Orc User Guide

Orc User Guide

Table of Contents

List of Tables

Introduction

Orc is a programming language designed to make distributed and concurrent programs simple and intuitive to write. Orc expresses orchestration, a type of structured concurrency. It emphasizes the flow of control and gives a global view of a concurrent system. Orc is well-suited for task orchestration, a form of concurrent programming with applications in workflow, business process management, and web service orchestration. Orc provides constructs to orchestrate the concurrent invocation of services while managing time-outs, priorities, and failures of services or communication. To learn more about Orc and run your own Orc programs, visit the website: `http://orc.csres.utexas.edu/` [http://orc.csres.utexas.edu/].

Chapter 1. The Orc Programming Language

1.1. Introduction

This chapter describes the Orc programming language in three steps. In Section 1.2, we discuss a small subset of Orc called Cor. Cor is a pure functional language, which has no features for concurrency, has no state, and does not communicate with external services. Cor introduces us to the parts of Orc that are most familiar from existing programming languages, such as arithmetic operations, variables, conditionals, and functions.

In Section 1.3, we consider Orc itself, which in addition to Cor, comprises external services and combinators for concurrent orchestration of those services. We show how Orc interacts with these external services, how the combinators can be used to build up complex orchestrations from simple base expressions, and how the functional constructs of Cor take on new, subtler behaviors in the concurrent context of Orc.

In Section 1.4, we discuss some additional features of Orc that extend the basic syntax. These are useful for creating large-scale Orc programs, but they are not essential to the understanding of the language.

1.2. Cor: A Functional Subset

In this section we introduce Cor, a pure functional subset of the Orc language. Users of functional programming languages such as Haskell and ML will already be familiar with many of the key concepts of Cor.

A Cor program is an *expression*. Cor expressions are built up recursively from smaller expressions. Cor *evaluates* an expression to reduce it to some simple *value* which cannot be evaluated further, for example a list of numbers or a Boolean truth value. This value is called the *result* of the expression.

In the following subsections we introduce the concepts of Cor. First, we talk about simple constants, such as numbers and truth values, and the operations that we can perform on those values. Then we introduce conditionals (**`if then else`**). Then we introduce variables and binding, as a way to give a name to the value of an expression. After that, we talk about constructing data structures, and examining those structures using patterns. Lastly, we introduce functions.

Table 1.1 describes the syntax of Cor. Each part of the syntax is explained in a subsequent section.

Table 1.1. Syntax of the Functional Subset of Orc (Cor)

E ::=		*Expression*	
	C		*constant value*
|	X		*variable*
|	`(` E `,` ... `,` E `)`		*tuple*
|	`[` E `,` ... `,` E `]`		*list*
|	X`(` E `,` ... `,` E `)`		*function call*
|	E *op* E		*operator*
|	**`if`** E **`then`** E **`else`** E		*conditional*
|	**`lambda`** `(` P `,` ... `,` P `)` = E		*closure*
|	D E		*scoped declaration*
C ::=	*Boolean* | *Number* | *String*	*Constant*	
X ::=	*Identifier*	*Identifier*	
D ::=		*Declaration*	
	`val` P = E		*value declaration*
|	**`def`** X`(` P `,` ... `,` P `)` = E		*function declaration*
P ::=		*Pattern*	
	X		*variable*
|	C		*constant*
|	_		*wildcard*
|	`(` P `,` ... `,` P `)`		*tuple*
|	`[` P `,` ... `,` P `]`		*list*

Where relevant, syntactic constructs are ordered by precedence, from highest to lowest. For example, among expressions, calls have higher precedence than operators, which in turn have higher precedence than conditionals.

1.2.1. Constants

The simplest expression one can write is a constant. It evaluates trivially to that constant value.

Cor has three types of constants, and thus for the moment three types of values:

- Boolean: `true` and `false`
- Number: `5, -1, 2.71828, ...`
- String: `"orc"`, `"ceci n'est pas une |"`

1.2.2. Operators

Cor has a standard set of arithmetic, logical, and comparison operators. As in most other programming languages, they are written in the usual infix style. They have Java-like operator precedence, which can be overridden by adding parentheses.

Examples

- `1+2` evaluates to `3`.

- `(98+2)*17` evaluates to `1700`.
- `4 = 20 / 5` evaluates to `true`.
- `3-5 >= 5-3` evaluates to `false`.
- `true && (false || true)` evaluates to `true`.
- `"leap" + "frog"` evaluates to `"leapfrog"`.

Here is the full set of operators that Cor supports:

Table 1.2. Operators of Cor

Arithmetic		Comparison		Logical		String	
`+`	addition	`=`	equality	`&&`	logical and	`+`	concatenation
`-`	subtraction	`/=`	inequality	`||`	logical or		
`*`	multiplication	`<`	less than	`~`	logical not		
`/`	division	`>`	greater than				
`%`	modulus	`<=`	less than or equal				
`**`	exponent	`>=`	greater than or equal				

There is also a unary negation operator, written `-`, for example `-(2 ** 5)`.

The `=` operator can compare values of any type. Values of different type are always unequal; for example, `10 = true` evaluates to `false`.

1.2.2.1. Silent Expression

There are situations where an expression evaluation is stuck, because it is attempting to perform some impossible operation and cannot compute a value. In that case, the expression is *silent*. An expression is also silent if it depends on the result of a silent subexpression. For example, the following expressions are silent: `10/0`, `6 + false`, `3 + 1/0`, `4 + true = 5`.

Cor is a dynamically typed language. A Cor implementation does not statically check the type correctness of an expression; instead, an expression with a type error is simply silent when it is evaluated.

1.2.3. Conditionals

A conditional expression in Cor is of the form **`if`** E **`then`** F **`else`** G. Its meaning is similar to that in other languages: the value of the expression is the value of F if and only if E evaluates to true, or the value of G if and only if E evaluates to false. Note that G is not evaluated at all if E evaluates to true, and similarly F is not evaluated at all if E evaluates to false. Thus, for example, evaluation of **`if`** `true` **`then`** `2+3` **`else`** `1/0` does not evaluate `1/0`; it only evaluates `2+3`.

Unlike other languages, expressions in Cor may be silent. If E is silent, then the entire expression is silent. And if E evaluates to true but F is silent (or if E evaluates to false and G is silent) then the expression is silent.

Note that conditionals have lower precedence than any of the operators. For example, **`if`** `false` **`then`** `1` **`else`** `2 + 3` is equivalent to **`if`** `false` **`then`** `1` **`else`** `(2 + 3)`, not `(`**`if`**` false `**`then`**` 1 `**`else`**` 2) + 3`.

The behavior of conditionals is summarized by the following table (*v* denotes a value).

E	F	G	**if E then F else G**
`true`	*v*	-	*v*
`true`	silent	-	silent
`false`	-	*v*	*v*
`false`	-	silent	silent
silent	-	-	silent

Examples

- **if** `true` **then** `4` **else** `5` evaluates to 4.
- **if** `2 < 3 && 5 < 4` **then** `"blue"` **else** `"green"` evaluates to "green".
- **if** `true || "fish"` **then** `"yes"` **else** `"no"` is silent.
- **if** `false || false` **then** `4+5` **else** `4+true` is silent.
- **if** `0 < 5` **then** `0/5` **else** `5/0` evaluates to 0.

1.2.4. Variables

A *variable* can be bound to a value. A *declaration* binds one or more variables to values. The simplest form of declaration is **val**, which evaluates an expression and binds its result to a variable. Declarations follow the rules of lexical scoping [http://en.wikipedia.org/wiki/Lexical_scope].

```
val x = 1 + 2
val y = x + x
```

These declarations bind variable `x` to 3 and variable `y` to 6.

If the expression on the right side of a **val** is silent, then the variable is not bound, but evaluation of other declarations and expressions continues. If an evaluated expression depends on that variable, that expression is silent.

```
val x = 1/0
val y = 4+5
if false then x else y
```

Evaluation of the declaration **val** `y = 4+5` and the expression **if** `false` **then** `x` **else** `y` may continue even though `x` is not bound. The expression evaluates to 9.

1.2.5. Data Structures

Cor supports two basic data structures, *tuples* and *lists*.

A *tuple expression* is a comma-separated sequence of at least two expressions, enclosed by parentheses. Each expression is evaluated; the value of the whole tuple expression is a tuple containing each of these values in order. If any of the expressions is silent, then the whole tuple expression is silent.

Examples

- `(1+2, 7)` evaluates to `(3,7)`.

- `("true" + "false", true || false, true && false)` evaluates to `("truefalse", true, false)`.
- `(2/2, 2/1, 2/0)` is silent.

A *list expression* is a comma-separated sequence of expressions enclosed by square brackets. It may be of any length, including zero. Each expression is evaluated; the value of the whole list expression is a list containing each of these values in order. If any of the expressions is silent, then the whole list expression is silent.

Examples

- `[1,2+3]` evaluates to `[1,5]`.
- `[true && true]` evaluates to `[true]`.
- `[]` evaluates vacuously to `[]`, the empty list.
- `[5, 5 + true, 5]` is silent.

There is also a concatenation (*cons*) operation on lists, written F`:`G, where F and G are expressions. Its result is a new list whose first element is the value of F and whose remaining elements are the list value of G. The `:` operator is right associative, so F`:`G`:`H is F`(`G`:`H`)`.

Examples

- `(1+3):[2+5,6]` evaluates to `[4,7,6]`.
- `2:2:5:[]` evaluates to `[2,2,5]`.
- Suppose `t` is bound to [3,5]. Then `1:t` evaluates to `[1,3,5]`.
- `2:3` is silent, because `3` is not a list.

1.2.6. Patterns

We have seen how to construct data structures. But how do we examine them, and use them? We use *patterns*.

