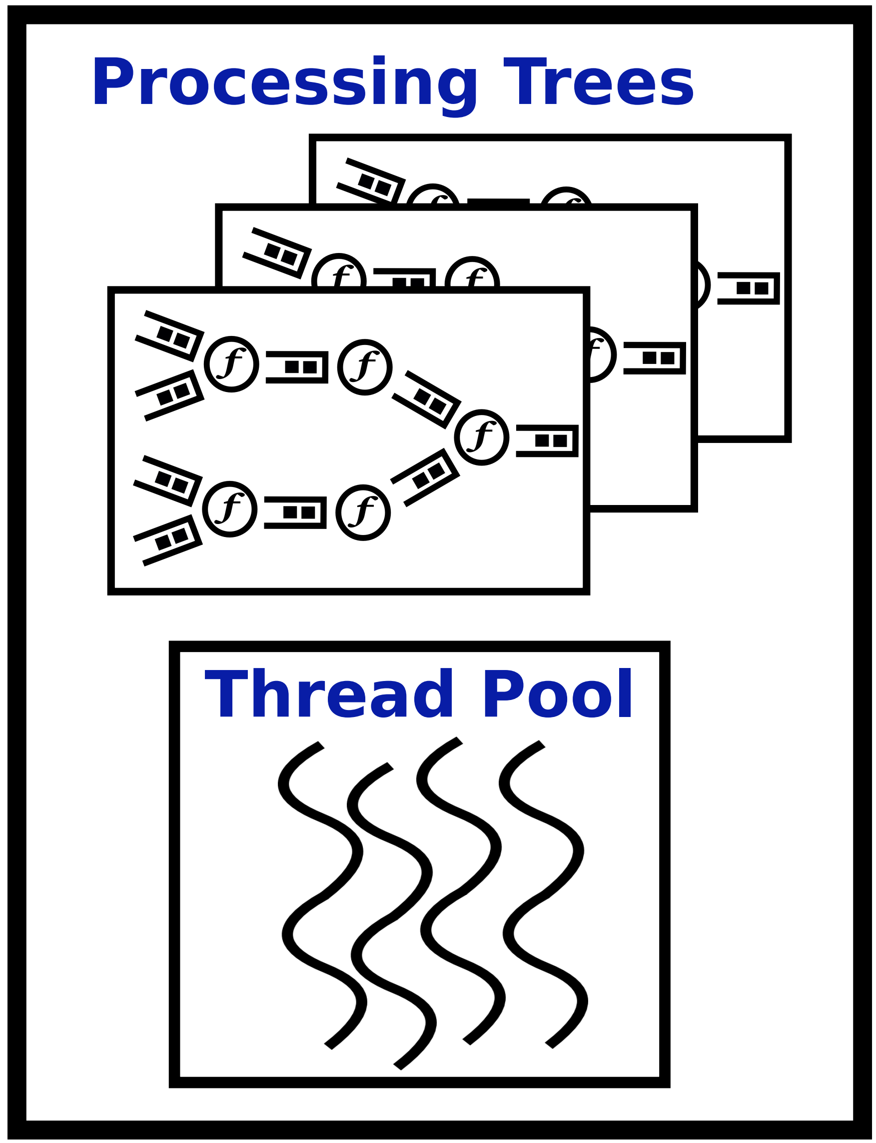
Benefits:

- avoid contention for a single work queue
- use the ManagedBlocker feature to avoid losing threads when blocking
- taps much specialized work on keeping threads hot, minimizing context switches, etc.

Processor

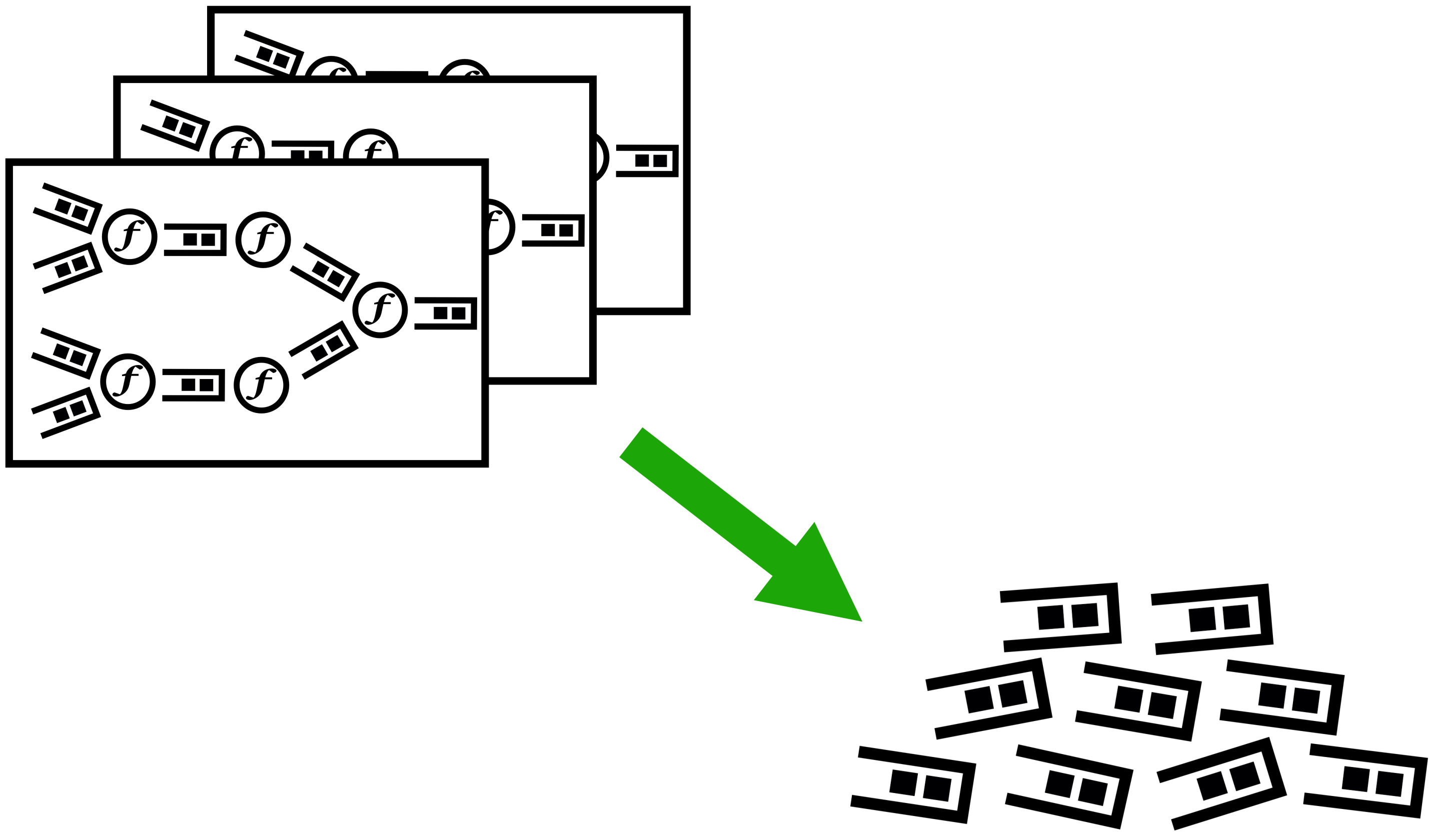
A "processor" consists of a thread pool and many processing trees to be executed.

```
(register processor
    processing-tree)
```

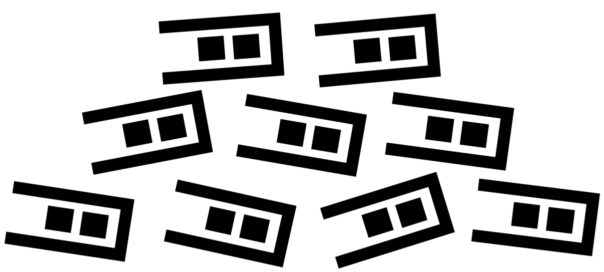


How do we handle large streams?

