Orc Standard Library v0.9.8

Table of Contents

1. Reference

1.1. core.inc: Fundamental sites and operators.

Fundamental sites and operators.

These declarations include both prefix and infix sites (operators). For consistency, all declarations are written in prefix form, with the site name followed by the operands. When the site name is surrounded in parentheses, as in `(+)`, it denotes an infix operator.

For a more complete description of the built-in operators and their syntax, see the Operators section of the User Guide.

`let` — **site** `let() :: Top`

When applied to no arguments, return a signal.

`let` — **site** `let[A](A) :: A`

When applied to a single argument, return that argument (behaving as the identity function).

`let` — **site** `let[A, ...](A, ...) :: (A, ...)`

When applied to two or more arguments, return the arguments in a tuple.

`if` — **site if**`(Boolean) :: Top`

Fail silently if the argument is false. Otherwise return a signal.

Example:

```
-- Publishes: "Always publishes"
  if(false) >> "Never publishes"
| if(true) >> "Always publishes"
```

`error` — **site** `error(String) :: Bot`

Halt with the given error message.

Example, using `error` to implement assertions:

```
def assert(b) =
  if b then signal else error("assertion failed")

-- Fail with the error message: "assertion failed"
assert(false)
```

(+) **site** `(+)(Number, Number) :: Number`

`a+b` returns the sum of `a` and `b`.

(-) **site** `(-)(Number, Number) :: Number`

`a-b` returns the value of `a` minus the value of `b`.

(0-) **site** `(0-)(Number) :: Number`

Return the additive inverse of the argument. When this site appears as an operator, it is written in prefix form without the zero, i.e. `(-a)`

(*) **site** `(*)(Number, Number) :: Number`

`a*b` returns the product of `a` and `b`.

(**) **site** `(**)(Number, Number) :: Number`

`a ** b` returns a^b, i.e. `a` raised to the `b`th power.

(/) **site** `(/)(Number, Number) :: Number`

`a/b` returns `a` divided by `b`. If both arguments have integral types, `(/)` performs integral division, rounding towards zero. Otherwise, it performs floating-point division. If `b=0`, `a/b` halts with an error.

Example:

```
  7/3   -- publishes 2
| 7/3.0 -- publishes 2.333...
```

(%) **site** `(%)(Number, Number) :: Number`

`a%b` computes the remainder of `a/b`. If `a` and `b` have integral types, then the remainder is given by the expression `a - (a/b)*b`. For a full description, see the Java Language Specification, 3rd edition [http://java.sun.com/docs/books/jls/third_edition/html/expressions.html#15.17.3].

(<) **site** `(<)(Top, Top) :: Boolean`

`a < b` returns true if `a` is less than `b`, and false otherwise.

(<=) **site** `(<=)(Top, Top) :: Boolean`

`a <= b` returns true if `a` is less than or equal to `b`, and false otherwise.

(>) **site** (>)(Top, Top) :: Boolean

a > b returns true if a is greater than b, and false otherwise.

(>=) **site** (>=)(Top, Top) :: Boolean

a >= b returns true if a is greater than or equal to b, and false otherwise.

(=) **site** (=)(Top, Top) :: Boolean

a = b returns true if a is equal to b, and false otherwise. The precise definition of "equal" depends on the values being compared, but always obeys the rule that if two values are considered equal, then one may be substituted locally for the other without affecting the behavior of the program.

Two values with the same object identity are always considered equal. In addition, Cor constant values and data structures are considered equal if their contents are equal. Other types are free to implement their own equality relationship provided it conforms to the rules given here.

Note that although values of different types may be compared with =, the substitutability principle requires that such values are always considered inequal, i.e. the comparison will return false.

(/=) **site** (/=)(Top, Top) :: Boolean

a/=b returns false if a=b, and true otherwise.

(~) **site** (~)(Boolean) :: Boolean

Return the logical negation of the argument.

(&&) **site** (&&)(Boolean, Boolean) :: Boolean

Return the logical conjunction of the arguments. This is not a short-circuiting operator; both arguments must be evaluated and available before the result is computed.

(||) **site** (||)(Boolean, Boolean) :: Boolean

Return the logical disjunction of the arguments. This is not a short-circuiting operator; both arguments must be evaluated and available before the result is computed.

(:) **site** (:)[A](A, List[A]) :: List[A]

The list a:b is formed by prepending the element a to the list b.

Example:

```
-- Publishes: (3, [4, 5])
3:4:5:[] >x:xs> (x,xs)
```

In patterns, the (:) deconstructor can be applied to a variety of list-like values such as Arrays and Java Iterables, in which case it returns the first element of the list-like value, and a new list-like value (not necessarily of the same type as the original list-like value) representing the tail. Modifying the structure of the original value (e.g. adding an element to an Iterable) may render old "tail"s unusable, so you should refrain from modifying a value while you are deconstructing it. This feature is highly experimental and will probably change in future versions of the implementation.

`abs`

```
def abs(Number) :: Number
```

Return the absolute value of the argument.

Implementation.

```
def abs(Number) :: Number
def abs(x) = if x < 0 then -x else x
```

`signum`

```
def signum(Number) :: Number
```

`signum(a)` returns `-1` if `a<0`, `1` if `a>0`, and `0` if `a=0`.

Implementation.

```
def signum(Number) :: Number
def signum(x) =
  if x < 0 then -1
  else if x > 0 then 1
  else 0
```

`min`

```
def min[A](A,A) :: A
```

Return the lesser of the arguments. If the arguments are equal, return the first argument.

Implementation.

```
def min[A](A,A) :: A
def min(x,y) = if y < x then y else x
```

`max`

```
def max[A](A,A) :: A
```

Return the greater of the arguments. If the arguments are equal, return the second argument.

Implementation.

```
def max[A](A,A) :: A
def max(x,y) = if x > y then x else y
```

`floor`

```
site floor(Number) :: Integer
```

Return the greatest integer less than or equal to this number.

`ceil`

```
site ceil(Number) :: Integer
```

Return the least integer greater than or equal to this number.

1.2. state.inc: General-purpose supplemental data structures.

General-purpose supplemental data structures.

`Some` **site** `Some[A](A) :: Option[A]`

An optional value which is available. This site may also be used in a pattern.

Example:

```
-- Publishes: (3,4)
Some((3,4)) >s> (
    s >Some((x,y))> (x,y)
  | s >None()> signal
)
```

`None` **site** `None[A]() :: Option[A]`

An optional value which is not available. This site may also be used in a pattern.

`Semaphore` **site** `Semaphore(Integer) :: Semaphore`

Return a semaphore with the given value. The semaphore maintains the invariant that its value is always non-negative.

An example using a semaphore as a lock for a critical section:

```
-- Prints:
-- Entering critical section
-- Leaving critical section
val lock = Semaphore(1)
lock.acquire() >>
println("Entering critical section") >>
println("Leaving critical section") >>
lock.release()
```

`acquire` **site** `Semaphore.acquire() :: Top`

If the semaphore's value is greater than `0`, decrement the semaphore and return a signal. If the semaphore's value is `0`, block until it becomes greater than `0`.

`acquirenb` **site** `Semaphore.acquirenb() :: Top`

If the semaphore's value is greater than `0`, decrement the semaphore and return a signal. If the semaphore's value is `0`, halt.

`release` **site** `Semaphore.release() :: Top`

If any calls to `acquire` are blocked, allow the oldest such call to return. Otherwise, increment the value of the semaphore. This may increment the value beyond that with which the semaphore was constructed.