A pattern is a powerful way to bind variables. When writing **`val`** declarations, instead of just binding one variable, we can replace the variable name with a more complex pattern that follows the structure of the value, and matches its components. A pattern's structure is called its *shape*; a pattern may take the shape of any structured value. A pattern can hierarchically match a value, going deep into its structure. It can also bind an entire structure to a variable.

Examples

- **`val`** `(x,y) = (2+3,2*3)` binds `x` to 5 and `y` to 6.
- **`val`** `[a,b,c] = ["one", "two", "three"]` binds `a` to "one", `b` to "two", and `c` to "three".
- **`val`** `((a,b),c) = ((1, true), (2, false))` binds `a` to 1, `b` to `true`, and `c` to `(2,false)`.

Patterns are *linear*; that is, a pattern may mention a variable name at most once. For example, `(x,y,x)` is not a valid pattern.

Note that a pattern may fail to match a value, if it does not have the same shape as that value. In a **`val`** declaration, this has the same effect as evaluating a silent expression. No variable in the pattern is bound, and if any one of those variables is later evaluated, it is silent.

It is often useful to ignore parts of the value that are not relevant. We can use the wildcard pattern, written `_`, to do this; it matches any shape and binds no variables.

Examples

- **`val`** `(x,_,_) = (1,(2,2),[3,3,3])` binds `x` to 1.
- **`val`** `[[_,x],[_,y]] = [[1,3],[2,4]]` binds `x` to 3 and `y` to 4.

1.2.7. Functions

Like most other programming languages, Cor provides the capability to define *functions*, which are expressions that have a defined name, and have some number of parameters. Functions are declared using the keyword **`def`**, in the following way.

```
def Add(x,y) = x+y
```

The expression on the right of the `=` is called the *body* of the function.

After defining the function, we can *call* it. A call looks just like the left side of the declaration except that the variable names (the *formal parameters*) have been replaced by expressions (the *actual parameters*).

To evaluate a call, we treat it as a sequence of **`val`** declarations associating the formal parameters with the actual parameters, followed by the body of the function.

```
{- Evaluation of Add(1+2,3+4) -}
val x = 1+2
val y = 3+4
x+y
```

Examples

- `Add(10,10*10)` evaluates to `110`.
- `Add(Add(5,3),5)` evaluates to `13`.

Notice that the evaluation strategy of functions allows a call to proceed even if some of the actual parameters are silent, so long as the values of those actual parameters are not used in the evaluation of the body.

```
def cond(b,x,y) = if b then x else y
cond(true, 3, 5/0)
```

This evaluates to `3` even though `5/0` is silent, because `y` is not needed.

A function definition or call may have zero arguments, in which case we write `()` for the arguments.

```
def Zero() = 0
```

1.2.7.1. Recursion

Functions can be recursive; that is, the name of a function may be used in its own body.

```
def Sumto(n) = if n < 1 then 0 else n + Sumto(n-1)
```

Then, `Sumto(5)` evaluates to 15.

Mutual recursion is also supported.

```
def Even(n) =
  if (n > 0) then Odd(n-1)
  else if (n < 0) then Odd(n+1)
  else true
def Odd(n) =
  if (n > 0) then Even(n-1)
  else if (n < 0) then Even(n+1)
  else false
```

There is no special keyword for mutual recursion; any contiguous sequence of function declarations is assumed to be mutually recursive.

1.2.7.2. Closures

Functions are actually values, just like any other value. Defining a function creates a special value called a *closure*; the name of the function is a variable and its bound value is the closure. Thus, a closure can be put into a data structure, or bound to some other variable, just like any other value.

```
def a(x) = x-3
def b(y) = y*4
val funs = (a,b)
```

Like any other value, a closure can be passed as an argument to another function. This means that Cor supports *higher-order* functions.

```
def Onetwosum(f) = f(1) + f(2)
def Triple(x) = x * 3
```

Then, `Onetwosum(Triple)` is `Triple(1) + Triple(2)`, which is `1 * 3 + 2 * 3` which evaluates to `9`.

Since all declarations (including function declarations) in Cor are lexically scoped, these closures are *lexical closures*. This means that when a closure is created, if the body of the function contains any variables other than the formal parameters, the bindings for those variables are stored in the closure. Then, when the closure is called, the evaluation of the function body uses those stored variable bindings.

1.2.7.3. Lambda

Sometimes one would like to create a closure directly, without bothering to give it a name. There is a special keyword **`lambda`** for this purpose. By writing a function definition without the keyword **`def`**

and replacing the function name with the keyword **lambda**, that definition becomes an expression which evaluates to a closure.

```
def Onetwosum(f) = f(1) + f(2)

Onetwosum( lambda(x) = x * 3 )
{-
  identical to:
  def triple(x) = x * 3
  onetwosum(triple)
-}
```

Then, `Onetwosum( lambda(x) = x * 3 )` evaluates to 9.

1.2.7.4. Clauses

The combination of functions and pattern matching offers a powerful capability: *clausal* definition of functions. We can define expressions which execute different code depending on the structure of their arguments.

Here's an example.

```
def Sum([]) = 0
def Sum(h:t) = h + Sum(t)
```

`Sum(L)` publishes the sum of the numbers in the list `L`. It has two clauses: one which matches the empty list, and one which matches any nonempty list. If its argument is an empty list, it returns 0, the appropriate sum for an empty list. If the argument is a nonempty list, it adds the first element of that list to the sum of all of the other elements. In this way, it recursively finds the sum of the list.

A function may have multiple clauses, each of which has a sequence of patterns to match each argument, and a body expression. Naturally, all clauses of a function must have the same number of arguments. Any contiguous sequence of definitions with the same name and different arguments is interpreted as a clausal definition, where each individual declaration is a clause of the larger function.

When the function is called, the clauses are tried in the order in which they appear until a match is found. If no clause matches, the call remains silent.

We allow a new form of pattern which is very useful in clausal definition of functions: a constant pattern. A constant pattern is a match only for the same constant value. We can use this to define the "base case" of a recursive function in a straightforward way.

```
{- Fibonacci numbers -}
def Fib(0) = 1
def Fib(1) = 1
def Fib(n) = if (n < 0) then 0 else Fib(n-1) + Fib(n-2)
```

This definition of the Fibonacci function is straightforward, but slow, due to the repeated work in recursive calls to `Fib`. We can define a linear-time version, again with the help of pattern matching:

```
{- Alternate definition of the Fibonacci function -}
```

```
{- A helper function: find the pair (Fibonacci(n-1), Fibonacci(n)) -}
def H(0) = (1,1)
def H(n) = H(n-1) >(x,y)> (y,x+y)

def Fib(n) = if (n < 0) then 0 else H(n) >(x,_)> x
```

As a more complex example of matching, consider the following function which finds the first `n` elements of a list (the list is assumed to be at least `n` elements long).

```
def Take(0,_) = []
def Take(n,h:t) = h:(Take(n-1,t))
```

Mutual recursion and clausal definitions are allowed to occur together. For example, this function takes a list and computes a new list in which every other element is repeated:

```
def Stutter([]) = []
def Stutter(h:t) = h:h:Mutter(t)
def Mutter([]) = []
def Mutter(h:t) = h:Stutter(t)
```

`Stutter([1,2,3])` evaluates to `[1,1,2,3,3]`.

Clauses of mutually recursive functions may also be interleaved, to make them easier to read.

```
def Even(0) = true
def Odd(0) = false
def Even(n) = Odd(if n > 0 then n-1 else n+1)
def Odd(n) = Even(if n > 0 then n-1 else n+1)
```

1.2.8. Comments

Cor has two kinds of comments.

A line which begins with two dashes (`--`), preceded only by whitespace, is a single line comment. The region from the two dashes to the next encountered newline, inclusive, is ignored.

```
-- This is a single line comment.
  -- This is also a single line comment.
```

Multi-line comments are enclosed by matching braces of the form *{- -}*. Multi-line comments may be nested. They may appear anywhere, even in the middle of an expression.

```
{-
   This is a
   multiline comment.
-}

{- Multiline comments {- can be nested -} -}
```

```
{- They may appear anywhere, -}
1 + {- even in the middle of an expression. -} 2 + 3
```

1.3. Orc: Orchestrating services

Cor is a pure declarative language. It has no state, since variables are bound at most once and cannot be reassigned. Evaluation of an expression results in at most one value. It cannot communicate with the outside world except by producing a value. The full Orc language transcends these limitations by incorporating the orchestration of external services. We introduce the term *site* to denote an external service which can be called from an Orc program.

As in Cor, an Orc program is an *expression*; Orc expressions are built up recursively from smaller expressions. Orc is a superset of Cor, i.e., all Cor expressions are also Orc expressions. Orc expressions are *executed*, rather than evaluated; an execution may call external services and *publish* some number of values (possibly zero). Different executions of the same expression may have completely different behaviors; they may call different services, may receive different responses from the same site, and may publish different values. Expressions in the functional subset, though, will display the same behavior in all executions.

In the following sections we discuss the features of Orc. First, we discuss how Orc communicates with external services. Then we introduce Orc's concurrency *combinators*, which combine expressions into larger orchestrations and manage concurrent executions. We have already discussed the functional subset of Orc in our coverage of Cor, but we reprise some of those topics; some Cor constructs exhibit new behaviors in the concurrent, stateful context of Orc.

The following figure summarizes the syntax of Orc as an extension of the syntax of Cor. The original Cor grammar rules are abbreviated by ellipses (...).

Table 1.3. Basic Syntax of Orc

D ::= ...	*Declaration*	
| **site** X = *address*		*site declaration*
C ::= ...	*Constant*	
| **signal**		*signal value*
E ::= ...	*Expression*	
| **stop**		*silent expression*
| E **>**P**>** E		*sequential combinator*
| E **|** E		*parallel combinator*
| E **<**P**<** E		*pruning combinator*
| E **;** E		*otherwise combinator*

1.3.1. Communicating with external services

An Orc expression may be a site call. Sites are called using the same syntax as a function call, but with a slightly different meaning. Sites are introduced and bound to variables by a special declaration.