All of these pipes are on the heap.

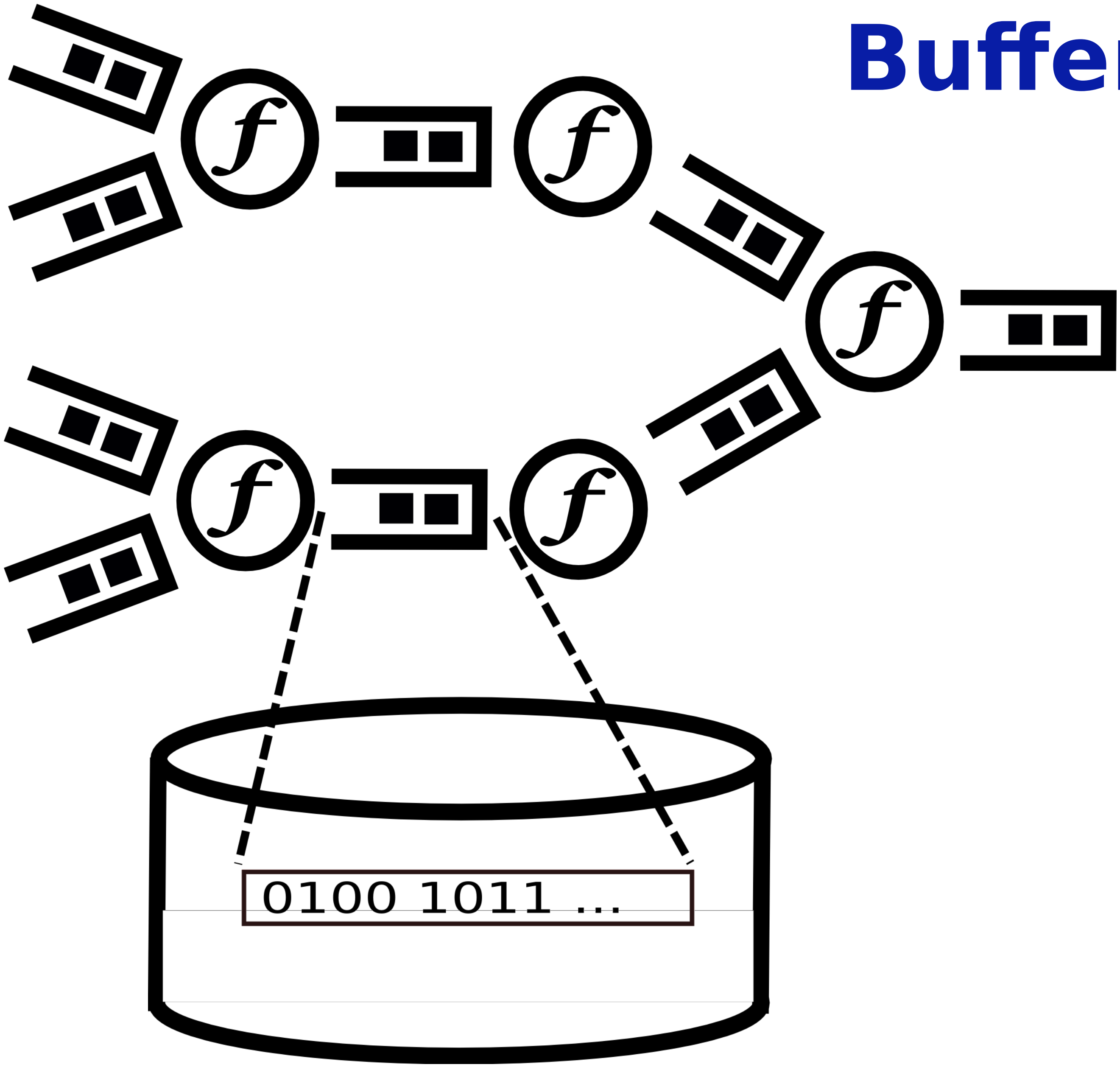


What if we run out of heap space?