`snoop` **site** `Semaphore.snoop() :: Top`

If any calls to `acquire` are blocked, return a signal. Otherwise, block until some call to `acquire` blocks.

snoopnb `site Semaphore.snoopnb() :: Top`

If any calls to `acquire` are blocked, return a signal. Otherwise, halt.

Buffer `site Buffer[A]() :: Buffer[A]`

Create a new buffer (FIFO channel) of unlimited size. A buffer supports get, put and close operations.

A buffer may be either empty or non-empty, and either open or closed. When empty and open, calls to `get` block. When empty and closed, calls to `get` halt. When closed, calls to `put` halt. In all other cases, calls return normally.

Example:

```
-- Publishes: 10
val b = Buffer()
  Rtimer(1000) >> b.put(10) >> stop
| b.get()
```

get `site Buffer[A].get() :: A`

Get an item from the buffer. If the buffer is open and no items are available, block until one becomes available. If the buffer is closed [6] and no items are available, halt.

getnb `site Buffer[A].getnb() :: A`

Get an item from the buffer. If no items are available, halt.

put `site Buffer[A].put(A) :: Top`

Put an item in the buffer. If the buffer is closed [6], halt.

close `site Buffer[A].close() :: Top`

Close the buffer and block until it is empty. This has the effect of immediately causing any blocked calls to `get` to halt. In addition, any subsequent calls to `put` will halt, and once the buffer becomes empty, any subsequent calls to `get` will halt.

closenb `site Buffer[A].closenb() :: Top`

Close the buffer and return a signal immediately. This has the effect of immediately causing any blocked calls to `get` to halt. In addition, any subsequent calls to `put` will halt, and once the buffer becomes empty, any subsequent calls to `get` will halt.

isClosed `site Buffer[A].isClosed() :: Boolean`

If the buffer is currently closed, return true, otherwise return false.

getAll `site Buffer[A].getAll() :: List[A]`

Get all of the items currently in the buffer, emptying the buffer and returning a list of the items in the order they were added. If there are no items in the buffer, return an empty list.

`BoundedBuffer` **`site`** `BoundedBuffer[A](Integer) :: BoundedBuffer[A]`

Create a new buffer (FIFO channel) with the given number of slots. Putting an item into the buffer fills a slot, and getting an item opens a slot. A buffer with zero slots is equivalent to a synchronous channel [8].

A bounded buffer may be empty, partly filled, or full, and either open or closed. When empty and open, calls to `get` block. When empty and closed, calls to `get` halt. When full and open, calls to `put` block. When closed, calls to `put` halt. In all other cases, calls return normally.

Example:

```
-- Publishes: "Put 1" "Got 1" "Put 2" "Got 2"
val c = BoundedBuffer(1)
  c.put(1) >> "Put " + 1
| c.put(2) >> "Put " + 2
| Rtimer(1000) >> (
    c.get() >n> "Got " + n
  | c.get() >n> "Got " + n
  )
```

`get` **`site`** `BoundedBuffer[A].get() :: A`

Get an item from the buffer. If the buffer is open and no items are available, block until one becomes available. If the buffer is closed [7] and no items are available, halt.

`getnb` **`site`** `BoundedBuffer[A].getnb() :: A`

Get an item from the buffer. If no items are available, halt.

`put` **`site`** `BoundedBuffer[A].put(A) :: Top`

Put an item in the buffer. If no slots are open, block until one becomes open. If the buffer is closed [7], halt.

`putnb` **`site`** `BoundedBuffer[A].putnb(A) :: Top`

Put an item in the buffer. If no slots are open, halt. If the buffer is closed [7], halt.

`close` **`site`** `BoundedBuffer[A].close() :: Top`

Close the buffer and block until it is empty. This has the effect of immediately causing any blocked calls to `get` to halt. In addition, any subsequent calls to `put` will halt, and once the buffer becomes empty, any subsequent calls to `get` will halt. Note that any blocked calls to `put` initiated prior to closing the buffer may still be allowed to return as usual.

`closenb` **site** `BoundedBuffer[A].closenb() :: Top`

Close the buffer and return a signal immediately. This has the effect of immediately causing any blocked calls to `get` to halt. In addition, any subsequent calls to `put` will halt, and once the buffer becomes empty, any subsequent calls to `get` will halt. Note that any blocked calls to `put` initiated prior to closing the buffer may still be allowed to return as usual.

`isClosed` **site** `BoundedBuffer[A].isClosed() :: Boolean`

If the buffer is currently closed, return true, otherwise return false.

`getOpen` **site** `BoundedBuffer[A].getOpen() :: Integer`

Return the number of open slots in the buffer. Because of concurrency this value may become out-of-date so it should only be used for debugging or statistical measurements.

`getBound` **site** `BoundedBuffer[A].getBound() :: Integer`

Return the total number of slots (open or filled) in the buffer.

`getAll` **site** `BoundedBuffer[A].getAll() :: [A]`

Get all of the items currently in the buffer or waiting to be added, emptying the buffer and returning a list of the items in the order they were added. If there are no items in the buffer or waiting to be added, return an empty list.

`SyncChannel` **site** `SyncChannel[A]() :: SyncChannel[A]`

Create a synchronous channel, or rendezvous.

Example:

```
-- Publish: 10
val c = SyncChannel()
  c.put(10)
| Rtimer(1000) >> c.get()
```

`get` **site** `SyncChannel[A].get() :: A`

Receive an item over the channel. If no sender is available, block until one becomes available.

`put` **site** `SyncChannel[A].put(A) :: Top`

Send an item over the channel. If no receiver is available, block until one becomes available.

`Cell` **site** `Cell[A]() :: Cell[A]`

Create a write-once storage location.

Example:

```
-- Publishes: 5 5
val c = Cell()
  c.write(5) >> c.read()
| Rtimer(1) >> ( c.write(10) ; c.read() )
```

`read` **site** `Cell[A].read() :: A`

Read a value from the cell. If the cell does not yet have a value, block until it receives one.

`readnb` **site** `Cell[A].readnb() :: A`

Read a value from the cell. If the cell does not yet have a value, halt.

`write` **site** `Cell[A].write() :: Top`

Write a value to the cell. If the cell already has a value, halt.

`Ref` **site** `Ref[A]() :: Ref[A]`

Create a rewritable storage location without an initial value.

Example:

```
val r = Ref()
Rtimer(1000) >> r := 5 >> stop
| println(r?) >>
  r := 10 >>
  println(r?) >>
  stop
```

`Ref` **site** `Ref[A](A) :: Ref[A]`

Create a rewritable storage location initialized to the provided value.

`read` **site** `Ref[A].read() :: A`

Read the value of the ref. If the ref does not yet have a value, block until it receives one.

`readnb` **site** `Ref[A].readnb() :: A`

Read the value of the ref. If the ref does not yet have a value, halt.

`write` **site** `Ref[A].write(A) :: Top`

Write a value to the ref.

`(?)` **def** `(?)[A](Ref[A]) :: A`

Get the value held by a reference. `x?` is equivalent to `x.read()`.

Implementation.

```
def (?)[A](Ref[A]) :: A
def (?)(r) = r.read()
```

`(:=)` `def (:=)[A](Ref[A], A) :: Top`

Set the value held by a reference. `x := y` is equivalent to `x.write(y)`.

Implementation.

```
def (:=)[A](Ref[A], A) :: Top
def (:=)(r,v) = r.write(v)
```

`Array` `site Array[A](Integer) :: Array[A]`

Create a new native array of the given size. The array is initialized to contain `null`s.

The resulting array can be called directly with an index, as if its type were **`lambda`** `(Integer) :: Ref[A]` In this case, it returns a Ref [9] pointing to the element of the array specified by an index, counting from 0. Changes to the array are reflected immediately in the ref and visa versa.