1.3.1.1. Calling a site

Suppose that the variable `Google` is bound to a site which invokes the Google search engine service in "I'm Feeling Lucky" mode. A call to `Google` looks just like a function call. Calling `Google` requests the URL of the top result for the given search term.

```
{- Get the top search result for "computation orchestration" -}
Google("computation orchestration")
```

Once the Google search service determines the top result, it sends a response. The site call then publishes that response. Note that the service might not respond: Google's servers might be down, the network might be down, or the search might yield no result URL.

A site call sends only a single request to an external service and receives at most one response. These restrictions have two important consequences. First, all of the information needed for the call must be present before contacting the service. Thus, site calls are strict; all arguments must be bound before the call can proceed. If any argument is silent, the call never occurs. Second, a site call publishes at most one value, since at most one response is received.

A call to a site has exactly one of the following effects:

1. The site returns a value, called its *response*.

2. The site communicates that it will never respond to the call; we say that the call has *halted*

3. The site neither returns a value nor indicates that the call has halted; we say that the call is *pending*.

In the last two cases, the site call is said to be silent. However, unlike a silent expression in Cor, a silent site call in Orc might perform some meaningful computation or communication; silence does not necessarily indicate an error. Since halted site calls and pending site calls are both silent, they cannot usually be distinguished from each other; only the otherwise combinator can tell the difference.

A site is a value, just like an integer or a list. It may be bound to a variable, passed as an argument, or published by an execution, just like any other value.

```
{-
  Create a search site from a search engine URL,
  bind the variable Search to that site,
  then use that site to search for a term.
-}
val Search = SearchEngine("http://www.google.com/")
Search("first class value")
```

A site is sometimes called only for its effect on the external world; its return value is irrelevant. Many sites which do not need to return a meaningful value will instead return a *signal*: a special value which carries no information (analogous to the unit value `()` in ML). The signal value can be written as **`signal`** within Orc programs.

```
{-
  Use the 'println' site to print a string, followed by
  a newline, to an output console.
  The return value of this site call is a signal.
-}
println("Hello, World!")
```

1.3.1.2. Declaring a site

A **`site`** declaration makes some service available as a site and binds it to a variable. The service might be an object in the host language (e.g. a class instance in Java), or an external service on the web which

is accessed through some protocol like SOAP or REST, or even a primitive operation like addition. Many useful sites are already defined in the Orc standard library, documented in Appendix B. For now, we present a simple type of site declaration: using an object in the host language as a site.

The following example instantiates a Java object to be used as a site in an Orc program, specifically a Java object which provides an asynchronous buffer service. The declaration uses a fully qualified Java class name to find and load a class, and creates an instance of that class to provide the service.

```
{-  Define the Buffer site  -}
site Buffer = orc.lib.state.Buffer
```

1.3.2. The concurrency combinators of Orc

Orc has four *combinators*: parallel, sequential, pruning, and otherwise. A combinator forms an expression from two component expressions. Each combinator captures a different aspect of concurrency. Syntactically, the combinators are written infix, and have lower precedence than operators, but higher precedence than conditionals or declarations.

1.3.2.1. The parallel combinator

Orc's simplest combinator is **|**, the parallel combinator. Orc executes the expression F **|** G, where F and G are Orc expressions, by executing F and G concurrently. Whenever F or G communicates with a service or publishes a value, F **|** G does so as well.

```
-- Publish 1 and 2 in parallel
1 | 1+1
```

```
{-
  Access two search sites, Google and Yahoo, in parallel.

  Publish any results they return.

  Since each call may publish a value, the expression
  may publish up to two values.
-}
Google("cupcake") | Yahoo("cupcake")
```

The parallel combinator is fully associative: (F **|** G) **|** H and F **|** (G **|** H) and F **|** G **|** H are all equivalent.

It is also commutative: F **|** G is equivalent to G **|** F.

```
-- Publish 1, 2, and 3 in parallel
1+0 | 1+1 | 1+2
```

1.3.2.2. The sequential combinator

Now that we have expressions which publish multiple values, what can we do with those publications? The sequential combinator, written F **>x>** G, combines the expression F, which may publish some values, with another expression G, which will use the values as they are published; `x` transmits the values from F to G.

The execution of F **>x>** G starts by executing F. Whenever F publishes a value, a new copy of G is executed in parallel with F (and with any previous copies of G); in that copy of G, variable x is bound to the value published by F. Values published by copies of G are published by the whole expression, but the values published by F are not published by the whole expression; they are consumed by the variable binding.

```
-- Publish 1 and 2 in parallel
(0 | 1) >n> n+1


-- Publish 3 and 4 in parallel
2 >n> (n+1 | n+2)


-- Publish 0, 1, 2 and 3 in parallel
(0 | 2) >n> (n | n+1)


-- Prepend the site name to each published search result
-- The cat site concatenates any number of arguments into one string
  Google("cupcake") >s> cat("Google: ", s)
| Yahoo("cupcake") >s> cat("Yahoo: ", s)
```

The sequential combinator may be written as F **>**P**>** G, where P is a pattern instead of just a variable name. Any value published by F is matched against the pattern P. If this match is successful, a new copy of G is started with all of the bindings from the match. Otherwise, the published value is simply ignored, and no new copy of G is executed.

```
-- Publish 3, 6, and 9 in arbitrary order.
(3,6,9)  >(x,y,z)>  ( x | y | z )


-- Filter out values of the form (_,false)
( (4,true) | (5,false) | (6,true) )  >(x,true)> x
-- Publishes 4 and 6
```

We may also omit the variable entirely, writing **>>** . This is equivalent to using a wildcard pattern: **>_>**

We may want to execute an expression just for its effects, and hide all of its publications. We can do this using **>>** together with the special expression **stop**, which is always silent.

```
{-
  Print two strings to the console,
  but don't publish the return values of the calls.
-}
( println("goodbye") | println("world") ) >> stop
```

The sequential combinator is right associative: F **>x>** G **>y>** H is equivalent to F **>x>** (G **>y>** H). It has higher precedence than the parallel combinator: F **>x>** G **|** H is equivalent to (F **>x>** G) **|** H.

The right associativity of the sequential combinator makes it easy to bind variables in sequence and use them together.

```
{-
  Publish the cross product of {1,2} and {3,4}:
  (1,3), (1,4), (2,3), and (2,4).
-}
(1 | 2) >x> (3 | 4) >y> (x,y)
```

1.3.2.3. The pruning combinator

The pruning combinator, written F <x< G, allows us to block a computation waiting for a result, or terminate a computation. The execution of F <x< G starts by executing F and G in parallel. Whenever F publishes a value, that value is published by the entire execution. When G publishes its first value, that value is bound to x in F, and then the execution of G is immediately *terminated*. A terminated expression cannot call any sites or publish any values.

During the execution of F, any part of the execution that depends on x will be suspended until x is bound (to the first value published by G). If G never publishes a value, that part of the execution is suspended forever.

```
-- Publish either 5 or 6, but not both
x+2 <x< (3 | 4)
```

```
-- Query Google and Yahoo for a search result
-- Print out the result that arrives first; ignore the other result
println(result) <result< ( Google("cupcake") | Yahoo("cupcake") )
```

Though a terminated execution may not make any new calls, the calls that it has already made will continue normally; their responses are simply ignored. This may have surprising consequences when a call has side effects, as in the following example.

```
{-
  This example actually prints both "true" and "false" to the
  console, regardless of which call responds first.
-}
stop <x< println("true") | println("false")
```

Both of the `println` calls are initiated before either one of them publishes a value and terminates the expression. Once the expression is terminated, no new calls occur, but the other `println` call still proceeds and still has the effect of printing its message to the console.

The pruning combinator may include a full pattern P instead of just a variable name. Any value published by G is matched against the pattern P. If this match is successful, then G terminates and all of the bindings of the pattern P are made in F. Otherwise, the published value is simply ignored and G continues to execute.

```
-- Publish either 9 or 25, but not 16.
x*x <(x,true)< ( (3,true) | (4,false) | (5,true) )
```

Note that even if `(4,false)` is published before `(3,true)` or `(5,true)`, it is ignored. The right side continues to execute and will publish one of `(3,true)` or `(5,true)`.

The pruning combinator is left associative: F <x< G <y< H is equivalent to (F <x< G) <y< H. It has lower precedence than the parallel combinator: F <x< G | H is equivalent to F <x< (G | H).

1.3.2.4. The otherwise combinator

Orc has a fourth concurrency combinator: the *otherwise* combinator, written F `;` G. The execution of F `;` G proceeds as follows. First, F is executed. If F *completes*, and has not published any values, then G executes. If F did publish one or more values, then G is ignored. The publications of F `;` G are those of F if F publishes, or those of G otherwise.

We determine when an expression completes using the following rules.

- A Cor expression completes when it is fully evaluated; if it is silent, it completes immediately.
- A site call completes when it has published a value or halted.
- F `|` G completes when its subexpressions F and G have both completed.
- F `>x>` G completes when F has completed and all instantiated copies of G have completed.
- F `<x<` G completes when F has completed, and G has either completed or published a value. Also, if G completes without publishing a value, then all expressions in F which use `x` also complete, since they will never be able to proceed.
- F `;` G completes either when F has published a value and subsequently completed, or when F and G have both completed.
- **`stop`** completes immediately.

The otherwise combinator is fully associative, so F `;` G `;` H and (F `;` G) `;` H and F `;` (G `;` H) are all equivalent. It has lower precedence than the other three combinators.

The otherwise combinator was not present in the original formulation of the Orc concurrency calculus; it has been added to support computation and iteration over strictly finite data. Sequential programs conflate the concept of producing a value with the concept of completion. Orc separates these two concepts; variable binding combinators like `>x>` and `<x<` handle values, whereas `;` detects the completion of an execution.