> max heap size

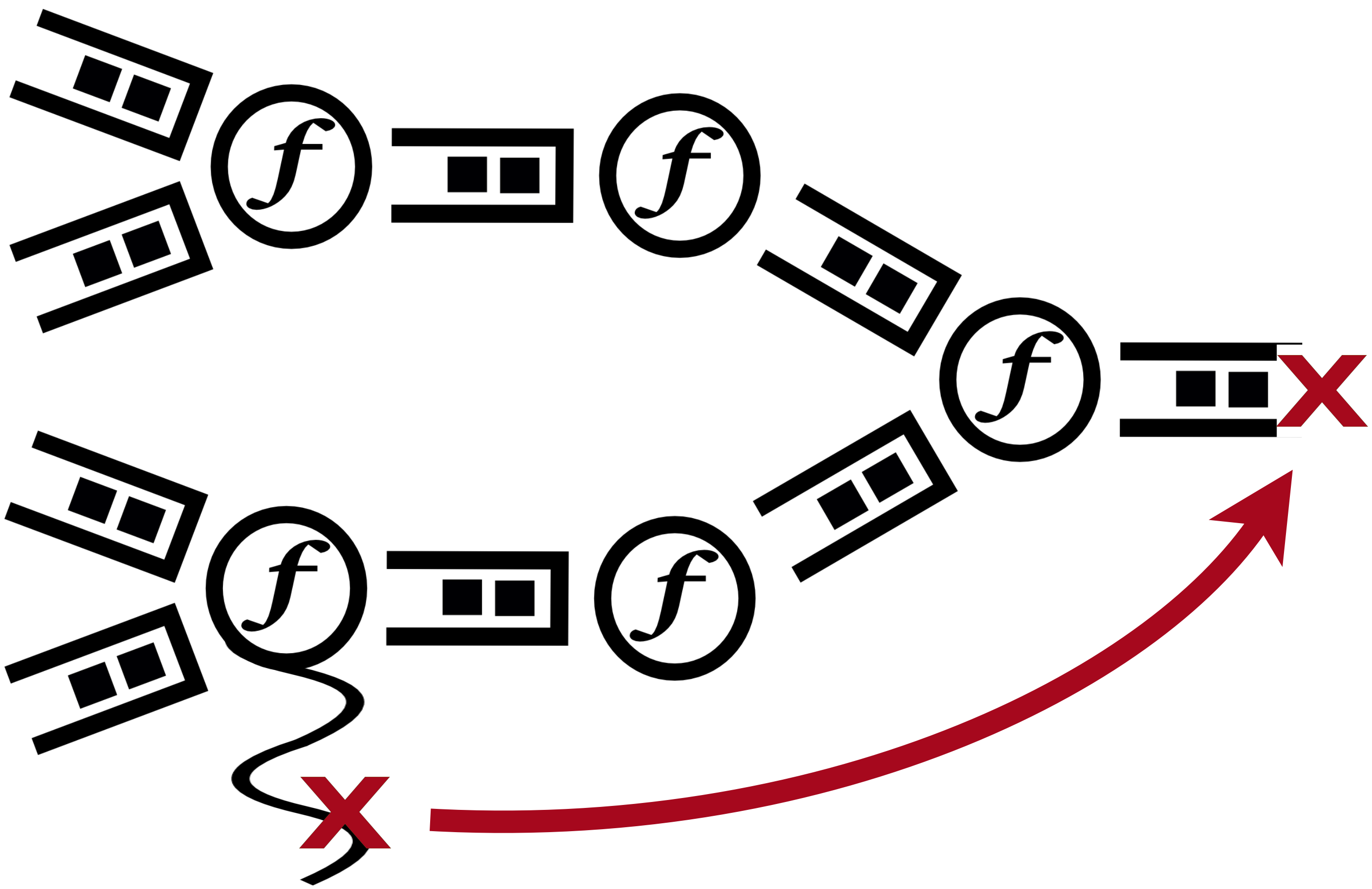
Buffered Pipes



Data flowing through processing trees can be buffered to disk when available heap space is low.

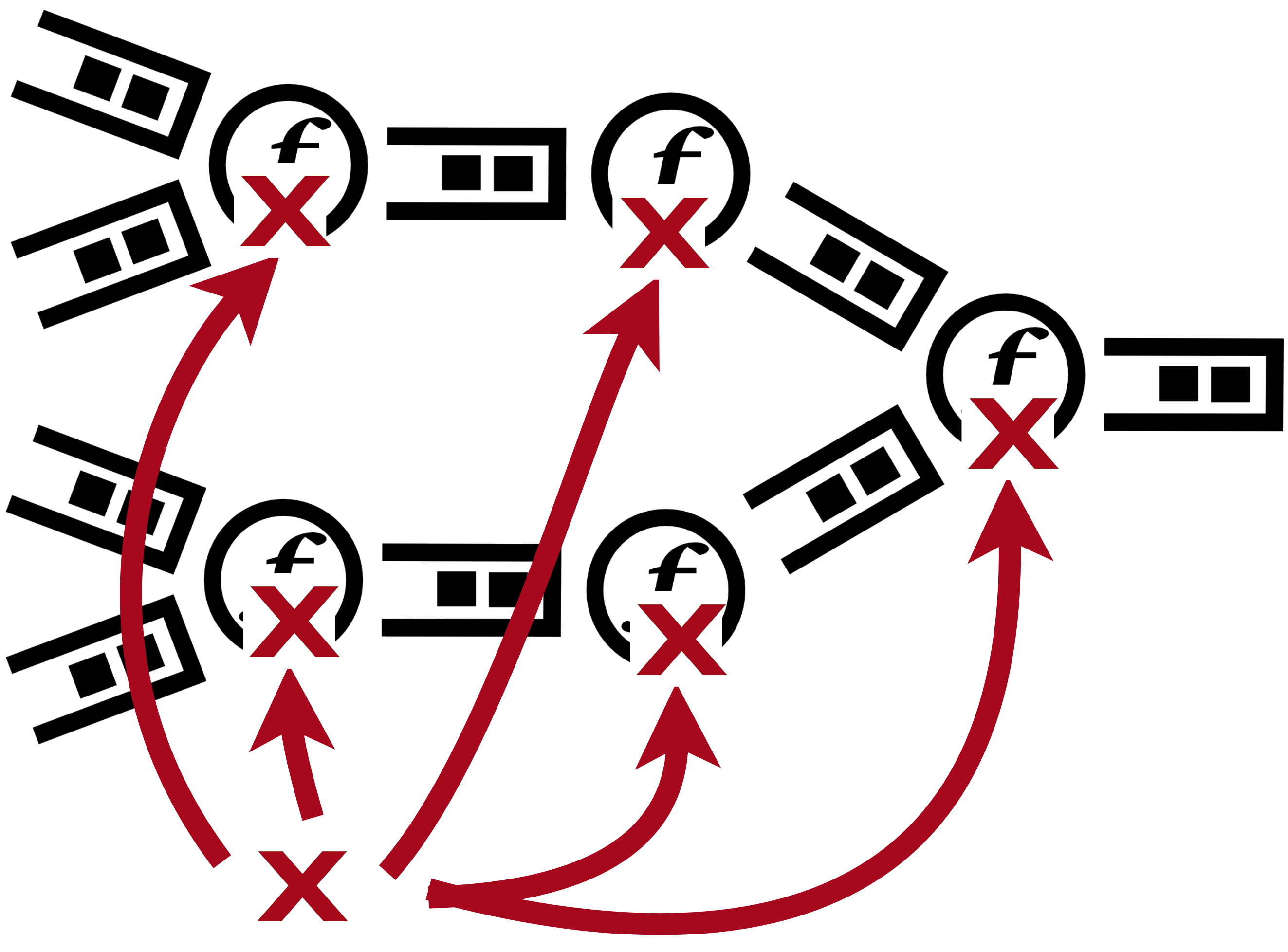
Exception Handling

If exceptions occur in a node,
the exception is provided as the result of the final output pipe.



Cancellations / Timeouts

If an execution needs to be cancelled then we kill all the nodes (and sources).



What is the API for building processing trees?

Object Construction

We could construct all the parts and wire them up.

```
(let [pipe1 (make-pipe)
      pipe2 (make-pipe)
      pipe3 (make-pipe)
      node1 (make-node pipe1 pipe2 ...)
      node2 (make-node pipe2 pipe3 ...))
```

Example: Word Count

Define stream operators
and express processing trees as s-expressions.

```
(preduce+ + 0 +
          (pmap+ (comp count #(split % #" "))
                 (source-data+ [["hello"
                                 "a simple test"]])))
```