Simple example:

```
-- Publishes: 3
val a = Array(1)
a(0) := 3 >>
a(0)?
```

More complex example:

```
-- Publishes: 0 1 2
val a = Array(3)
for(0, a.length()) >i>
a(i) := f(i) >>
stop
; a(0)? | a(1)? | a(2)?
```

`Array` `site Array[A](Integer, String) :: Array[A]`

Create a new primitive array of the given size with the given primitive type. The initial values in the array depend on the primitive type: for numeric types, it is `0`; for booleans, `false`; for chars, the character with codepoint `0`.

The element type of the array should be the appropriate wrapper type for the given primitive type, although a typechecker may not be able to verify this. This constructor is only necessary when interfacing with certain Java libraries; most programs will just use the `Array(Integer)` constructor.

`get` `site Array[A].get(Integer) :: A`

Get the element of the array given by the index, counting from 0. `a.get(i)` is equivalent to `a(i)?`.

`set` **site** `Array[A].set(Integer, A) :: Top`

Set the element of the array given by the index, counting from 0. `a.set(i,v)` is equivalent to `a(i) := v`.

`slice` **site** `Array[A].slice(Integer, Integer) :: Array[A]`

Return a copy of the portion of the array with indices covered by the given half-open range. The result array is still indexed counting from 0.

`length` **site** `Array[A].length() :: Integer`

Return the size of the array.

`fill` **site** `Array[A].fill(A) :: Top`

Set every element of the array to the given value. The given value is not copied, but is shared by every element of the array, so for example `a.fill(Semaphore(1))` would allow you to access the same semaphore from every element `a`.

This method is primarily useful to initialize or reset an array to a constant value, for example:

```
-- Publishes: 0 0 0
val a = Array(3)
a.fill(0) >> each(a)
```

`IArray` **def** `IArray[A](Integer,` **lambda** `(Integer) :: A)(Integer) :: A`

The call `IArray(n,f)`, where `n` is a natural number and `f` a total function over natural numbers, creates and returns a partial, pre-computed version of `f` restricted to the range (0, `n`-1). If `f` halts on any number in this range, the call to `IArray` will halt.

The user may also think of the call as returning an array whose `i`th element is `f(i)`.

This function provides a simple form of memoisation; we avoid recomputing the value of `f(i)` by storing the result in an array.

Example:

```
val a = IArray(5, fib)
-- Publishes the 4th number of the fibonnaci sequence: 5
a(3)
```

Implementation.

```
def IArray[A](Integer, lambda (Integer) :: A)(Integer) :: A
```

```
def IArray(n, f) =
  val a = Array[A](n)
  def fill(Integer, lambda (Integer) :: A) :: Top
  def fill(i, f) =
    if i < 0 then signal
    else (a.set(i, f(i)), fill(i-1, f)) >> signal
  fill(n-1, f) >> a.get
```

`Set` **site** `Set[A]() :: Set[A]`

Construct an empty mutable set. The set considers two values `a` and `b` to be the same if and only if `a=b`. This site conforms to the Java interface `java.util.Set`, except that it obeys Orc rules for equality of elements rather than Java rules.

`add` **site** `Set[A].add(A) :: Boolean`

Add a value to the set, returning true if the set did not already contain the value, and false otherwise.

`remove` **site** `Set[A].remove(Top) :: Boolean`

Remove a value from the set, returning true if the set contained the value, and false otherwise.

`contains` **site** `Set[A].contains(Top) :: Boolean`

Return true if the set contains the given value, and false otherwise.

`isEmpty` **site** `Set[A].isEmpty() :: Boolean`

Return true if the set contains no values.

`clear` **site** `Set[A].clear() :: Top`

Remove all values from the set.

`size` **site** `Set[A].size() :: Integer`

Return the number of unique values currently contained in the set.

`Map` **site** `Map[K,V]() :: Map[K,V]`

Construct an empty mutable map from keys to values. Each key contained in the map is associated with exactly one value. The mapping considers two keys `a` and `b` to be the same if and only if `a=b`. This site conforms to the Java interface `java.util.Map`, except that it obeys Orc rules for equality of keys rather than Java rules.

`put` **site** `Map[K,V].put(K, V) :: V`

`map.put(k,v)` associates the value `v` with the key `k` in `map`, such that `map.get(k)` returns `v`. Return the value previously associated with the key, if any, otherwise return `Null()`.

`get` **site** `Map[K,V].get(K) :: V`

Return the value currently associated with the given key, if any, otherwise return `Null()`.

`remove` **site** `Map[K,V].remove(Top) :: V`

Remove the given key from the map. Return the value previously associated with the key, if any, otherwise return `Null()`.

`containsKey` **site** `Map[K,V].containsKey(Top) :: Boolean`

Return true if the map contains the given key, and false otherwise.

`isEmpty` **site** `Map[K,V].isEmpty() :: Boolean`

Return true if the map contains no keys.

`clear` **site** `Map[K,V].clear() :: Top`

Remove all keys from the map.

`size` **site** `Map[K,V].size() :: Integer`

Return the number of unique keys currently contained in the map.

`Counter` **site** `Counter(Integer) :: Counter`

Create a new counter initialized to the given value.

`Counter` **site** `Counter() :: Counter`

Create a new counter initialized to zero.

`inc` **site** `Counter.inc() :: Top`

Increment the counter.

`dec` **site** `Counter.dec() :: Top`

If the counter is already at zero, halt. Otherwise, decrement the counter and return a signal.

`onZero` **site** `Counter.onZero() :: Top`

If the counter is at zero, return a signal. Otherwise block until the counter reaches zero.

`value` **site** `Counter.value() :: Integer`

Return the current value of the counter.

Example:

```
-- Publishes five signals
val c = Counter(5)
repeat(c.dec)
```

`Dictionary` **site** `Dictionary() :: Dictionary`

Create a new dictionary (a mutable map from field names to values), initially empty. The first time each field of the dictionary is accessed (using dot notation), the dictionary creates and returns a new empty Ref [9] which will also be returned on subsequent accesses of the same field. Dictionaries allow you to easily create object-like data structures.

Example:

```
-- Prints: 1 2
val d = Dictionary()
  println(d.one.read()) >>
  println(d.two.read()) >>
  stop
| d.one.write(1) >>
  d.two.write(2) >>
  stop
```

Here is the same example rewritten using Orc's reference syntax to improve clarity:

```
-- Prints: 1 2
val d = Dictionary()
  println(d.one?) >>
  println(d.two?) >>
  stop
| d.one := 1 >>
  d.two := 2 >>
  stop
```

To create a multi-level dictionary, you must explicitly create sub-dictionaries for each field. For example:

```
-- Prints: 2
val d = Dictionary()
d.one := Dictionary() >>
d.one?.two := 2 >>
println(d.one?.two?) >>
stop
```

Note that you cannot write `d.one.two`: because `d.one` is a reference to a dictionary, and not simply a dictionary, you must dereference before accessing its fields, as in `d.one? >x> x.two`. For readers familiar with the C language, this is the same reason you must write `s->field` instead of `s.field` when `s` is a pointer to a struct.

`Record` **site** `Record(String, A, String, B, ...) :: Record[A, B, ...]`

Create a new record (an immutable map from field names to values). Arguments are consumed in pairs; the first argument of each pair is the key, and the second is the value for that key.

To access the value in record `r` for key `"x"`, use the syntax `r.x`. For example:

```
-- Publishes: 1
val r = Record(
  "one", 1,
  "two", 2)
r.one
```

`fst`

```
def fst[A,B]((A,B)) :: A
```

Return the first element of a pair.