1.3.3. Revisiting Cor expressions

Some Cor expressions have new behaviors in the context of Orc, due to the introduction of concurrency and of sites.

1.3.3.1. Operators

The arithmetic, logical, and comparison operators are actually calls to sites, simply written in infix style with the expected operator symbols. For example, `2+3` is actually `(+)(2,3)`, where `(+)` is a primitive site provided by the language itself. All of the operators can be used directly as sites in this way; the name of the site is the operator enclosed by parentheses, e.g. `(**)`, `(>=)`, etc. Negation (unary minus) is named `(0-)`.

1.3.3.2. Conditionals

The conditional expression **`if`** E **`then`** F **`else`** G is actually a derived form based on a site named **`if`**. The **`if`** site takes a boolean argument; it returns a signal if that argument is `true`, or remains silent if the argument is `false`.

if E **then** F **else** G is equivalent to `( if(b) >>` F `| if(~b) >>` G`) <b<` E.

When the **else** branch of a conditional is unneeded, we can write **if** F **then** G, with no **else** branch. This is equivalent to **if**(E) **>>** F.

1.3.3.3. val

The declaration **val** x = G, followed by expression F, is actually just a different way of writing the expression F **<**x**<** G. Thus, **val** shares all of the behavior of the pruning combinator, which we have already described. (This is also true when a pattern is used instead of variable name x).

1.3.3.4. Nesting Orc expressions

The execution of an Orc expression may publish many values. What does such an expression mean in a context where only one value is expected? For example, what does `2 + (3 | 4)` publish?

The specific contexts in which we are interested are as follows (where E is any Orc expression):

E *op* E	*operand*
if E **then** ...	*conditional test*
X(... , E , ...)	*call argument*
(... , E , ...)	*tuple element*
[... , E , ...]	*list element*

Whenever an Orc expression appears in such a context, it executes until it publishes its first value, and then it is terminated. The published value is used in the context as if it were the result of evaluating the expression.

```
-- Publish either 5 or 6
2 + (3 | 4)
```

```
-- Publish exactly one of 0, 1, 2 or 3
(0 | 2) + (0 | 1)
```

To be precise, whenever an Orc expression appears in such a context, it is treated as if it was on the right side of a pruning combinator, using a fresh variable name to fill in the hole. Thus, C[E] (where E is the Orc expression and C is the context) is equivalent to the expression C[x] **<**x**<** E.

1.3.3.5. Functions

The body of a function in Orc may be any Orc expression; thus, function bodies in Orc are executed rather than evaluated, and may engage in communication and publish multiple values.

A function call in Orc, as in Cor, binds the values of its actual parameters to its formal parameters, and then executes the function body with those bindings. Whenever the function body publishes a value, the function call publishes that value. Thus, unlike a site call, or a pure functional Cor call, an Orc function call may publish many values.

```
-- Publish all integers in the interval 1..n, in arbitrary order.
```

```
def range(n) = if (n > 0) then (n | range(n-1)) else stop

-- Publish 1, 2, and 3 in arbitrary order.
range(3)
```

In the context of Orc, function calls are not strict. When a function call executes, it begins to execute the function body immediately, and also executes the argument expressions in parallel. When an argument expression publishes a value, it is terminated, and the corresponding formal parameter is bound to that value in the execution of the function body. Any part of the function body which uses a formal parameter that has not yet been bound suspends until that parameter is bound to a value.

1.3.4. Time

Orc is designed to communicate with the external world, and one of the most important characteristics of the external world is the passage of time. Orc implicitly interacts with the passage of time by calling external services which take time to respond. However, Orc can also explicitly wait for some amount of time, using the special site `Rtimer`.

The site `Rtimer` is a relative timer. It takes as an argument a number of milliseconds to wait. It waits for exactly that amount of time, and then responds with a signal.

```
-- Print "red", wait for 3 seconds (3000 ms), and then print "green"
println("red") >> Rtimer(3000) >> println("green") >> stop
```

The following example defines a metronome, which publishes a signal once every `t` milliseconds, indefinitely.

```
def metronome(t) = signal | Rtimer(t) >> metronome(t)
```

We can also use `Rtimer` together with the pruning combinator to enforce a timeout.

```
{-
  Publish the result of a Google search.
  If it takes more than 5 seconds, time out.
-}
result
  <result< ( Google("impatience")
           | Rtimer(5000) >> "Search timed out.")
```

We present many more examples of programming techniques using real time in Chapter 2.

1.4. Advanced Features of Orc

In this section we introduce some advanced features of Orc. These include curried function definitions and curried calls, writing an arbitrary expression as the target of a call, a special syntax for writing calls in an object-oriented style, extensions to pattern matching, and new forms of declarations, and ML-style datatypes.

The following figure summarizes further extensions to the syntax of Orc. The Orc and Cor grammar rules presented earlier are abbreviated by ellipses (...). The item G+ means "one or more instances of G concatenated together".

Table 1.4. Advanced Syntax of Orc

E	::=	...	*Expression*
	|	E G+	*generalized call*
G	::=		*Argument group*
		`(` E `,` ... `,` E `)`	*curried arguments*
	|	`.` *field*	*field access*
P	::=	...	*Pattern*
	|	`!`P	*publish pattern*
	|	P **`as`** X	*as pattern*
	|	X`(` P `,` ... `,` P `)`	*datatype pattern*
D	::=	...	*Declaration*
	|	**`class`** X = *classname*	*class declaration*
	|	**`type`** X = DT **`|`** ... **`|`** DT	*datatype declaration*
	|	**`include`** `"` *filename* `"`	*inclusion*
DT	::=	X`(_,` ... `,_)`	*Datatype*

1.4.1. Special call forms

1.4.1.1. The `.` notation

In many object-oriented programming languages, one calls a method or accesses a field of an object using the dot operator; for example, `obj.m()` calls the method `m` of the object `obj`.

There is a special kind of site call in Orc which serves a similar purpose. One may write `x.msg`, for any identifiers `x` and `msg`. This treats the value bound to `x` as a site, and calls it with a special *message* value `msg`. If the site understands the message `msg` (for example, if `x` is bound to a Java object with a field called `msg`), the site interprets the message and responds with some appropriate value. If the site does not understand the message sent to it, it does not respond, and no publication occurs. If `x` cannot be interpreted as a site, no call is made.

Typically this capability is used so that sites may be syntactically treated like objects, with multiple methods and fields. For example, a channel `c` might understand the messages `get` and `put`, to get values from and put values on that channel, respectively. Such calls would be written `c.get()`, or `c.put(6)`.

A call such as `c.put(6)` actually occurs in two steps. First `c.put` sends the message `put` to the site `c`; this publishes a site whose only purpose is to put values on the channel. Next, that site is called on the argument `6`, sending 6 on the channel. Readers familiar with functional programming will recognize this technique as *currying*.

1.4.1.2. Currying

It is sometimes useful to *stage* the arguments to a function; that is, rather than writing a function on two arguments, one instead writes a function on one argument which returns a function taking the second argument and performing the remainder of the evaluation.

This technique is known as *currying* and it is common in functional programming languages. We can write curried functions using closures. Suppose we want to define a curried addition function on two arguments, and later apply that function to the arguments 3 and 4. We could write such a program in the following way:

```
def Sum(a) = ( lambda(b) = a+b )
val f = Sum(3)
f(4)
```

This defines a function `Sum` which, given an argument `a`, creates a function which will take an argument `b` and add `a` to `b`. It then creates the function which adds `3` to its argument, binds that to `f`, and then invokes `f` on `4` to yield `3+4`.

When defining a curried function, we have abstracted it in two steps, and when applying it we have written two separate calls. However, this is verbose and not very clear. Orc has a special syntax for curried function definitions and curried applications that will simplify both of these steps. Function definitions may have multiple argument sequences; they are enclosed in parentheses and concatenated. Curried function calls chain together multiple applications in a similar way. Here is the previous program, written in this simplified syntax:

```
def Sum(a)(b) = a+b
Sum(3)(4)
```

Naturally, this syntax is backwards compatible; e.g. both of the following programs are also equivalent:

```
def Sum(a) = ( lambda(b) = a+b )
Sum(3)(4)
```

```
def Sum(a)(b) = a+b
val f = Sum(3)
f(4)
```

1.4.2. Extensions to pattern matching

1.4.2.1. Publish pattern

A publish pattern, written `!p`, will publish the value that matches the pattern `p` if the match is successful.

```
(1,2,3)  >(x,!y,!z)> stop
```

This publishes `2` and `3`.

Note that a publish pattern will not publish if the overall match fails, even if its particular match succeeds:

```
( (1,2,3) | (4,5,6) )  >(1,!x,y)>  stop
```

This publishes only `2`. Even though the pattern `x` matches the value `5`, the overall pattern `(1,!x,y)` does not match the value `(4,5,6)`, so `5` is not published.

1.4.2.2. As pattern

In taking apart a value with a pattern, it is often useful to capture intermediate results.

```
val (a,b) = ((1,2),(3,4))
val (ax,ay) = a
val (bx,by) = b
```

We can use the **as** keyword to simplify this program fragment, giving a name to an entire sub-pattern. Here is an equivalent version of the above code.

```
val ((ax,ay) as a, (bx,by) as b) = ((1,2),(3,4))
```

1.4.3. Datatypes

We have seen Orc's predefined data structures: tuples and lists. Orc also provides the capability for programmers to define their own data structures, using a feature adopted from the ML/Haskell language family called *datatypes* (also called variants or tagged sums).

Datatypes are defined using the **type** declaration:

```
type tree = Node(_,_,_) | Empty()
```

This declaration defines two new sites named `Node` and `Empty`. `Node` takes three arguments, and publishes a *tagged value* wrapping those arguments. `Empty` takes no arguments and does the same.