Stream Expressions

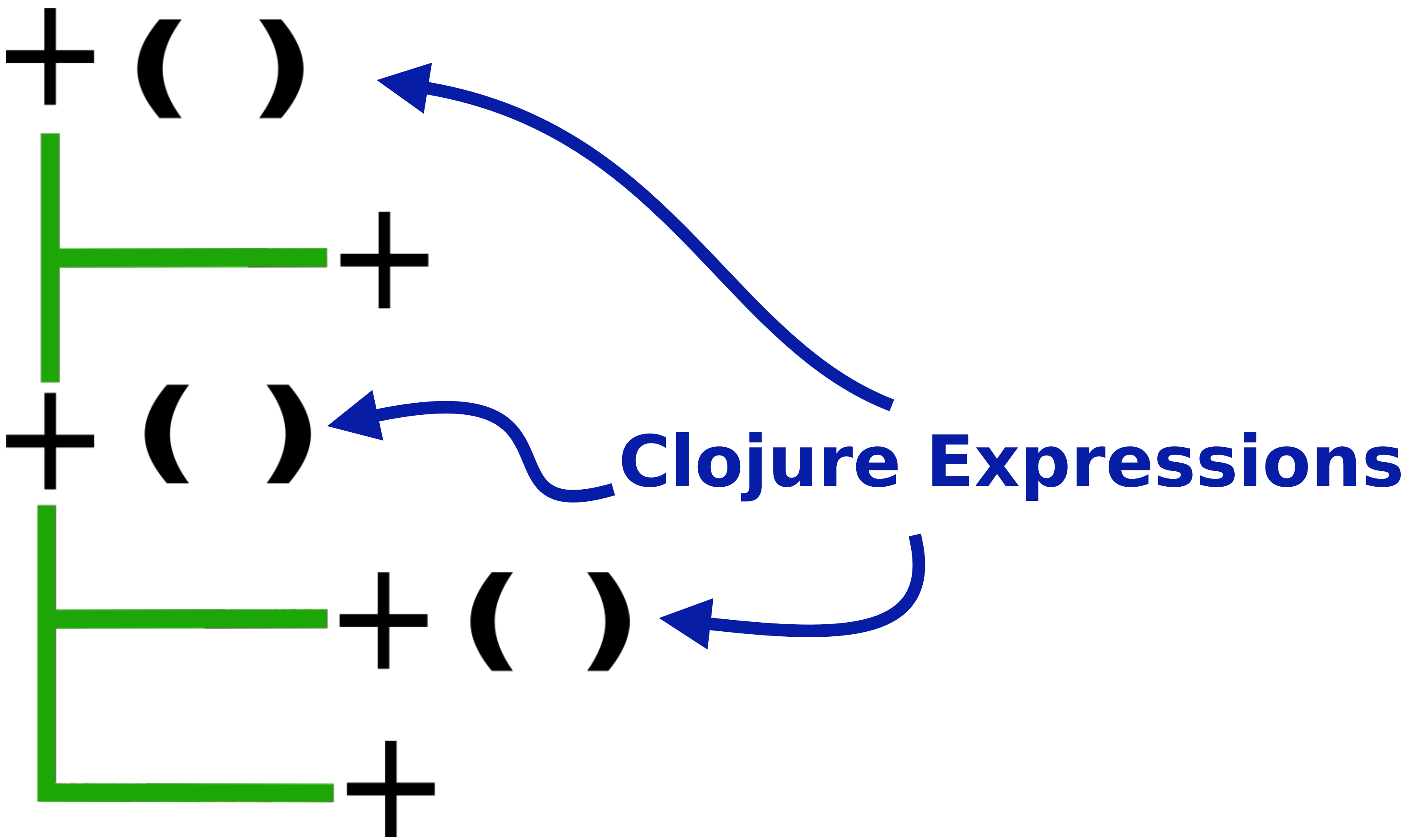
Stream expressions are built of a core tree of stream operators.
Stream expression have "holes" where Clojure expressions appear.

```
(preduce+ + 0 +
          (pmap+ (comp count #(split % #" "))
                 (source-data+ [["hello"
                                  "a simple test"]])))
```

```
(preduce+ _ _ _
          (pmap+ _
                 (source-data+ _))))
```

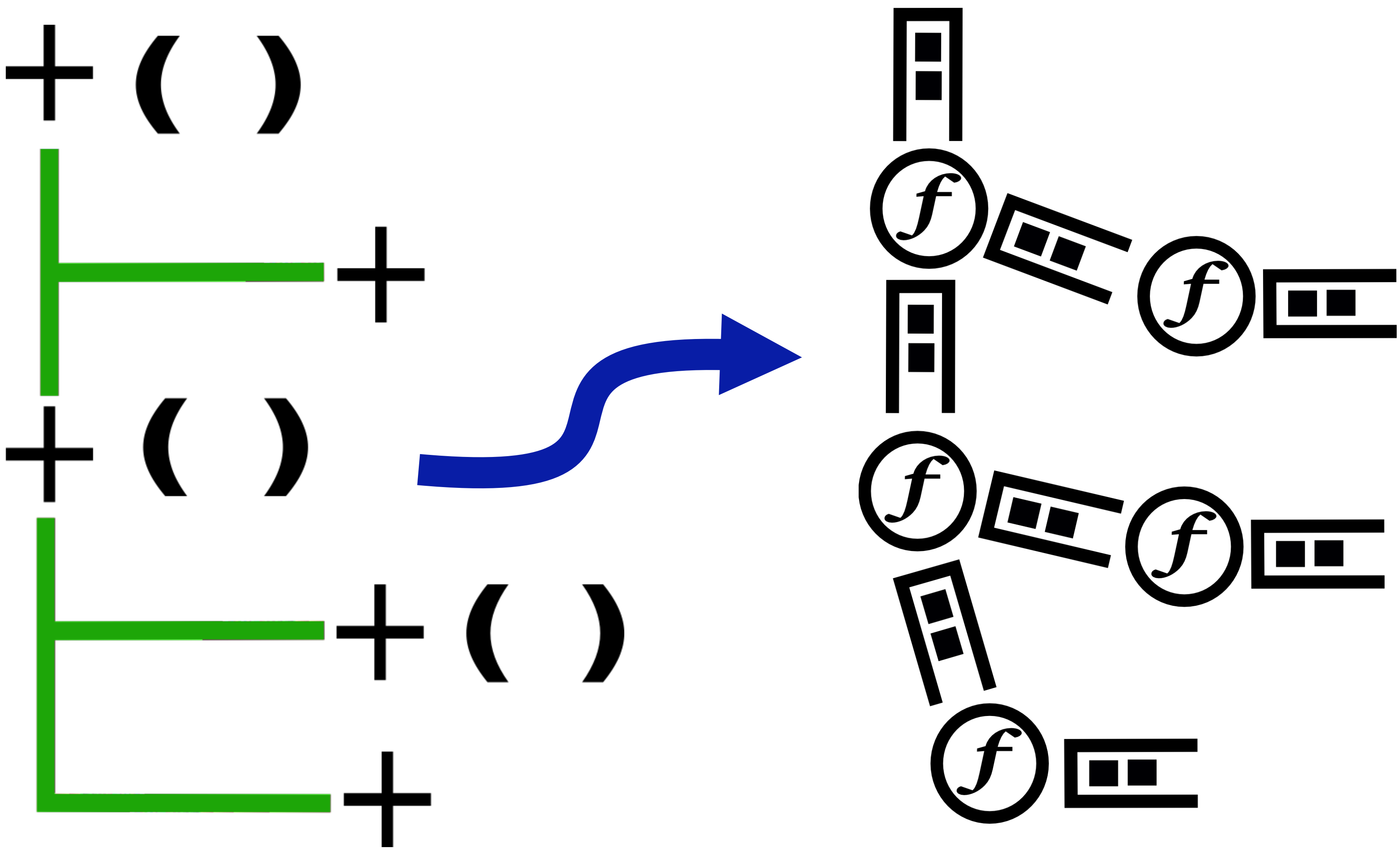
Stream Expressions

Clojure expressions appear, in well-defined places, as leaves in stream expression trees.