Implementation.

```
def fst[A,B]((A,B)) :: A
def fst((x,_)) = x
```

`snd`

```
def snd[A,B]((A,B)) :: B
```

Return the second element of a pair.

Implementation.

```
def snd[A,B]((A,B)) :: B
def snd((_,y)) = y
```

`Interval`

```
site Interval[A](A, A) :: Interval[A]
```

`Interval(a,b)` returns an object representing the half-open interval [a,b).

`isEmpty`

```
site Interval[A].isEmpty() :: Boolean
```

Return true if this interval is empty.

`spans`

```
site Interval[A].spans(A) :: Boolean
```

Return true if the interval spans the given point, false otherwise.

`intersects`

```
site Interval[A].intersects(Interval[A]) :: Boolean
```

Return true if the given interval has a non-empty intersection with this one, and false otherwise.

`intersect`

```
site Interval[A].intersect(Interval[A]) :: Interval[A]
```

Return the intersection of this interval with another. If the two intervals do not intersect, returns an empty interval.

`contiguous`

```
site Interval[A].contiguous(Interval[A]) :: Boolean
```

Return true if the given interval is contiguous with this one (overlaps or abuts), and false otherwise.

`union`

```
site Interval[A].union(Interval[A]) :: Interval[A]
```

Return the union of this interval with another. Halts with an error if the two intervals are not contiguous.

`Intervals` **site** `Intervals[A]() :: Intervals[A]`

Return an empty set of intervals. An Intervals object is iterable; iterating over the set returns disjoint intervals in increasing order.

`isEmpty` **site** `Intervals[A].isEmpty() :: Boolean`

Return true if this set of intervals is empty.

`spans` **site** `Intervals[A].spans(A) :: Boolean`

Return true if this set of intervals spans the given point, and false otherwise.

`intersect` **site** `Intervals[A].intersect(Intervals[A]) :: Intervals[A]`

Return the intersection of this set of intervals with another.

`union` **site** `Intervals[A].union(Interval[A]) :: Intervals[A]`

Return the union of this set of intervals with the given interval. This method is most efficient when the given interval is before most of the intervals in the set.

1.3. idioms.inc: Higher-order Orc programming idioms.

Higher-order Orc programming idioms. Many of these are standard functional-programming combinators borrowed from Haskell or Scheme.

`apply` **site** `apply[A, ..., B](`**lambda** `(A, ...) :: B, List[A]) :: B`

Apply a function to a list of arguments.

`curry` **def** `curry[A,B,C](`**lambda** `(A,B) :: C)(A)(B) :: C`

Curry a function of two arguments.

Implementation.

```
def curry[A,B,C](lambda (A,B) :: C)(A)(B) :: C
def curry(f)(x)(y) = f(x,y)
```

`curry3` **def** `curry3[A,B,C,D](`**lambda** `(A,B,C) :: D)(A)(B)(C) :: D`

Curry a function of three arguments.

Implementation.

```
def curry3[A,B,C,D](lambda (A,B,C) :: D)(A)(B)(C) :: D
def curry3(f)(x)(y)(z) = f(x,y,z)
```

`uncurry`

```
def uncurry[A,B,C](lambda (A)(B) :: C)(A, B) :: C
```

Uncurry a function of two arguments.

Implementation.

```
def uncurry[A,B,C](lambda (A)(B) :: C)(A, B) :: C
def uncurry(f)(x,y) = f(x)(y)
```

`uncurry3`

```
def uncurry3[A,B,C,D](lambda (A)(B)(C) :: D)(A,B,C) :: D
```

Uncurry a function of three arguments.

Implementation.

```
def uncurry3[A,B,C,D](lambda (A)(B)(C) :: D)(A,B,C) :: D
def uncurry3(f)(x,y,z) = f(x)(y)(z)
```

`flip`

```
def flip[A,B,C](lambda (A, B) :: C)(B, A) :: C
```

Flip the order of parameters of a two-argument function.

Implementation.

```
def flip[A,B,C](lambda (A, B) :: C)(B, A) :: C
def flip(f)(x,y) = f(y,x)
```

`constant`

```
def constant[A](A)() :: A
```

Create a function which returns a constant value.

Implementation.

```
def constant[A](A)() :: A
def constant(x)() = x
```

`defer`

```
def defer[A,B](lambda (A) :: B, A)() :: B
```

Given a function and its argument, return a thunk which applies the function.

Implementation.

```
def defer[A,B](lambda (A) :: B, A)() :: B
def defer(f, x)() = f(x)
```

`defer2`

```
def defer2[A,B,C](lambda (A,B) :: C, A, B)() :: C
```

Given a function and its arguments, return a thunk which applies the function.

Implementation.

```
def defer2[A,B,C](lambda (A,B) :: C, A, B)() :: C
def defer2(f, x, y)() = f(x, y)
```

`ignore`

```
def ignore[A,B](lambda () :: B)(A) :: B
```

From a function of no arguments, create a function of one argument, which is ignored.

Implementation.

```
def ignore[A,B](lambda () :: B)(A) :: B
def ignore(f)(_) = f()
```

ignore2 `def ignore2[A,B,C](lambda () :: C)(A, B) :: C`

From a function of no arguments, create a function of two arguments, which are ignored.

Implementation.

```
def ignore2[A,B,C](lambda () :: C)(A, B) :: C
def ignore2(f)(_, _) = f()
```

compose `def compose[A,B,C](lambda (B) :: C, lambda (A) :: B)(A) :: C`

Compose two single-argument functions.

Implementation.

```
def compose[A,B,C](lambda (B) :: C,
                   lambda (A) :: B)(A) :: C
def compose(f,g)(x) = f(g(x))
```

while `def while[A](lambda (A) :: Boolean, lambda (A) :: A)(A) :: A`

Iterate a function while a predicate is satisfied, publishing each value passed to the function. The exact behavior is specified by the following implementation:

```
def while(p,f) =
  def loop(x) = if(p(x)) >> ( x | loop(f(x)) )
  loop
```

Example:

```
-- Publishes: 0 1 2 3 4 5
while(
  lambda (n) = (n <= 5),
  lambda (n) = n+1
)(0)
```

Implementation.

```
def while[A](lambda (A) :: Boolean,
             lambda (A) :: A)(A)
  :: A
def while(p,f) =
  def loop(A) :: A
  def loop(x) = if(p(x)) >> ( x | loop(f(x)) )
  loop
```

repeat `def repeat[A](lambda () :: A) :: A`

Call a function sequentially, publishing each value returned by the function. The expression `repeat(f)` is equivalent to the infinite expression `f() >x> ( x | f() >x> ( x | f() >x> ... ) )`

Implementation.

```
def repeat[A](lambda () :: A) :: A
def repeat(f) = f() >x> (x | repeat(f))
```

`fork` `def fork[A](List[lambda () :: A]) :: A`

Call a list of functions in parallel, publishing all values published by the functions.

The expression `fork([f,g,h])` is equivalent to the expression `f() | g() | h()`

Implementation.

```
def fork[A](List[lambda () :: A]) :: A
def fork([]) = stop
def fork(p:ps) = p() | fork(ps)
```

`forkMap` `def forkMap[A,B](lambda (A) :: B, List[A]) :: B`

Apply a function to a list in parallel, publishing all values published by the applications.

The expression `forkMap(f, [a,b,c])` is equivalent to the expression `f(a) | f(b) | f(c)`

Implementation.

```
def forkMap[A,B](lambda (A) :: B, List[A]) :: B
def forkMap(f, []) = stop
def forkMap(f, x:xs) = f(x) | forkMap(f, xs)
```

`seq` `def seq[A](List[lambda () :: A]) :: Top`

Call a list of functions in sequence, publishing a signal whenever the last function publishes. The actual publications of the given functions are not published.