Once we have created these tagged values, we use a new pattern called a datatype pattern to match them and unwrap the arguments:

```
type tree = Node(_,_,_)
{- Build up a small binary tree -}
val l = Node(Empty(), 0, Empty())
val r = Node(Empty(), 2, Empty())
val t = Node(l,1,r)

{- And then match it to extract its contents -}
t >Node(l,j,r)>
l >Node(_,i,_)>
r >Node(_,k,_)>
( i | j | k )
```

One pair of datatypes is so commonly used that it is already predefined in the standard library: `some(_)` and `none`. These are used as return values for calls that need to distinguish between successfully returning a value (`some(v)`), and successfully completing but having no meaningful value to return (`none()`). For example, a lookup function might return `some(result)` if it found a result, or return `none()` if it successfully performed the lookup but found no suitable result.

1.4.4. New forms of declarations

1.4.4.1. `class` declaration

When Orc is run on top of an object-oriented programming language, classes from that language may be used as sites in Orc itself, via the **class** declaration.

```
{- Use the String class from Java's standard library as a site -}
class String = java.lang.String
val s = String("foo")
s.concat("bar")
```

This program binds the variable `String` to Java's String class. When it is called as a site, it constructs a new instance of String, passing the given arguments to the constructor.

This instance of String is a Java object; its methods are called and its fields are accessed using the dot (`.`) notation, just as one would expect in Java. For complete details of how Orc interacts with Java, see Java Integration.

1.4.4.2. `include` declaration

It is often convenient to group related declarations into units that can be shared between programs. The **`include`** declaration offers a simple way to do this. It names a source file containing a sequence of Orc declarations; those declarations are incorporated into the program as if they had textually replaced the include declaration. An included file may itself contain **`include`** declarations.

```
{- Contents of fold.inc -}
def foldl(f,[],s) = s
def foldl(f,h:t,s) = foldl(f,t,f(h,s))

def foldr(f,l,s) = foldl(f,rev(l),s)
```

```
{- This is the same as inserting the contents of fold.inc here -}
include "fold.inc"

def sum(L) = foldl(lambda(a,b) = a+b, L, 0)

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

Note that these declarations still obey the rules of lexical scope. Also, Orc does not detect shared declarations; if the same file is included twice, its declarations occur twice.

Chapter 2. Programming Methodology

In Chapter 1, we described the syntax and semantics of the Orc language. Now, we turn our attention to how the language is used in practice, with guidelines on style and programming methodology, including a number of common concurrency patterns.

2.1. Syntactic and Stylistic Conventions

In this section we suggest some syntactic conventions for writing Orc programs. None of these conventions are required by the parser; newlines are used only to disambiguate certain corner cases in parsing, and other whitespace is ignored. However, following programming convention helps to improve the readability of programs, so that the programmer's intent is more readily apparent.

2.1.1. Parallel combinator

When the combined expressions are small, write them all on one line.

```
F | G | H
```

Note that we do not need parentheses here, since **|** is fully associative and commutative.

When the combined expressions are large enough to take up a full line, write one expression per line, with each subsequent expression aligned with the first and preceded by **|**. Indent the first expression to improve readability.

```
  long expression
| long expression
| long expression
```

A sequence of parallel expressions often form the left hand side of a sequential combinator. Since the sequential combinator has higher precedence, use parentheses to group the combined parallel expressions together.

```
( expression
| expression
) >x>
another expression
```

2.1.2. Sequential combinator

When the combined expressions are small, write a cascade of sequential combinators all on the same line.

```
F >x> G >y> H
```

Remember that sequential is right associative; in this example, x is bound in both G and H, and y is bound in H.

When the combined expressions are large enough to take up a full line, write one expression per line; each line ends with the combinator which binds the publications produced by that line.

```
long expression >x>
long expression >y>
long expression
```

For very long-running expressions, or expressions that span multiple lines, write the combinators on separate lines, indented, between each expression.

```
very long expression
  >x>
very long expression
  >y>
very long expression
```

2.1.3. Pruning combinator

When the combined expressions are small, write them on the same line:

```
F <x< G
```

When multiple pruning combinators are used to bind multiple variables (especially when the scoped expression is long), start each line with a combinator, aligned and indented, and continue with the expression.

```
long expression
  <x< G
  <y< H
```

The pruning combinator is not often written in its explicit form in Orc programs. Instead, the **`val`** declaration is often more convenient, since it is semantically equivalent and mentions the variable `x` before its use in scope, rather than after.

```
val x = G
val y = H
long expression
```

Additionally, when the variable is used in only one place, and the expression is small, it is often easier to use a nested expression. For example,

```
val x = G
val y = H
M(x,y)
```

is equivalent to

```
M(G,H)
```

Sometimes, we use the pruning combinator simply for its capability to terminate expressions and get a single publication; binding a variable is irrelevant. This is a special case of nested expressions. We use the identity site `let` to put the expression in the context of a function call.

For example,

```
x <x< F | G | H
```

is equivalent to

```
let(F | G | H)
```

The translation uses a pruning combinator, but we don't need to write the combinator, name an irrelevant variable, or worry about precedence (since the expression is enclosed in parentheses as part of the call).

2.1.4. Declarations

Declarations should always end with a newline.

```
def add(x,y) = x + y
val z = 7
add(z,z)
```

While the parser does not require a newline to end a declaration, it uses the newline to disambiguate certain corner cases in parsing, such as function application.

When the body of a declaration spans multiple lines, start the body on a new line after the = symbol, and indent the entire body.

```
def f(x,y) =
    declaration
    declaration
    body expression
```

Apply this style recursively; if a def appears within a def, indent its contents even further.

```
def f(x,y) =
    declaration
    def helper(z) =
     declaration in helper
     declaration in helper
     body of helper
    declaration
    body expression
```

2.2. Programming Idioms

In this section we look at some common idioms used in the design of Orc programs. Many of these idioms will be familiar to programmers using concurrency, and they are very simple to express in Orc.

2.2.1. Fork-join

One of the most common concurrent idioms is a *fork-join*: run two processes concurrently, and wait for a result from each one. This is very easy to express in Orc. Whenever we write a **`val`** declaration, the process

computing that value runs in parallel with the rest of the program. So if we write two **`val`** declarations, and then form a tuple of their results, this performs a fork-join.

```
val x = F
val y = G
(x,y)
```

Fork-joins are a fundamental part of all Orc programs, since they are created by all nested expression translations. In fact, the fork-join we wrote above could be expressed even more simply as just:

```
(F,G)
```

2.2.1.1. Example: Machine initialization

In Orc programs, we often use fork-join and recursion together to dispatch many tasks in parallel and wait for all of them to complete. Suppose that given a machine `m`, calling `m.init()` initializes `m` and then publishes a signal when initialization is complete. The function `initAll` initializes a list of machines.

```
def initAll([]) = signal
def initAll(m:ms) = ( m.init() , initAll(ms) ) >> signal
```

For each machine, we fork-join the initialization of that machine (`m.init()`) with the initialization of the remaining machines (`initAll(ms)`). Thus, all of the initializations proceed in parallel, and the function returns a signal only when every machine in the list has completed its initialization.

Note that if some machine fails to initialize, and does not return a signal, then the initialization procedure will never complete.

2.2.1.2. Example: Simple parallel auction

We can also use a recursive fork-join to obtain a value, rather than just signaling completion. Suppose we have a list of bidders in a sealed-bid, single-round auction. Calling `b.ask()` requests a bid from the bidder `b`. We want to ask for one bid from each bidder, and then return the highest bid. The function `auction` performs such an auction for a list of bidders (`max` finds the maximum of its arguments):

```
def auction([]) = 0
def auction(b:bs) = max(b.ask(), auction(bs))
```

2.2.1.3. Example: Barrier synchronization

Suppose we have an expression of the following form, where F and G are expressions and M and N are sites:

```
M() >x> F | N() >y> G
```

However, we would also like to *synchronize* the completion of M and N, so that neither F nor G starts executing until both M and N have published. This is a particular example of a fork-join:

```
( M() , N() ) >(x,y)> ( F | G )
```

We assume that x does not occur free in G, nor y in F.

2.2.2. Parallel Or

Next we consider a classic example of parallel programming which builds up from fork-join: a parallel-or program. Given two expressions F and G which may publish boolean values (or might stay silent forever), we want to find the disjunction of their results as soon as possible. If one of the expressions publishes `true`, then the disjunction is `true`; hence, it is not necessary to wait for the other expression to publish a value.

Here is the code:

```
val r =
    val a = F
    val b = G
      if(a) >> true
    | if(b) >> true
    | (a || b)
r
```

Recall that the `||` operator is strict; if one of `a` or `b` is not bound, it cannot compute a result. So, we have added **`if`**`(a) >> true` and **`if`**`(b) >> true`, which wait in parallel for either variable to become `true` and then publish the result `true`. That way, the parallel-or can evaluate to `true` based on only one of the results, even if the other result is not forthcoming.

Also note that the entire computation is within **`val`**` r =`, to prevent `true` from being published multiple times.

2.2.3. Finite Sequential Composition

F >x> G instantiates a copy of G for each published value of F. Suppose we know that **F** will publish only a finite number of values, and then complete; for example, **F** publishes the contents of a text file, one line at a time, and G prints each line to the console. After all of the lines have been printed, we want to start executing a new expression **H**.

Sequential composition alone is not sufficient, because we have no way to detect when all of the lines have been published. Fork-join is also not suitable, since the values to be printed are not stored in some traversable data structure like a list; instead, they are streaming as publications out of an expression. Instead, we use the **`;`** combinator, which waits for all of the lines to be printed, then the left side completes and the right side can run. Note that we must suppress the publications on the left side using **`stop`**; if the left side published, the right side would not run.