Compiling Stream Expressions

Stream expressions compiled into pipe/node trees.
Pipes are implicit in the structure of the stream expression.



Stream Operators

Core stream operators mirror Clojure sequence functions.

```
map+
mapcat+
filter+

pmap+
pmapcat+
pfilter+
preduce+
```

Stream Operators

Operators for taking parts of streams.

map+
mapcat+
filter+

pmap+
pmapcat+
filter+
preduce+

first+
take+
drop+
distinct+
mux+ *combines two streams into one*

Stream "let" Operators

"let" operators allow multiple incoming streams to be processed together

map+
mapcat+
filter+

pmap+
pmapcat+
pfilter+
preduce+

first+
take+
drop+
distinct+
mux+

let+
let-stream+

let+

The results of processing one stream are captured in the "word-count" variable and used while processing a second stream.

```
(exec-stream
  '(let+ [word-count (first+
                          (preduce+ + 0 +
                             (pmap+ (comp count #(split % #" "))
                                    (source-data+ ["hello"
                                                   "a simple test"]))))]
         (pfilter+ #(= (count %) word-count)
                   (source-data+ [[:a]
                                  [:a :b :c :d]
                                  [:x :y :z]
                                  [:p :q :r :s]]))))

=> ([:a :b :c :d] [:p :q :r :s])
```

let-stream+

let-stream+ assigns a name to an entire stream,
each use of the name gets a copy of the data stream.

```
(exec-stream '(let-stream+ [tuples (pmap+ #(assoc % :c 100)
                                          (source-data+ [{:a 1 :b 2}
                                                         {:a 10 :b 20}]))]
                           (mux+ (pmap+ :a tuples)
                                 (pmap+ #(* (:b %)
                                            (:c %)) tuples))))

=> (200 2000 1 10)
```

Stream "Chunk" Operators

Chunk operators allow operations on chunks rather than individual data items

```
map+
mapcat+
filter+

pmap+
pmapcat+
pfilter+
preduce+

first+
take+
drop+
distinct+
mux+
```

```
let+
let-stream+

pmap-chunk+
preduce-chunk+
number+
reorder+
rechunk+
```

Stream "Processing" Operators

The processing operators are automatically added to stream expressions by the stream expression compiler.

```
map+
mapcat+
filter+

pmap+
pmapcat+
pfilter+
preduce+

first+
take+
drop+
distinct+
mux+
```

```
let+
let-stream+

pmap-chunk+
preduce-chunk+
number+
reorder+
rechunk+

node+
no+
```

node+ operator

Stream expression compiler identifies node boundaries based on concurrency and forks in the data stream.

```
(map+ inc
      (preduce+ + 0 +
                (pmap+ (comp count #(split % #" "))
                (source-data+ ["hello"
                               "a simple test"]))))
```

```
(node+
  (map+
    (node+
      (preduce+ + 0 +
        (pmap+ (comp count
                (fn* [p1__41053#]
                     (split p1__41053# #" ")))
        (source-data+ ["hello"
                       "a simple test"])))))
```

no+ operator

Internally the stream expression compiler numbers nodes with "no+". This provides an "expression number" for debugging and for naming.

```
(map+ inc
      (preduce+ + 0 +
                (pmap+ (comp count #(split % #" "))
                (source-data+ ["hello"
                               "a simple test"]))))
```

```
(no+ 1
  (map+
    (no+ 2
      (preduce+ + 0 +
        (no+ 3 (pmap+ (comp count
                        (fn* [p1__41053#]
                            (split p1__41053# #" ")))
                 (no+ 4 (source-data+ ["hello"
                                       "a simple test"])))))
```

Stream Expressions: Macros

Clojure macros can produce stream expressions.

```
(defmacro word-counter [regex source]
  `(~'preduce+ + 0 +
              (~'pmap+ (comp count #(split % ~regex))
                       ~source)))
```

Stream Expressions: Macros

Stream expressions can include macro invocations.

```
(defmacro word-counter [regex source]
  `(~'preduce+ + 0 +
             (~'pmap+ (comp count #(split % ~regex))
                    ~source)))


(exec-stream
  '(word-counter #"_"
                (source-data+ ["hello"
                               "a_simple_test"])))
=> (4)
```

Stream Expressions: Macros

Macros are expanded before executing stream expressions.

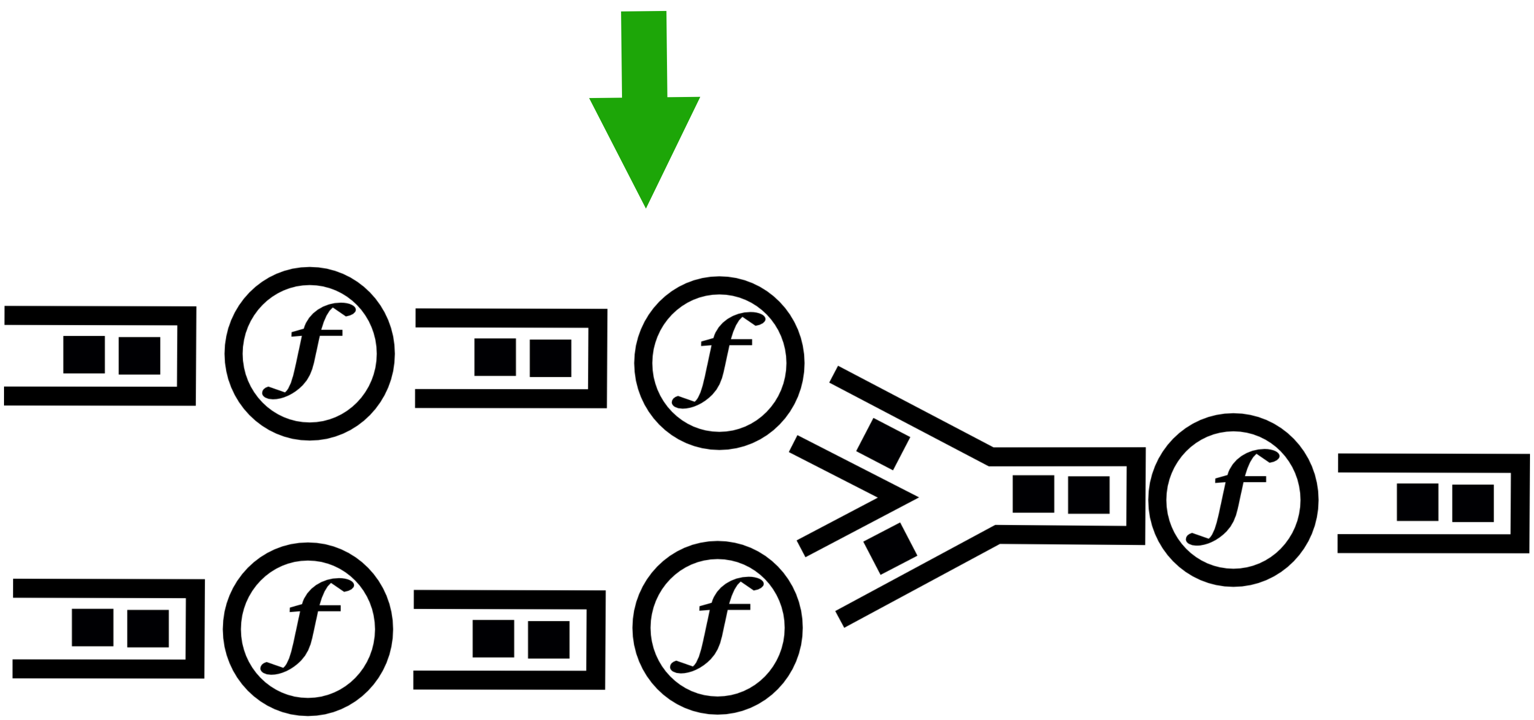
```
(clojure.pprint/pprint
  (macroexpand-all
    '(word-counter #"_"
                   (source-data+ ["hello"
                                  "a_simple_test"]))))

=>
(preduce+
 clojure.core/+
 0
 clojure.core/+
 (pmap+
  (clojure.core/comp
   clojure.core/count
   (fn*
    [p1__26302__26303__auto__]
    (split p1__26302__26303__auto__ #"_")))
  (source-tuples [["hello" "a_simple_test"]])))
```