The expression `seq([f,g,h])` is equivalent to the expression `f() >> g() >> h() >> signal`

Implementation.

```
def seq[A](List[lambda () :: A]) :: Top
def seq([]) = signal
def seq(p:ps) = p() >> seq(ps)
```

`seqMap` `def seqMap[A,B](lambda (A) :: B, List[A]) :: Top`

Apply a function to a list in in sequence, publishing a signal whenever the last application publishes. The actual publications of the given functions are not published.

The expression `seqMap(f, [a,b,c])` is equivalent to the expression `f(a) >> f(b) >> f(c) >> signal`

Implementation.

```
def seqMap[A,B](lambda (A) :: B, List[A]) :: Top
def seqMap(f, []) = signal
def seqMap(f, x:xs) = f(x) >> seqMap(f, xs)
```

`join` `def join[A](List[lambda () :: A]) :: Top`

Call a list of functions in parallel and publish a signal once all functions have completed.

The expression `join([f,g,h])` is equivalent to the expression `(f(), g(), h()) >> signal`

Implementation.

```
def join[A](List[lambda () :: A]) :: Top
def join([]) = signal
def join(p:ps) = (p(), join(ps)) >> signal
```

`joinMap` `def joinMap[A,B](lambda (A) :: B, List[A]) :: Top`

Apply a function to a list in parallel and publish a signal once all applications have completed.

The expression `joinMap(f, [a,b,c])` is equivalent to the expression `(f(a), f(b), f(c)) >> signal`

Implementation.

```
def joinMap[A,B](lambda (A) :: B, List[A]) :: Top
def joinMap(f, []) = signal
def joinMap(f, x:xs) = (f(x), joinMap(f, xs)) >> signal
```

`alt` `def alt[A](List[lambda () :: A]) :: A`

Call each function in the list until one publishes.

The expression `alt([f,g,h])` is equivalent to the expression `f() ; g() ; h()`

Implementation.

```
def alt[A](List[lambda () :: A]) :: A
def alt([]) = stop
def alt(p:ps) = p() ; alt(ps)
```

`altMap` `def altMap[A,B](lambda (A) :: B, List[A]) :: B`

Apply the function to each element in the list until one publishes.

The expression `altMap(f, [a,b,c])` is equivalent to the expression `f(a) ; f(b) ; f(c)`

Implementation.

```
def altMap[A,B](lambda (A) :: B, List[A]) :: B
```

```
def altMap(f, []) = stop
def altMap(f, x:xs) = f(x) ; altMap(f, xs)
```

por

```
def por(List[lambda () :: Boolean]) :: Boolean
```

Parallel or. Evaluate a list of boolean functions in parallel, publishing a value as soon as possible, and terminating any unnecessary ongoing computation.

Implementation.

```
def por(List[lambda () :: Boolean]) :: Boolean
def por([]) = false
def por(p:ps) =
  let(
    val b1 = p()
    val b2 = por(ps)
    if(b1) >> true | if(b2) >> true | (b1 || b2)
  )
```

pand

```
def pand(List[lambda () :: Boolean]) :: Boolean
```

Parallel and. Evaluate a list of boolean functions in parallel, publishing a value as soon as possible, and terminating any unnecessary ongoing computation.

Implementation.

```
def pand(List[lambda () :: Boolean]) :: Boolean
def pand([]) = true
def pand(p:ps) =
  let(
    val b1 = p()
    val b2 = pand(ps)
    if(~b1) >> false | if(~b2) >> false | (b1 && b2)
  )
```

collect

```
def collect[A](lambda () :: A) :: List[A]
```

Run a function, collecting all publications in a list. Return the list when the function terminates.

Example:

```
-- Publishes: [signal, signal, signal, signal, signal]
collect(defer(signals, 5))
```

Implementation.

```
def collect[A](lambda () :: A) :: List[A]
def collect(p) =
  val b = Buffer[A]()
  p() >x> b.put(x) >> stop
  ; b.getAll()
```

1.4. list.inc: Operations on lists.

Operations on lists. Many of these functions are similar to those in the Haskell prelude, but operate on the elements of a list in parallel.

`each` **def** each[A](List[A]) :: A

Publish every value in a list, simultaneously.

Implementation.

```
def each[A](List[A]) :: A
def each([]) = stop
def each(h:t) = h | each(t)
```

`map` **def** map[A,B](**lambda** (A) :: B, List[A]) :: List[B]

Apply a function to every element of a list (in parallel), returning a list of the results.

Implementation.

```
def map[A,B](lambda (A) :: B, List[A]) :: List[B]
def map(f,[]) = []
def map(f,h:t) = f(h):map(f,t)
```

`reverse` **def** reverse[A](List[A]) :: List[A]

Return the reverse of the given list.

Implementation.

```
def reverse[A](List[A]) :: List[A]
def reverse(l) =
  def tailrev(List[A], List[A]) :: List[A]
  def tailrev([],x) = x
  def tailrev(h:t,x) = tailrev(t,h:x)
  tailrev(l,[])
```

`filter` **def** filter[A](**lambda** (A) :: Boolean, List[A]) :: List[A]

Return a list containing only those elements which satisfy the predicate. The filter is applied to all list elements in parallel.

Implementation.

```
def filter[A](lambda (A) :: Boolean, List[A]) :: List[A]
def filter(p,[]) = []
def filter(p,x:xs) =
  val fxs = filter(p, xs)
  if p(x) then x:fxs else fxs
```

`head` **def** head[A](List[A]) :: A

Return the first element of a list.

Implementation.

```
def head[A](List[A]) :: A
def head(x:xs) = x
```

tail `def tail[A](List[A]) :: List[A]`

Return all but the first element of a list.

Implementation.

```
def tail[A](List[A]) :: List[A]
def tail(x:xs) = xs
```

init `def init[A](List[A]) :: List[A]`

Return all but the last element of a list.

Implementation.

```
def init[A](List[A]) :: List[A]
def init([x]) = []
def init(x:xs) = x:init(xs)
```

last `def last[A](List[A]) :: A`

Return the last element of a list.

Implementation.

```
def last[A](List[A]) :: A
def last([x]) = x
def last(x:xs) = last(xs)
```

empty `def empty[A](List[A]) :: Boolean`

Is the list empty?

Implementation.

```
def empty[A](List[A]) :: Boolean
def empty([]) = true
def empty(_) = false
```

index `def index[A](List[A], Integer) :: A`

Return the nth element of a list, counting from 0.

Implementation.

```
def index[A](List[A], Integer) :: A
def index(h:t, 0) = h
def index(h:t, n) = index(t, n-1)
```

append `def append[A](List[A], List[A]) :: List[A]`

Return the first list concatenated with the second.

Implementation.

```
def append[A](List[A], List[A]) :: List[A]
def append([],l) = l
def append(h:t,l) = h:append(t,l)
```

`foldl`

```
def foldl[A,B](lambda (B, A) :: B, B, List[A]) :: B
```

Reduce a list using the given left-associative binary operation and initial value. Given the list `[x1, x2, x3, ...]` and initial value `x0`, returns `f(... f(f(f(x0, x1), x2), x3) ...)`

Example using `foldl` to reverse a list:

```
-- Publishes: [3, 2, 1]
foldl(flip((:)), [], [1,2,3])
```

Implementation.

```
def foldl[A,B](lambda (B, A) :: B, B, List[A]) :: B
def foldl(f,z,[]) = z
def foldl(f,z,x:xs) = foldl(f,f(z,x),xs)
```

`foldl1`

```
def foldl1[A](lambda (A, A) :: A, List[A]) :: A
```

A special case of `foldl` which uses the last element of the list as the initial value. It is an error to call this on an empty list.