```
F >x> println(x) >> stop ; H
```

2.2.4. Timeout

One of the most powerful idioms available in Orc is a *timeout*: execute an expression for at most a specified amount of time. We accomplish this using **`val`** and a timer. The following program runs F for at most one second to get a value from it. If it does not publish within one second, the value defaults to 0.

```
let( F | Rtimer(1000) >> 0 )
```

2.2.4.1. Auction with timeout

In the auction example in Section 2.2.1.2, the auction may never complete if one of the bidders does not respond. We can add a timeout so that a bidder has at most 8 seconds to provide a bid:

```
def auction([]) = 0
def auction(b:bs) =
  val bid = b.ask() | Rtimer(8000) >> 0
  Max(bid, auction(bs))
```

Now, the auction is guaranteed to complete in at most 8 seconds.

2.2.4.2. Detecting timeout

Sometimes, rather than just yielding a default value, we would like to determine whether an expression has timed out, and if so, perform some other computation. To detect the timeout, we pair the original expression with `true` and the timer with `false`, so if the expression does time out, then we can detect it using the truth value. We then test the truth value, performing the usual computation if no timeout occurred, or some error-correcting computation if the expression did time out.

Here, we run the expression F with a time limit t. If it publishes within the time limit, we execute G (the result of F is bound to r). Otherwise, we execute H.

```
val (r,b) = (F,true) | (Rtimer(t),false)
if b then G else H
```

Or, instead of using a boolean, we could use pattern matching:

```
val s = some(F) | Rtimer(t) >> none()
  s >some(r)> G
| s >none()>  H
```

2.2.5. Priority

We can also use a timer to give a window of priority to one computation over another. In this example, we run expressions F and G concurrently. F has priority for the first second: if F publishes, its value is used immediately, but if G publishes, that publication is suspended for the first second and may be superseded by a publication from F during that time. If neither F nor G publishes within a second, then whichever publishes first after that point is the winner.

```
val x = F
val y = G
let( y | Rtimer(1000) >> x )
```

2.2.6. Metronome

Recall the definition of `metronome` from the previous chapter:

```
def metronome(t) = signal | Rtimer(t) >> metronome()
```

We can use `metronome` and the sequential combinator to repeatedly start an expression, so that copies of that expression will execute at regular intervals.

```
{-
   Publish "tick" once every second.
   Publish "tock" once every second, with an initial 500ms delay.
   The publications alternate: tick tock tick tock ...
-}
  metronome(1000) >> "tick"
| Rtimer(500) >> metronome(1000) >> "tock"
```

2.2.7. Priority Poll

Suppose we have a list of buffers. We would like to poll these buffers to see if they have any available data. The list of buffers is ordered by priority; the first buffer in the list has the highest priority, so it should be polled first, and if it has no data, then the next buffer should be polled, and so on.

Here is a function which polls a prioritized list of buffers in this way. It publishes the first item that it finds, removing it from the originating buffer. If all buffers are empty, it remains silent. We use the `getnb` ("get non-blocking") method of the buffer, which retrieves the first available item if there is one, or else remains silent and completes immediately if the buffer is empty (it does not wait for an item to become available).

```
def PriorityPoll([]) = stop
def PriorityPoll(b:bs) = b.getnb() ; PriorityPoll(bs)
```

2.2.8. Lists

In the section on Cor, we were introduced to lists: how to construct them, and how to match them against patterns. While it is certainly feasible to write a specific function with an appropriate pattern match every time we want to access a list, it is helpful to have a handful of common operations on lists and reuse them.

One of the most common uses for a list is to send each of its elements through a sequential combinator. Since the list itself is a single value, we want to walk through the list and publish each one of its elements in parallel as a value. The library function `each` does exactly that.

Suppose we want to send the message `invite` to each email address in the list `inviteList`:

```
each(inviteList) >address> Email(address, invite)
```

Orc also adopts many of the list idioms of functional programming. The Orc library contains definitions for most of the standard list functions, such as `map` and `fold`. Many of the list functions internally take advantage of concurrency to make use of any available parallelism; for example, the `map` function dispatches all of the mapped calls concurrently, and assembles the result list once they all return using a fork-join.

2.2.9. Streams

Sometimes a source of data is not explicitly represented by a list or other data structure. Instead, it is made available through a site, which returns the values one at a time, each time it is called. We call such a site a

stream. It is analogous to an iterator in a language like Java. Functions can also be used as streams, though typically they will not be pure functions, and should only return one value. A call to a stream may halt, to indicate that the end of the data has been reached, and no more values will become available. It is often useful to detect the end of a stream using the otherwise combinator.

Streams are common enough in Orc programming that there is a library function to take all of the available publications from a stream; it is called `repeat`, and it is analogous to `each` for lists.

```
def repeat(f) = f() >x> (x | repeat(f))
```

The `repeat` function calls the site or function `f` with no arguments, publishes its return value, and recurses to query for more values. `repeat` should be used with sites or functions that block until a value is available. Notice that if any call to `f` halts, then `repeat(f)` consequently halts.

For example, it is very easy to treat a channel `c` as a stream, reading any values put on the channel as they become available:

```
repeat(c.get)
```

2.2.10. Arrays

While lists are a very useful data structure, they are not indexed; it is not possible to get the nth element of a list in constant time. However, this capability is often needed in practice, so the Orc standard library provides a function `IArray` to create immutable arrays. Once initialized, an immutable array cannot be rewritten; it can only be read.

`IArray` takes two arguments: an array size, and a function to initialize the array. The function takes the index being initialized as an argument (indices start at 0), and publishes the value to be stored at that array position. Here are a few examples:

```
{- Create an array of 10 elements; element i is the ith power of 2 -}
IArray(10, lambda(i) = 2 ** i)
```

```
{- Create an array of 5 elements; each element is a newly created buffer -}
IArray(5, lambda(_) = Buffer())
```

The array is used like a function; the call `A(i)` returns the `i`th element of the array `A`. A call with an out-of-bounds index halts.

```
{- Create an array of 3 buffers -}
val A = IArray(10, lambda(_) = Buffer())

{- Send true on the 0th channel, and listen for a value on the 0th channel. -}
A(0).put(true) | A(0).get()
```

Since arrays are accessed by index, there is a library function specifically designed to make programming with indices easier. The function `upto(n)` publishes all of the numbers from `0` to `n-1` simultaneously; thus, it is very easy to access all of the elements of an array simultaneously. Suppose we have an array `A` of `n` email addresses and would like to send the message `m` to each one.

```
upto(n) >i> A(i) >address> Email(address, m)
```

2.2.11. Loops

Orc does not have any explicit looping constructs. Most of the time, where a loop might be used in other languages, Orc programs use one of two strategies:

1. When the iterations of the loops can occur in parallel, write an expression that expands the data into a sequence of publications, and use a sequential operator to do something for each publication. This is the strategy that uses functions like `each`, `repeat`, and `upto`.

2. When the iterations of the loops must occur in sequence, write a tail recursive function that iterates over the data. Any loop can be rewritten as a tail recursion. Typically the data of interest is in a list, so one of the standard list functions, such as `foldl`, applies. The library also defines a function `while`, which handles many of the common use cases of while loops.

2.2.12. Mutable Storage

Variables in Orc are immutable. There is no assignment operator, and there is no way to change the value of a bound variable. However, it is often useful to have mutable state when writing certain algorithms. The Orc library contains two sites that offer simple mutable storage: `Ref` and `Cell`.

The `Ref` site creates rewritable reference cells.

```
val r = Ref(0)
println(r.read()) >>
r.write(2) >>
println(r.read()) >>
stop
```

These are very similar to ML's `ref` cells. `r.write(v)` stores the value `v` in the reference `r`, overwriting any previous value, and publishes a signal. `r.read()` publishes the current value stored in `r`.

However, unlike in ML, a reference cell can be left initially empty by calling `Ref` with no arguments. A read operation on an empty cell blocks until the cell is written.

```
{- Create a cell, and wait 1 second before initializing it. -}
{- The read operation blocks until the write occurs. -}
val r = Ref()
r.read() | Rtimer(1000) >> r.write(1) >> stop
```

The Orc library also offers write-once reference cells, using the `Cell` site. A write-once cell has no initial value. Read operations block until the cell has been written. A write operation succeeds only if the cell is empty; subsequent write operations simply halt.

```
{- Create a cell, try to write to it twice, and read it -}
{- The read will block until a write occurs, and only one write will succeed. -}
val c = Cell()
  Rtimer(1000) >> r.write(2) >> println("Wrote 2") >> stop
| Rtimer(1000) >> r.write(3) >> println("Wrote 3") >> stop
```

```
| r.read()
```

Write-once cells are very useful for concurrent programming, and they are often safer than rewritable reference cells, since the value cannot be changed once it has been written. The use of write-once cells for concurrent programming is not a new idea; they have been studied extensively in the context of the Oz programming language [http://en.wikipedia.org/wiki/Oz_programming_language].

A word of caution: References, cells, and other mutable objects may be accessed concurrently by many different parts of an Orc program, so race conditions may arise.

2.2.13. Parallel Matching

Matching a value against multiple patterns, as we have seen it so far, is a linear process, and requires a **`def`** whose clauses have patterns in their argument lists. Such a match is linear; each pattern is tried in order until one succeeds.

What if we want to match a value against multiple patterns in parallel, executing every clause that succeeds? Fortunately, this is very easy to do in Orc. Suppose we have an expression F which publishes pairs of integers, and we want to publish a signal for each 3 that occurs.

We could write:

```
F >(x,y)>
  ( if(x=3) >> signal
  | if(y=3) >> signal )
```

But there is a more general alternative:

```
F >x>
  ( x >(3,_)> signal
  | x >(_,3)> signal )
```

The interesting case is the pair `(3,3)`, which is counted twice because both patterns match it in parallel.

This is sometimes used as an alternative to pattern matching using function clauses, but only when the patterns are mutually exclusive.

For example,