Clojure FTW!

Low-level processing constructs accessed through a high-level DSL integrated tightly with Clojure.

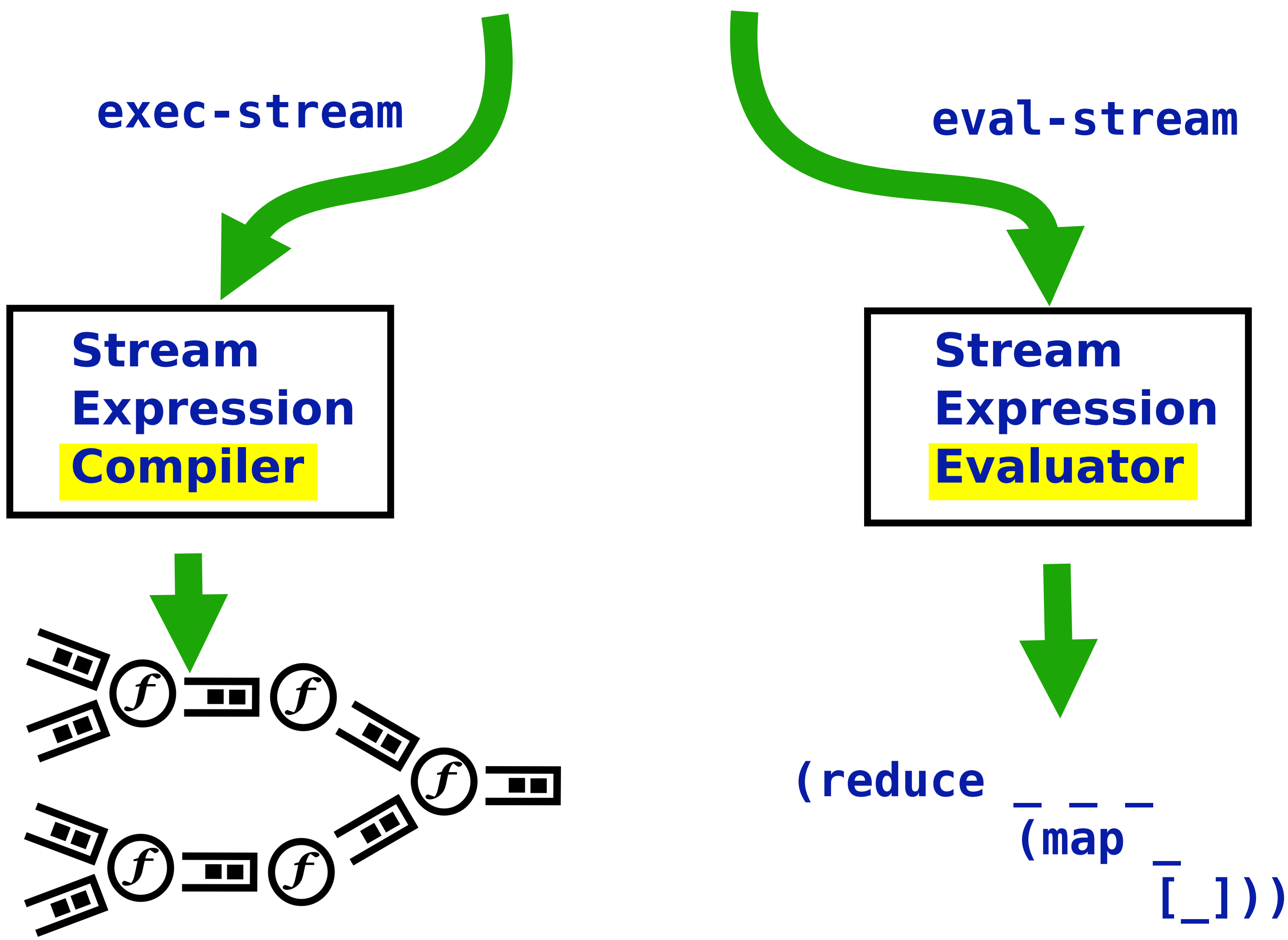
```
(preduce+ + 0 +
          (pmap+ (comp count #(split % #" "))
                 (source-data+ [["hello"
                                 "a simple test"]])))
```



Compiling vs Evaluating

Compiler produces a node/pipe tree.
Evaluator converts to a Clojure sequence equivalent.

```
(preduce+ _ _ _
   (pmap+ _
      (source-data+ _))))
```



Evaluating Stream Expressions

Alternatively stream expressions can be "evaluated" as simple Clojure sequence operations.

```
(exec-stream
  '(preduce+ + 0 +
             (pmap+ (comp count #(split % #" "))
                    (source-data+ ["hello"
                                   "a simple test"]))))

=> (4)
```