Implementation.

```
def foldl1[A](lambda (A, A) :: A, List[A]) :: A
def foldl1(f,x:xs) = foldl(f,x,xs)
```

`foldr`

```
def foldr[A,B](lambda (A, B) :: B, B, List[A]) :: B
```

Reduce a list using the given right-associative binary operation and initial value. Given the list `[..., x3, x2, x1]` and initial value `x0`, returns `f(... f(x3, f(x2, f(x1, x0))) ...)`

Example summing the numbers in a list:

```
-- Publishes: 6
foldr((+), 0, [1,2,3])
```

Implementation.

```
def foldr[A,B](lambda (A, B) :: B, B, List[A]) :: B
def foldr(f,z,xs) = foldl(flip(f),z,reverse(xs))
```

`foldr1`

```
def foldr1[A](lambda (A, A) :: A, List[A]) :: A
```

A special case of `foldr` which uses the last element of the list as the initial value. It is an error to call this on an empty list.

Implementation.

```
def foldr1[A](lambda (A, A) :: A, List[A]) :: A
```

```
def foldr1(f,xs) = foldl1(flip(f),reverse(xs))
```

afold

```
def afold[A](lambda (A, A) :: A, List[A]) :: A
```

Reduce a non-empty list using the given associative binary operation. This function reduces independent subexpressions in parallel; the calls exhibit a balanced tree structure, so the number of sequential reductions performed is O(log n). For expensive reductions, this is much more efficient than `foldl` or `foldr`.

Implementation.

```
def afold[A](lambda (A, A) :: A, List[A]) :: A
def afold(f, [x]) = x
{- Here's the interesting part -}
def afold(f, xs) =
  def afold'(List[A]) :: List[A]
  def afold'([]) = []
  def afold'([x]) = [x]
  def afold'(x:y:xs) = f(x,y):afold'(xs)
  afold(f, afold'(xs))
```

cfold

```
def cfold[A](lambda (A, A) :: A, List[A]) :: A
```

Reduce a non-empty list using the given associative and commutative binary operation. This function opportunistically reduces independent subexpressions in parallel, so the number of sequential reductions performed is as small as possible. For expensive reductions, this is much more efficient than `foldl` or `foldr`. In cases where the reduction does not always take the same amount of time to complete, it is also more efficient than `afold`.

Implementation.

```
def cfold[A](lambda (A, A) :: A, List[A]) :: A
def cfold(f, []) = stop
def cfold(f, [x]) = x
def cfold(f, [x,y]) = f(x,y)
def cfold(f, L) =
  val c = Buffer[A]()
  def work(Number, List[A]) :: A
  def work(i, x:y:rest) =
    c.put(f(x,y)) >> stop | work(i+1, rest)
  def work(i, [x]) = c.put(x) >> stop | work(i+1, [])
  def work(i, []) =
    if (i < 2) then c.get()
    else c.get() >x> c.get() >y>
         ( c.put(f(x,y)) >> stop | work(i-1,[]) )
  work(0, L)
```

zip

```
def zip[A,B](List[A], List[B]) :: List[(A,B)]
```

Combine two lists into a list of pairs. The length of the shortest list determines the length of the result.

Implementation.

```
def zip[A,B](List[A], List[B]) :: List[(A,B)]
```

```
def zip([],_) = []
def zip(_,[]) = []
def zip(x:xs,y:ys) = (x,y):zip(xs,ys)
```

`unzip`

```
def unzip[A,B](List[(A,B)]) :: (List[A], List[B])
```

Split a list of pairs into a pair of lists.

Implementation.

```
def unzip[A,B](List[(A,B)]) :: (List[A], List[B])
def unzip([]) = ([],[])
def unzip((x,y):z) = (x:xs,y:ys) <(xs,ys)< unzip(z)
```

`concat`

```
def concat[A](List[List[A]]) :: List[A]
```

Concatenate a list of lists into a single list.

Implementation.

```
def concat[A](List[List[A]]) :: List[A]
def concat([]) = []
def concat(h:t) = append(h,concat(t))
```

`length`

```
def length[A](List[A]) :: Integer
```

Return the number of elements in a list.

Implementation.

```
def length[A](List[A]) :: Integer
def length([]) = 0
def length(h:t) = 1 + length(t)
```

`take`

```
def take[A](Integer, List[A]) :: List[A]
```

Given a number `n` and a list `l`, return the first `n` elements of `l`. If `n` exceeds the length of `l`, or `n < 0`, take halts with an error.

Implementation.

```
def take[A](Integer, List[A]) :: List[A]
def take(0, _) = []
def take(n, x:xs) =
  if n > 0 then x:take(n-1, xs)
  else error("Cannot take(" + n + ", _)")
```

`drop`

```
def drop[A](Integer, List[A]) :: List[A]
```

Given a number `n` and a list `l`, return the elements of `l` after the first `n`. If `n` exceeds the length of `l`, or `n < 0`, drop halts with an error.

Implementation.

```
def drop[A](Integer, List[A]) :: List[A]
def drop(0, xs) = xs
def drop(n, x:xs) =
  if n > 0 then drop(n-1, xs)
```

```
  else error("Cannot drop(" + n + ", _)")
```

member

```
def member[A](A, List[A]) :: Boolean
```

Return true if the given item is a member of the given list, and false otherwise.

Implementation.

```
def member[A](A, List[A]) :: Boolean
def member(item, []) = false
def member(item, h:t) =
  if item = h then true
  else member(item, t)
```

merge

```
def merge[A](List[A], List[A]) :: List[A]
```

Merge two sorted lists.

Example:

```
-- Publishes: [1, 2, 2, 3, 4, 5]
merge([1,2,3], [2,4,5])
```

Implementation.

```
def merge[A](List[A], List[A]) :: List[A]
def merge(xs,ys) = mergeBy((<), xs, ys)
```

mergeBy

```
def mergeBy[A](lambda (A,A) :: Boolean, List[A], List[A]) :: List[A]
```

Merge two lists using the given less-than relation.

Implementation.

```
def mergeBy[A](lambda (A,A) :: Boolean,
               List[A], List[A]) :: List[A]
def mergeBy(lt, xs, []) = xs
def mergeBy(lt, [], ys) = ys
def mergeBy(lt, x:xs, y:ys) =
  if lt(y,x) then y:mergeBy(lt,x:xs,ys)
  else x:mergeBy(lt,xs,y:ys)
```

sort

```
def sort[A](List[A]) :: List[A]
```

Sort a list.

Example:

```
-- Publishes: [1, 2, 3]
sort([1,3,2])
```

Implementation.

```
def sort[A](List[A]) :: List[A]
def sort(xs) = sortBy((<), xs)
```

`sortBy`

```
def sortBy[A](lambda (A,A) :: Boolean, List[A]) :: List[A]
```

Sort a list using the given less-than relation.

Implementation.

```
def sortBy[A](lambda (A,A) :: Boolean, List[A]) :: List[A]
def sortBy(lt, []) = []
def sortBy(lt, [x]) = [x]
def sortBy(lt, xs) =
  val half = length(xs)/2
  val front = take(half, xs)
  val back = drop(half, xs)
  mergeBy(lt, sortBy(lt, front), sortBy(lt, back))
```

`mergeUnique`

```
def mergeUnique[A](List[A], List[A]) :: List[A]
```

Merge two sorted lists, discarding duplicates.