```
def helper([]) = 0
def helper([_]) = 1
def helper(_:_:_) = 2
helper([4,6])
```

is equivalent to

```
[4,6] >x>
  x >[]> 0
| x >[_]> 1
```

```
| x >_:_:_> 2
```

whereas

```
def helper([]) = 0
def helper([_]) = 1
def helper(_) = 2
helper([5])
```

is *not* equivalent to

```
[5] >x>
  x >[]> 0
| x >[_]> 1
| x >_> 2
```

because the clauses are not mutually exclusive. Function clauses must attempt to match in linear order, whereas this expression matches all of the patterns in parallel. Here, it will match `[5]` two different ways, publishing both `1` and `2`.

2.2.14. Routing

While Orc's combinators are powerful, sometimes they are too restrictive to solve certain problems. In these situations, we use channels or other concurrent objects to redirect or store publications and write clean solutions. We call this technique *routing*.

2.2.14.1. Generalizing Termination

The pruning combinator terminates an expression after it publishes its first value. We have already seen how to use pruning just for its termination capability, without binding a variable, using the `let` site. Now, we use routing to terminate an expression under different conditions, not just when it publishes a value; it may publish many values, or none, before being terminated.

Our implementation strategy is to route the publications of the expression through a channel, so that we can put the expression inside a pruning combinator and still see its publications without those publications terminating the expression.

2.2.14.1.1. Enhanced Timeout

As a simple demonstration of this concept, we construct a more powerful form of timeout: allow an expression to execute, publishing arbitrarily many values (not just one), until a time limit is reached.

```
val c = Buffer()
repeat(c.get) <<
    F >x> c.put(x) >> stop
  | Rtimer(1000)
```

This program allows F to execute for one second, and then terminates it. Each value published by F is silently routed through the channel `c` so that it does not terminate F. After one second, the signal published by the timeout terminates F.

2.2.14.1.2. Test Pruning

We can also decide to terminate based on the values published. This expression executes F until it publishes a negative number, and then terminates it:

```
val c = Buffer()
repeat(c.get) <<
  F >x>
    (if x >= 0
        then c.put(x) >> stop
        else c.close() >> signal)
```

Each value published by F is tested. If it is non-negative, it is placed on channel `c` (silently) and read by `repeat(c.get)`. If it is negative, the channel is closed and a signal is published, causing the termination of F.

2.2.14.1.3. Exceptions

Orc does not have a native exception handling mechanism. However, we can use routing to directly interrupt an expression by writing to a write-once cell. We set up the expression like a timeout, but instead of waiting for a timer, we wait for the cell `err` to be assigned. Anywhere in F, an `err.write` call will terminate the expression (because it will cause `err.read()` to publish), but otherwise F executes as normal and may publish any number of values.

```
val c = Buffer()
val err = Cell()
repeat(c.get) <<
    F >x> c.put(x) >> stop
  | err.read()
```

2.2.14.1.4. Publication Limit

We can limit an expression to *n* publications, rather than just one. Here is an expression which executes F until it publishes 5 values, and then terminates it.

```
val c = Buffer()
val done = Cell()
def allow(0) = c.close() >> done.write(signal) >> stop
def allow(n) = c.get() | allow(n-1)
allow(5) <<
    F >x> c.put(x) >> stop
  | done.read()
```

2.2.14.2. Modified Otherwise

We can use routing to create a modified version of the otherwise combinator. We'll run F until it completes, and then run G, regardless of whether F published any values or not. We write this modified F `;` G as follows:

```
val c = Buffer()
repeat(c.get) | (F >x> c.put(x) >> stop ; c.close() >> G)
```

2.2.14.3. Modified Pruning

Similarly, we can use routing to modify the pruning combinator. Instead of terminating the right side after the first value is published, we would like to continue running it, ignoring its remaining publications. We write this modified F <x< G as follows:

```
val c = Cell()
(F <x< c.read()) | (G >x> c.write(x))
```

2.2.15. Interruptible

Let's write a function `interruptible` that executes functions in an interruptible way. `interruptible(g)` runs `g`, which is assumed to take no arguments, and silences its publications. It immediately publishes another function, which we can call at any time to terminate the execution of `g`.

Here is a naive implementation that doesn't quite work:

```
def interrputible(f) =
  val c = new Cell()
  (lambda () = c.write(signal))
    << (f() >> stop | c.read())

{- wrong! -}
val stopper = interruptible(g)
...stopper()...
```

The function `interruptible` is correct, but the way it is used causes a strange error. The function `g` executes, but is always immediately terminated! This happens because the **`val`** declaration which binds `stopper` also kills all of the remaining computation in `interruptiuble(g)`, including the execution of `g` itself.

The solution is to bind the variable differently:

```
def interrputible(f) =
  val c = new Cell()
  (lambda () = c.write(signal))
    << (f() >> stop | c.read())

interruptible(g) >stopper>
...
```

This idiom, where a function publishes some value that can be used to monitor or control its execution, arises occasionally in Orc programming. When using this idiom, always remember to avoid terminating that execution accidentally. Since Orc is a structured concurrent language, every process is contained with some other process; kill the containing process, and the contained processes die too.

2.3. Larger Examples

2.3.1. Dining Philosophers

The dining philosophers problem is a well known and intensely studied problem in concurrent programming. For a detailed description, see the Wikipedia entry on the problem. Here, we present an Orc solution to the dining philosophers problem, based on a probabilistic solution by Rabin.

```
def shuffle(a,b) = if (random(2) = 1) then (a,b) else (b,a)

def take((a,b)) =
  a.acquire() >> b.acquirenb() ;
  a.release() >> take(shuffle(a,b))

def drop(a,b) = (a.release(), b.release()) >> signal

def phil(a,b,name) =
  def thinking() =
    if (urandom() < 0.9)
      then Rtimer(random(1000))
      else println(name + " is thinking forever.") >> stop
  def hungry() = take((a,b))
  def eating() =
    println(name + " is eating.") >>
    Rtimer(random(1000)) >>
    println(name + " has finished eating.") >>
    drop(a,b)
  thinking() >> hungry() >> eating() >> phil(a,b,name)

def dining(n) =
  if (n < 2)
    then println("Can't simulate fewer than two philosophers.")
    else
    (
      val forks = IArray(n, lambda(_) = Semaphore(1))
      def phils(0) = stop
      def phils(i) = phil(forks(i%n), forks(i-1), "Philosopher " + i)
                   | phils(i-1)
      phils(n)
    )
dining(5) ; println("Done.") >> stop
```

The program calls `dining(5)` to simulate the dining philosophers problem with size 5. It waits for the simulation to complete, then prints a message and stops.

The `dining` function takes the number `n` of philosophers to simulate. It creates an array of `n` binary semaphores to represent the forks. Then, it starts `n` philosopher processes in parallel using the function `phil`, giving each philosopher an identifier, and references to its left and right forks.

The `phil` function describes the behavior of an individual philosopher. It calls the `thinking`, `hungry`, and `eating` functions in a continuous loop. A `thinking` philosopher waits for a random amount of time, and also has a 10% chance of thinking forever. A `hungry` philosopher uses the `take` function

to acquire two forks. An `eating` philosopher waits for a random time interval and then uses the `drop` function to yield ownership of its forks.

The `take` and `drop` functions are the key to the algorithm. Calling `take` attempts to acquire a pair of forks in two steps: wait for one fork to become available, then immediately attempt to acquire the second fork. If the second fork is acquired, signal success; otherwise, release the first fork, and then try again, randomly changing the order in which the forks are acquired using the `shuffle` helper function. This reordering ensures that the algorithm will probabilistically avoid livelock. The `drop` function simply releases both of the forks.

2.3.2. Hygienic Dining Philosophers

Here we implement a different solution to the Dining Philosophers problem, described in "The Drinking Philosophers Problem", by K. M. Chandy and J. Misra. Briefly, this algorithm efficiently and fairly solves the dining philosophers problem for philosophers connected in an arbitrary graph (as opposed to a simple ring). The algorithm works by augmenting each fork with a clean/dirty state. Initially, all forks are dirty. A philosopher is only obliged to relinquish a fork to its neighbor if the fork is dirty. On receiving a fork, the philosopher cleans it. On eating, the philosopher dirties all forks. For full details of the algorithm, consult the original paper.

```
{-
Start a philosopher actor; never publishes.
Messages sent between philosophers include:
- ("fork", p): philosopher p relinquishes the fork
- ("request", p): philosopher p requests the fork
- ("rumble", p): sent by a philosopher to itself when it should
  become hungry

name: identify this process in status messages
mbox: our mailbox; the "address" of this philosopher is mbox.put
missing: set of neighboring philosophers holding our forks
-}
def philosopher(name, mbox, missing) =
  {- deferred requests for forks -}
  val deferred = Buffer()
  {- forks we hold which are clean -}
  val clean = Set()

  def sendFork(p) =
    {- remember that we no longer hold the fork -}
    missing.add(p) >>
    p(("fork", mbox.put))

  def requestFork(p) =
    p(("request", mbox.put))

  {- Start a timer which will tell us when we're hungry. -}
  def digesting() =
      println(name + " thinking") >>
      thinking()
    | Rtimer(random(1000)) >>
      mbox.put(("rumble", mbox.put)) >>
      stop
```

```
  {- Wait to become hungry -}
  def thinking() =
    def on(("rumble", _)) =
      println(name + " hungry") >>
      map(requestFork, missing) >>
      hungry()
    def on(("request", p)) =
      sendFork(p) >> thinking()
    on(mbox.get())

  {- Eat once we receive all forks -}
  def hungry() =
    def on(("fork", p)) =
      missing.remove(p) >>
      clean.add(p) >>
      if missing.isEmpty()
      then println(name + " eating") >> eating()
      else hungry()
    def on(("request", p)) =
      if clean.contains(p)
      then deferred.put(p) >> hungry()
      else sendFork(p) >> requestFork(p) >> hungry()
    on(mbox.get())