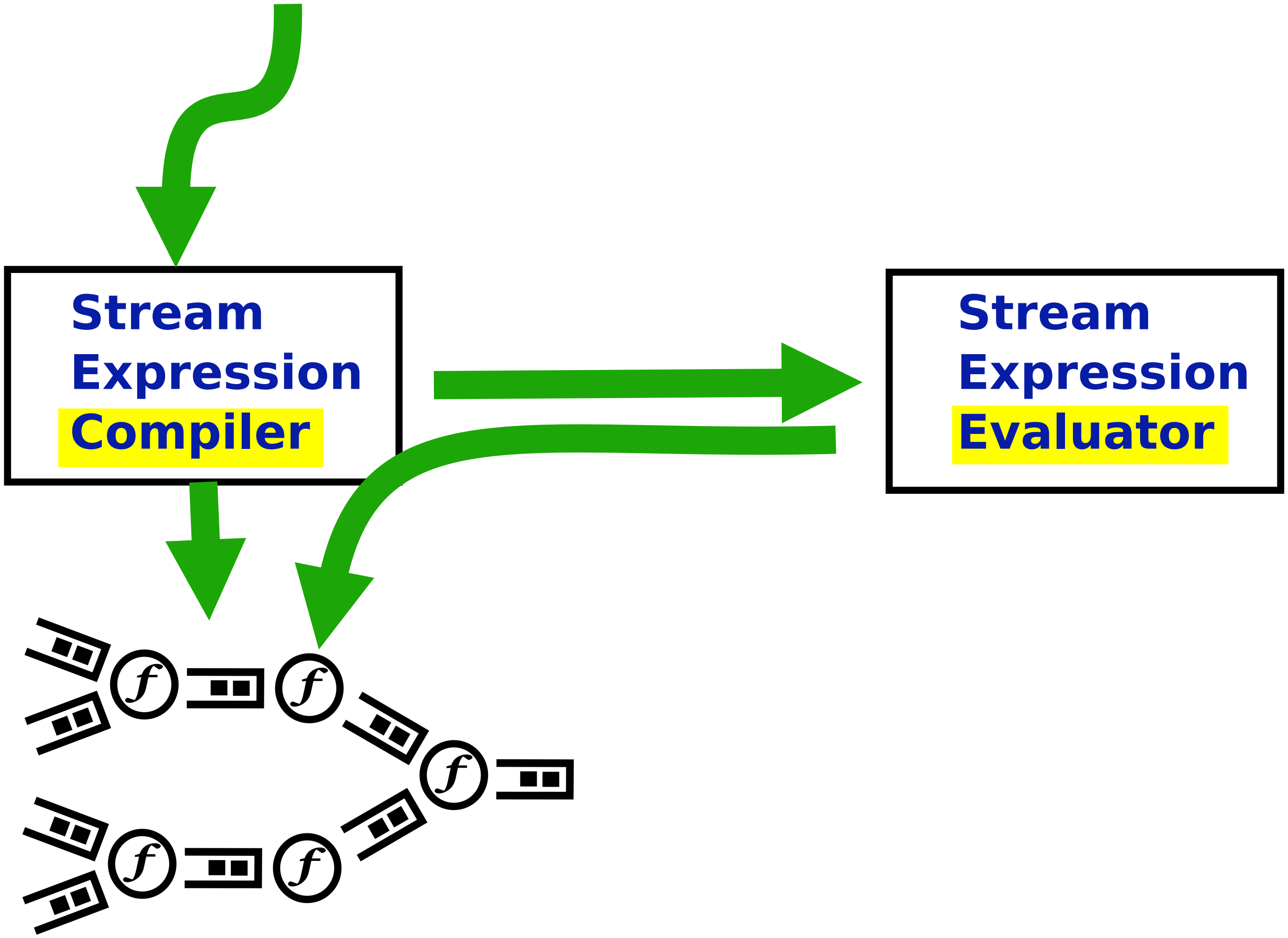
```
(eval-stream
  '(preduce+ + 0 +
             (pmap+ (comp count #(split % #" "))
                    (source-data+ ["hello"
                                   "a simple test"]))))

=> [4]
```

Compiler Uses Evaluator

Compiler uses the evaluator to produce the contents of each node.

```
(preduce+ ..._ _ _
   (pmap+ ..._
      (source-data+ ...))))
```

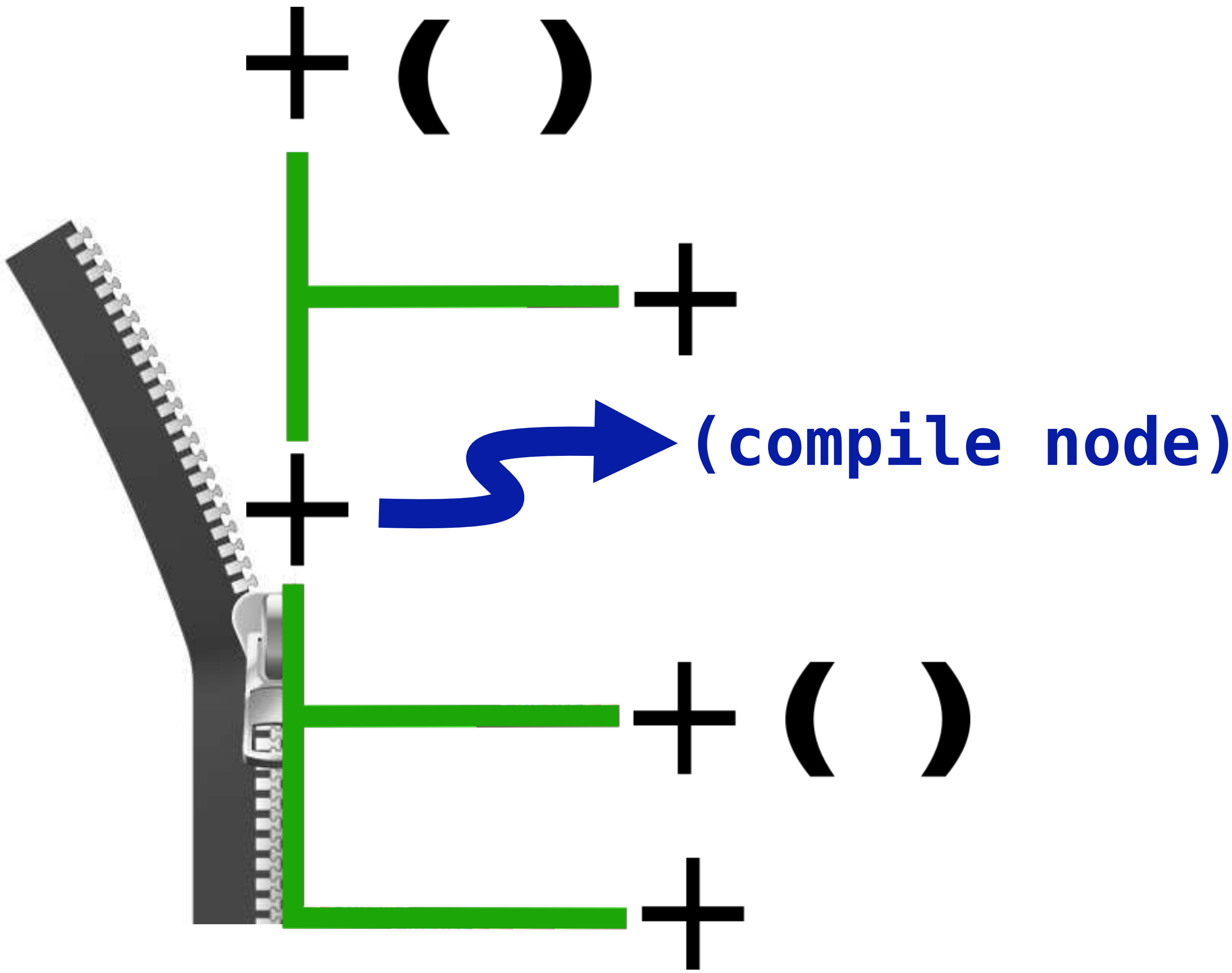


Inside the Stream Expression Compiler

Compiler is implemented with two key parts:

1) a zipper walks the expression tree

2) the "compile" multi-method is dispatched on each node



Inside the Stream Expression Compiler

Compiler is implemented with two key parts:
1) a zipper walks the expression tree
2) the "compile" multi-method is dispatched on each node

```
(defmulti compile first)

(defmethod compile 'pmap+ [[_ f stream]]
  ...)

(defmethod compile 'preduce+ [[_ reduce-f initial-value
                                  merge-f stream]]
  ...)

(defn compile-plan [plan-expr]
  ...
  (let [expr-zipper (zip/zipper op? streams new-streams plan-expr)]
       ;; iterate through the zipper applying the compile fn
       ))
```

Compiler Implemented via eval

We implemented an early version of the compiler as function & macros.
But, we found the zipper/multi-method based version simpler.
At the cost of losing the ability to have expressions
in the "structure" of the tree.

```
(defn pmap+ [f stream]
   ...)

(defn preduce+ [reduce-f initial-value merge-f stream]
   ...)
```

Our DSL Approach: Summary

s-expressions

- produced via Clojure mechanisms

- passed around as data

unqualified symbols for operators

tree of known operators

- avoid general code walking

- user macros expand to known operators

multiple operator implementations

- some invoked via Clojure "eval"

- some implemented via zippers and invoked as functions

expressions do not "eval" to their final value

```
(preduce+ + 0 +
                 (pmap+ (comp count #(split % #" "))
                             (source-data+ [["hello"
                                              "a simple test"]])))
```

How do we use stream processing in our apps?

Code Generation

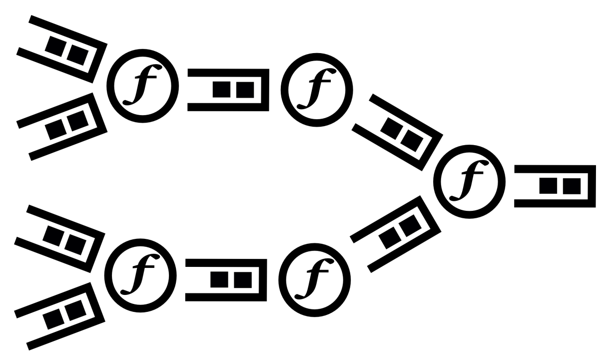
Alternate view:
our application generates a program to process each query.

Query Parse Tree

Query Plan (Clojure records)

```
(pmap+ ...
  (pfilter+ ...))
```

Query Plan "Program"



What next?

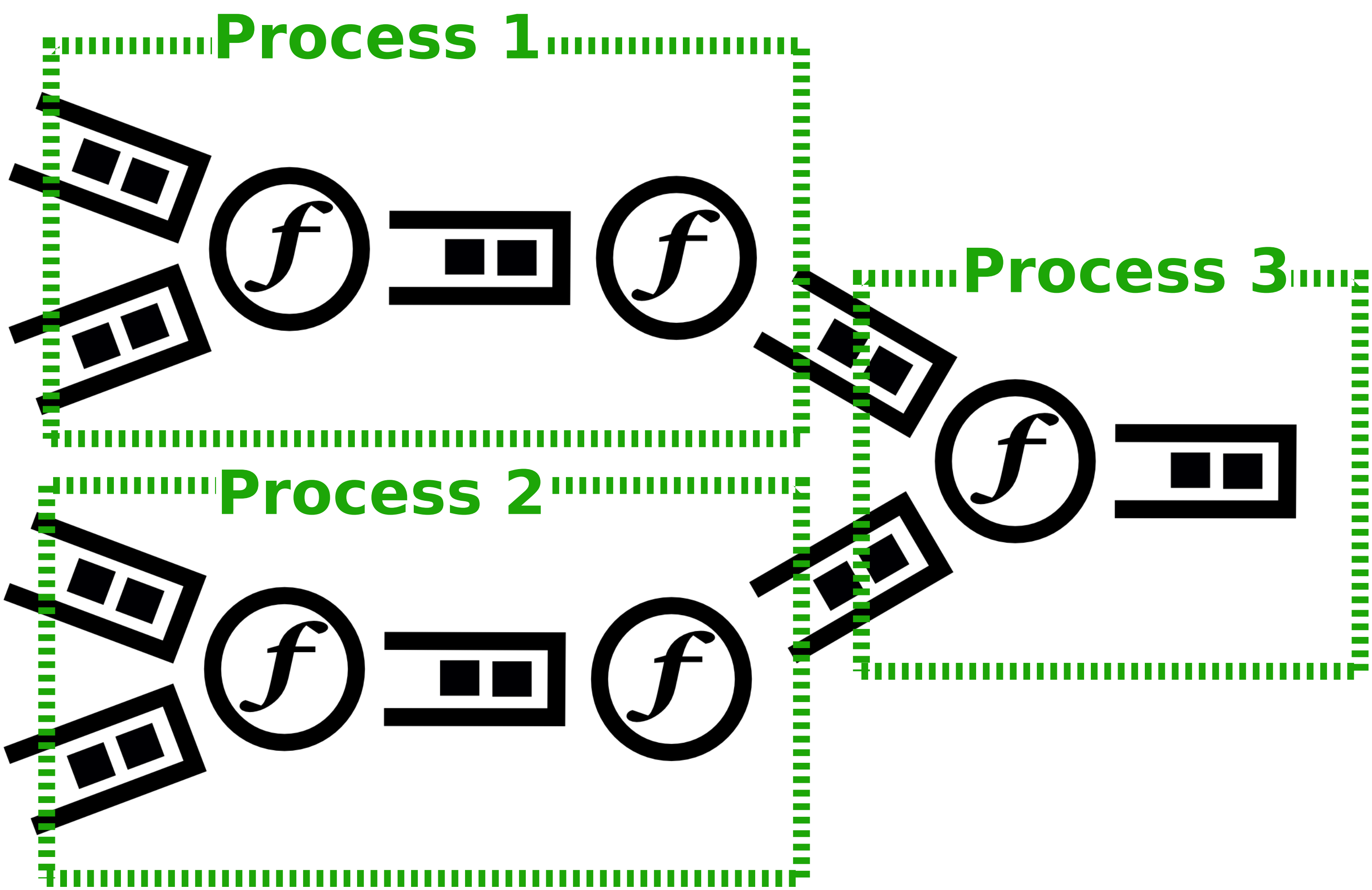
Tuple Processing Expressions

We are working on defining higher-level, tuple specific operators.

```
(select+ [x (+ y 20)]
         (join+ (= a/id b/id)
                [a (select+ ...)
                 b (select+ ...)]))
```

Distributed Stream Processing

Future possibility... break a processing tree into parts,
ship parts to separate processes and wire them together.



Related Clojure Links:

http://dev.clojure.org/display/design/Asynchronous+Events

https://github.com/nathanmarz/storm

https://github.com/ztellman/lamina

https://github.com/stuartsierra/cljque

https://github.com/jduey/conduit

https://github.com/hiredman/die-geister

SPARQL-to-SQL Mapping - "Spyder"

http://www.revelytix.com/content/spyder

SPARQL Federator - "Spinner"

http://www.revelytix.com/content/spinner

Check out the Revelytix site for more information on the products.