Example:

```
-- Publishes: [1, 2, 3, 4, 5]
mergeUnique([1,2,3], [2,4,5])
```

Implementation.

```
def mergeUnique[A](List[A], List[A]) :: List[A]
def mergeUnique(xs,ys) = mergeUniqueBy((=), (<), xs, ys)
```

`mergeUniqueBy`

```
def mergeUniqueBy[A](lambda (A,A) :: Boolean, lambda (A,A) :: Boolean, List[A], List[A]) :: List[A]
```

Merge two lists, discarding duplicates, using the given equality and less-than relations.

Implementation.

```
def mergeUniqueBy[A](lambda (A,A) :: Boolean,
                     lambda (A,A) :: Boolean,
                      List[A], List[A])
  :: List[A]
def mergeUniqueBy(eq, lt, xs, []) = xs
def mergeUniqueBy(eq, lt, [], ys) = ys
def mergeUniqueBy(eq, lt, x:xs, y:ys) =
  if eq(y,x) then mergeUniqueBy(eq, lt, xs, y:ys)
  else if lt(y,x) then y:mergeUniqueBy(eq,lt,x:xs,ys)
  else x:mergeUniqueBy(eq,lt,xs,y:ys)
```

`sortUnique`

```
def sortUnique[A](List[A]) :: List[A]
```

Sort a list, discarding duplicates.

Example:

```
-- Publishes: [1, 2, 3]
```

```
sortUnique([1,3,2,3])
```

Implementation.

```
def sortUnique[A](List[A]) :: List[A]
def sortUnique(xs) = sortUniqueBy((=), (<), xs)
```

sortUniqueBy `def sortUniqueBy[A](lambda (A,A) :: Boolean, lambda (A,A) :: Boolean, List[A]) :: List[A]`

Sort a list, discarding duplicates, using the given equality and less-than relations.

Implementation.

```
def sortUniqueBy[A](lambda (A,A) :: Boolean,
                    lambda (A,A) :: Boolean,
                    List[A])
  :: List[A]
def sortUniqueBy(eq, lt, []) = []
def sortUniqueBy(eq, lt, [x]) = [x]
def sortUniqueBy(eq, lt, xs) =
  val half = length(xs)/2
  val front = take(half, xs)
  val back = drop(half, xs)
  mergeUniqueBy(eq, lt,
    sortUniqueBy(eq, lt, front),
    sortUniqueBy(eq, lt, back))
```

group `def group[A,B](List[(A,B)]) :: List[(A,List[B])]`

Given a list of pairs, group together the second elements of consecutive pairs with equal first elements.

Example:

```
-- Publishes: [(1, [1, 2]), (2, [3]), (3, [4]), (1, [3])]
group([(1,1), (1,2), (2,3), (3,4), (1,3)])
```

Implementation.

```
def group[A,B](List[(A,B)]) :: List[(A,List[B])]
def group(xs) = groupBy((=), xs)
```

groupBy `def groupBy[A,B](lambda (A,A) :: Boolean, List[(A,B)]) :: List[(A,List[B])]`

Given a list of pairs, group together the second elements of consecutive pairs with equal first elements, using the given equality relation.

Implementation.

```
def groupBy[A,B](lambda (A,A) :: Boolean,
                 List[(A,B)])
  :: List[(A,List[B])]
def groupBy(eq, []) = []
def groupBy(eq, (k,v):kvs) =
```

```
  def helper(A, List[B], List[(A,B)]) :: List[(A,List[B])]
  def helper(k,vs, []) = [(k,vs)]
  def helper(k,vs, (k2,v):kvs) =
    if eq(k2,k) then helper(k, v:vs, kvs)
    else (k,vs):helper(k2, [v], kvs)
  helper(k,[v], kvs)
```

`rangeBy`

```
def rangeBy(Number, Number, Number) :: List[Number]
```

`rangeBy(low, high, skip)` returns a sorted list of numbers `n` which satisfy `n = low + skip*i` (for some integer `i`), `n >= low`, and `n < high`.

Implementation.

```
def rangeBy(Number, Number, Number) :: List[Number]
def rangeBy(low, high, skip) =
  if low < high
  then low:rangeBy(low+skip, high, skip)
  else []
```

`range`

```
def range(Number, Number) :: List[Number]
```

Generate a list of numbers in the given half-open range.

Implementation.

```
def range(Number, Number) :: List[Number]
def range(low, high) = rangeBy(low, high, 1)
```

`any`

```
def any[A](lambda (A) :: Boolean, List[A]) :: Boolean
```

Return true if any of the elements of the list match the predicate, and false otherwise. The predicate is applied to all elements of the list in parellel; the result is returned as soon as it is known and any unnecessary evaluation of the predicate terminated.

Implementation.

```
def any[A](lambda (A) :: Boolean, List[A]) :: Boolean
def any(p, []) = false
def any(p, x:xs) =
  let(
    val b1 = p(x)
    val b2 = any(p, xs)
    if(b1) >> true | if(b2) >> true | (b1 || b2)
  )
```

`all`

```
def all[A](lambda (A) :: Boolean, List[A]) :: Boolean
```

Return true if all of the elements of the list match the predicate, and false otherwise. The predicate is applied to all elements of the list in parellel; the result is returned as soon as it is known and any unnecessary evaluation of the predicate terminated.

Implementation.

```
def all[A](lambda (A) :: Boolean, List[A]) :: Boolean
def all(p, []) = true
def all(p, x:xs) =
```

```
  let(
    val b1 = p(x)
    val b2 = all(p, xs)
    if(~b1) >> false | if(~b2) >> false | (b1 && b2)
  )
```

`sum` `def sum(List[Number]) :: Number`

Return the sum of all numbers in a list. The sum of an empty list is 0.

Implementation.

```
def sum(List[Number]) :: Number
def sum(xs) = foldl(
  (+) :: lambda (Number, Number) :: Number,
  0, xs)
```

`product` `def product(List[Number]) :: Number`

Return the product of all numbers in a list. The product of an empty list is 1.

Implementation.

```
def product(List[Number]) :: Number
def product(xs) = foldl(
  (*) :: lambda (Number, Number) :: Number,
  1, xs)
```

`and` `def and(List[Boolean]) :: Boolean`

Return the boolean conjunction of all boolean values in the list. The conjunction of an empty list is `true`.

Implementation.

```
def and(List[Boolean]) :: Boolean
def and([]) = true
def and(false:xs) = false
def and(true:xs) = and(xs)
```

`or` `def or(List[Boolean]) :: Boolean`

Return the boolean disjunction of all boolean values in the list. The disjunction of an empty list is `false`.

Implementation.

```
def or(List[Boolean]) :: Boolean
def or([]) = false
def or(true:xs) = true
def or(false:xs) = or(xs)
```

`minimum` `def minimum[A](List[A]) :: A`

Return the minimum element of a non-empty list.

Implementation.

```
def minimum[A](List[A]) :: A
def minimum(xs) =
  -- this def appeases the typechecker
  def minA(x::A,y::A) = min(x,y)
  foldl1(minA, xs)
```

`maximum`

```
def maximum[A](List[A]) :: A
```

Return the maximum element of a non-empty list.

Implementation.

```
def maximum[A](List[A]) :: A
def maximum(xs) =
  -- this def appeases the typechecker
  def maxA(x::A,y::A) = max(x,y)
  foldl1(maxA, xs)
```

1.5. text.inc: Operations on strings.

Operations on strings.

`cat`

```
site cat(Top, ...) :: String
```

Return the string representation of one or more values, concatenated. For Java objects, this will call `toString()` to convert the object to a String.

`print`

```
site print(Top, ...) :: Top
```

Print one or more values as strings, concatenated, to standard output. For Java objects, this will call `toString()` to convert the object to a String.

`println`

```
site println(Top, ...) :: Top
```

Print one or more values as strings, concatenated, to standard output, with each value followed by a newline. For Java objects, this will call `toString()` to convert the object to a String.

`read`

```
site read[A](String) :: A
```

Given a string representing an Orc value (using standard Orc literal syntax), return the corresponding value. If the argument does not conform to Orc literal syntax, halt with an error.

Example:

```
  read("true") -- publishes the boolean true
| read("1") -- publishes the integer 1
| read("(3.0, [])") -- publishes the tuple (3.0, [])
| read("\"hi\"") -- publishes the string "hi"
```

`write`

```
site write(Top) :: String
```

Given an Orc value, return its string representation using standard Orc literal syntax. If the value is of a type with no literal syntax, (for example, it is a site), return an arbitrary string representation which is intended to be human-readable.

Example:

```
  write(true) -- publishes "true"
| write(1) -- publishes "1"
| write((3.0, [])) -- publishes "(3.0, [])"
| write("hi") -- publishes "\"hi\""
```

`lines`

```
def lines(String) :: List[String]
```

Split a string into lines, which are substrings terminated by an endline or the end of the string. DOS, Mac, and Unix endline conventions are all accepted. Endline characters are not included in the result.

Implementation.

```
def lines(String) :: List[String]
def lines(text) =
  val out = text.split("\n|\r\n|\r")
  if out.get(out.length()-1) = "" then
    out.split(0, out.length()-1)
  else out
```

`unlines`

```
def unlines(List[String]) :: String
```

Append a linefeed, "\n", to each string in the sequence and concatenate the results.

Implementation.

```
def unlines(List[String]) :: String
def unlines(line:lines) = cat(line, "\n", unlines(lines))
def unlines([]) = ""
```

`words`

```
def words(String) :: List[String]
```

Split a string into words, which are sequences of non-whitespace characters separated by whitespace.

Implementation.

```
def words(String) :: List[String]
def words(text) = text.trim().split("\\s+")
```

`unwords`

```
def unwords(List[String]) :: String
```

Concatenate a sequence of strings with a single space between each string.

Implementation.

```
def unwords(List[String]) :: String
def unwords([]) = ""
def unwords([word]) = word
```

```
def unwords(word:words) = cat(word, " ", unwords(words))
```

1.6. time.inc: Real and logical time.

Real and logical time.

`Rtimer` **site** `Rtimer(Integer) :: Top`

Publish a signal after the given number of milliseconds.

`time` **site** `Rtimer.time() :: Integer`

Return the current real time in milliseconds, as measured from midnight January 1, 1970 UTC. Ranges from 0 to `Long.MAX_VALUE`.

`Clock` **def** `Clock()() :: Integer`

A call to `Clock` creates a new relative real-time clock. Calling a relative clock returns the number of milliseconds which have elapsed since the clock was created.

Example:

```
-- Publishes a value near 1000
val c = Clock()
Rtimer(1000) >> c()
```

`Ltimer` **site** `Ltimer(Integer) :: Top`

Publish a signal after the given number of logical timesteps, as measured by the current logical clock. The logical time advances whenever the computation controlled by the logical clock is quiescent (i.e. cannot advance on its own).

`time` **site** `Ltimer.time() :: Integer`

Return the current logical time, as measured by logical clock which was current when `Ltimer.time` was evaluated. Ranges from 0 to `Integer.MAX_VALUE`.

`withLtimer` **def** `withLtimer[A](`**lambda**` () :: A) :: A`

Run the given thunk in the context of a new inner logical clock. Within the computation represented by the thunk, calls to `Ltimer` refer to the new clock. The outer clock can only advance when the inner clock becomes quiescent.

`metronome` **def** `metronome(Integer) :: Top`

Publish a signal at regular intervals, indefinitely. The period is given by the argument, in milliseconds.

1.7. util.inc: Miscellaneous utility functions.

Miscellaneous utility functions.

`random` **site** `random() :: Integer`

Return a random Integer value chosen from the range of all possible 32-bit Integer values.

`random` **site** `random(Integer) :: Integer`

Return a pseudorandom, uniformly distributed Integer value between 0 (inclusive) and the specified value (exclusive). If the argument is 0, halt.

`urandom` **site** `urandom() :: Number`

Returns a pseudorandom, uniformly distributed Double value between 0 and 1, inclusive.

`UUID` **site** `UUID() :: String`

Return a random (type 4) UUID represented as a string.

`Thread` **site** `Thread(Top) :: Bot`

Given a site, return a new site which calls the original site in a separate thread. This is necessary when calling a Java site which does not cooperate with Orc's scheduler and may block for an unpredictable amount of time.

A limited number of threads are reserved in a pool for use by this site, so there is a limit to the number of blocking, uncooperative sites that can be called simultaneously.

`Prompt` **site** `Prompt(String) :: String`

Prompt the user for some input. The user may cancel the prompt, in which case the site fails silently. Otherwise their response is returned as soon as it is received.

Example:

```
-- Publishes the user's name
Prompt("What is your name?")
```

`signals` **def** `signals(Integer) :: Top`

Publish the given number of signals, simultaneously.

Example:

```
-- Publishes five signals
signals(5)
```

Implementation.

```
def signals(Integer) :: Top
def signals(n) = if n > 0 then (signal | signals(n-1))
```

`for` **def** `for(Integer, Integer) :: Integer`

Publish all values in the given half-open range, simultaneously.

Example:

```
-- Publishes: 1 2 3 4 5
for(1,6)
```

Implementation.

```
def for(Integer, Integer) :: Integer
def for(low, high) =
  if low >= high then stop
  else ( low | for(low+1, high) )
```

`upto` **def** upto(Integer) :: Integer

`upto(n)` publishes all values in the range `(0..n-1)` simultaneously.

Example:

```
-- Publishes: 0 1 2 3 4
upto(5)
```

Implementation.

```
def upto(Integer) :: Integer
def upto(high) = for(0, high)
```

`fillArray` **def** fillArray[A](Array[A], **lambda** (Integer) :: A) :: Array[A]

Given an array and a function from indices to values, populate the array by calling the function for each index in the array.

For example, to set all elements of an array to zero:

```
-- Publishes: 0 0 0
val a = fillArray(Array(3), lambda (_) = 0)
a.get(0) | a.get(1) | a.get(2)
```

Implementation.

```
def fillArray[A](Array[A], lambda (Integer) :: A)
  :: Array[A]
def fillArray(a, f) =
  val n = a.length()
  def fill(Integer, lambda(Integer) :: A) :: Bot
  def fill(i, f) =
    if i = n then stop
    else ( a.set(i, f(i)) >> stop
         | fill(i+1, f) )
  fill(0, f) ; a
```

`takePubs` **def** takePubs[A](Integer, **lambda** () :: A) :: A

`takePubs(n, f)` calls `f()`, publishes the first `n` values published by `f()` (as they are published), and then halts.

Implementation.

```
def takePubs[A](Integer, lambda () :: A) :: A
def takePubs(n, f) =
  val out = Buffer[A]()
  val c = Counter(n)
  let(
    f() >x>
    if(c.dec() >> out.put(x) >> false
       ; out.closenb() >> true)
  ) >> stop | repeat(out.get)
```

`withLock`

```
def withLock[A](Semaphore, lambda () :: A) :: A
```

Acquire the semaphore and run the thunk, publishing all values published by the thunk. Once the thunk halts, release the semaphore.

Implementation.

```
def withLock[A](Semaphore, lambda () :: A) :: A
def withLock(s, f) =
  val out = Buffer[A]()
  ( s.acquire() >>
    f() >x>
    out.put(x) >>
    stop
  ; s.release() >>
    out.closenb() >>
    stop
  ) | repeat(out.get)
```