  {- Dirty forks, process deferred requests, then digest -}
  def eating() =
    clean.clear() >>
    Rtimer(random(1000)) >>
    map(sendFork, deferred.getAll()) >>
    digesting()

  {- All philosophers start out digesting -}
  digesting()

{-
Create an NxN 4-connected grid of philosophers.  Each philosopher
holds the fork for the connections below and to the right (so the
top left philosopher holds both its forks).
-}
def philosophers(n) =
  {- A set with 1 item -}
  def Set1(item) = Set() >s> s.add(item) >> s
  {- A set with 2 items -}
  def Set2(i1, i2) = Set() >s> s.add(i1) >> s.add(i2) >> s

  {- create an NxN matrix of mailboxes -}
  val cs = uncurry(IArray(n, lambda (_) = IArray(n, ignore(Buffer))))

  {- create the first row of philosophers -}
  philosopher((0,0), cs(0,0), Set())
  | for(1, n) >j>
    philosopher((0,j), cs(0,j), Set1(cs(0,j-1).put))
```

```
  {- create remaining rows -}
  | for(1, n) >i> (
      philosopher((i,0), cs(i,0), Set1(cs(i-1,0).put))
      | for(1, n) >j>
        philosopher((i,j), cs(i,j), Set2(cs(i-1,j).put, cs(i,j-1).put))
    )

{- Simulate a 3x3 grid of philosophers for 10 seconds -}
let(
  philosophers(3)
  | Rtimer(10000)
) >> "HALTED"
```

Our implementation is based on the actor model of concurrency. An actor is a state machine which reacts to messages. On receiving a message, an actor can send asynchronous messages to other actors, change its state, or create new actors. Each actor is single-threaded and processes messages sequentially, which makes some concurrent programs easier to reason about and avoids explicit locking. Erlang is one popular language based on the actor model.

Orc emulates the actor model very naturally. In Orc, an actor is an Orc thread of execution, together with a `Buffer` which serves as a mailbox. To send a message to an actor, you place it in the actor's mailbox, and to receive a message, the actor gets the next item from the mailbox. The internal states of the actor are represented by functions: while an actor's thread of execution is evaluating a function, it is considered to be in the corresponding state. Because Orc implements tail-call optimization, state transitions can be encoded as function calls without running out of stack space.

In this program, a philosopher is implemented by an actor with three primary states: `eating`, `thinking`, and `hungry`. An additional transient state, `digesting`, is used to start a timer which will trigger the state change from `thinking` to `hungry`. Each state is implemented by a function which reads a message from the mailbox, selects the appropriate action using pattern matching, performs the action, and finally transitions to the next state (possibly the same as the current state) by calling the corresponding function.

Forks are never represented explicitly. Instead each philosopher identifies a fork with the "address" (sending end of a mailbox) of the neighbor who shares the fork. Every message sent includes the sender's address. Therefore when a philosopher receives a request for a fork, it knows who requested it and therefore which fork to relinquish. Likewise when a philosopher receives a fork, it knows who sent it and therefore which fork was received.

2.3.3. Readers/Writers

Here we present an Orc solution to the readers-writers problem. Briefly, the readers-writers problem involves concurrent access to a mutable resource. Multiple readers can access the resource concurrently, but writers must have exclusive access. When readers and writers conflict, different solutions may resolve the conflict in favor of one or the other, or fairly. In the following solution, when a writer tries to acquire the lock, current readers are allowed to finish but new readers are postponed until after the writer finishes. Lock requests are granted in the order received, guaranteeing fairness. Normally, such a service would be provided to Orc programs by a site, but it is educational to see how it can be implemented directly in Orc.

```
{-
Create a reader/writer lock and return the functions
used to acquire and release the lock.
-}
def ReadWriteLock() =
  {- Queue of lock requests -}
```

```
  val m = Buffer()
  {- Count of active readers/writers -}
  val c = Counter()

  {- Grant read request -}
  def grant((false,s)) = c.inc() >> s.release()
  {- Grant write request -}
  def grant((true,s)) =
    c.onZero() >> grant((false,s)) >> c.onZero()

  {- Boolean argument is true if writing -}
  def acquire(write) =
    val s = Semaphore(0)
    m.put((write, s)) >> s.acquire()

  def release() = c.dec()

  {- Process requests in sequence -}
  repeat(lambda () = m.get() >r> grant(r)) >> stop
  {- Return the lock functions -}
  | (acquire, release)

{- Test the lock -}
let(
  ReadWriteLock() >(acquire, release)> (
    def test(write) =
      val label = if write then "WRITE" else "READ"
      acquire(write) >> println("START "+label) >>
      Rtimer(1000) >> println("END "+label) >>
      release()

    (                             {- Output:     -}
      Rtimer(10) >> test(false)   {- START READ  -}
    | Rtimer(20) >> test(false)   {- START READ  -}
                                  {- END READ    -}
                                  {- END READ    -}
    | Rtimer(30) >> test(true)    {- START WRITE -}
                                  {- END WRITE   -}
    | Rtimer(40) >> test(false)   {- START READ  -}
    | Rtimer(50) >> test(false)   {- START READ  -}
                                  {- END READ    -}
                                  {- END READ    -}
    ) >> stop
    ; signal
  )
) >> stop
```

The lock receives requests over the channel `m` and processes them sequentially with the function `grant`. Each request includes a boolean flag which is true for write requests and false for read requests, and a `Semaphore` which the requester blocks on. The lock grants access by releasing the semaphore, unblocking the requester.

The counter `c` tracks the number of readers or writers currently holding the lock. Whenever the lock is granted, `grant` increments `c`, and when the lock is released, `c` is decremented. To ensure that a writer

has exclusive access, `grant` waits for the `c` to become zero before granting the lock to the writer, and then waits for `c` to become zero again before granting any more requests.

Chapter 3. Accessing and Creating External Services

3.1. Java Integration

Java classes can be imported into Orc as sites using the `class` declaration. Imported classes must be in the classpath of the JVM running the Orc interpreter. The following sections describe in detail how such imported classes behave in Orc programs.

3.1.1. Dot Operator

`x.member`, where `x` evaluates to a Java class or object, is evaluated as follows:

- If `x` has one or more methods named `member`, a "method handle" site is returned which may be called like any other Orc site. When a method handle is actually called with arguments, the appropriate Java method is selected and called depending on the number and type of arguments, as described in Method Resolution below.
- Otherwise, if `x` has a field named `member`, the object's field is returned, encapsulated in a `Ref` object [56]. The `Ref` object has `read` and `write` methods which are used to get and set the value of the field.

Note that no distinction is made between static and non-static members; it is an error to reference a non-static member through a class, but this does not change how members are resolved. Note also that if a field shares a name with one or more methods, there is no way to access the field directly.

The following (rather useless) example illustrates how the dot operator can be used to access both static and non-static methods and fields:

```
{- bind Integer to a Java class -}
class Integer = java.lang.Integer

{- call a static method -}
val i = Integer.decode("5")
{- read a field -}
val m = Integer.MIN_VALUE.read()
{- write a field -}
Integer.MIN_VALUE.write(5) >>
{- call a non-static method -}
i.toString()
```

3.1.2. Direct Calls

When `x` evaluates to a Java object (but not a Java class), the syntax `x(...)` is equivalent to `x.apply(...)`.

When `x` evaluates to a Java class, the syntax `x(...)` calls the class's constructor. In case of overloaded constructors, the appropriate constructor is chosen based on the number and types of arguments as described in Method Resolution.

3.1.3. Method Resolution

When a method handle is called, the actual Java method called is chosen based on the runtime types of the arguments, as follows:

1. If only one method has the appropriate number of arguments, that method is called.
2. Otherwise, each method taking the appropriate number of arguments is tested for type compatibility as follows, and the first matching method is called.
    a. Every argument is compared to the corresponding formal parameter type as follows. All arguments must match for the method to match.
        i. If the argument is null, then the argument matches
        ii. If the formal parameter type is primitive (int, char, float, ...) and the argument is an instance of a wrapper class, then the argument is unboxed (unwrapped) and coerced to the type of the formal parameter according to Java's standard rules for implicit widening coercions.
        iii. If the formal parameter type is a primitive numeric type and the argument is an instance of `BigDecimal`, the argument is implicitly narrowed to the formal parameter type.
        iv. If the formal parameter type is a primitive integral type and the argument is an instance of `BigInteger`, the argument is implicitly narrowed to the formal parameter type.
        v. Otherwise, the argument must be a subtype of the formal parameter type.

The reason for the unusual implicit narrowing of `BigDecimal` and `BigInteger` is that Orc numeric literals have these types, and it would be awkward to have to perform an explicit conversion every time such a value is passed to a Java method expecting a primitive.

Currently we do not implement specificity rules for choosing the best matching method; the first matching method (according to some unspecified order) is chosen. Note also that we do not support varargs methods explicitly, but instead varargs may be passed as an array of the appropriate type.

3.1.4. Orc Values in Java

Any value in an Orc program may be treated as a Java object and passed to Java methods. Orc literals have the following Java types:

string	`java.lang.String`
boolean	`java.lang.Boolean`
integer	`java.math.BigInteger`
float	`java.math.BigDecimal`
tuple	`orc.runtime.values.TupleValue`
function	`orc.runtime.values.Closure`
list	`orc.runtime.values.ListValue`

3.1.5. Java Values in Orc

Java objects may be used directly as values anywhere in an Orc program. Primitive Java values cannot be used directly in an Orc program, but are automatically boxed (and unboxed) as necessary.

When both arguments of an arithmetic or comparison operator are Java numeric types, the arguments are implicitly coerced to the widest of the two argument types. "Widest" is defined by the following relation, where ">" means "is wider than": BigDecimal > Double > Float > BigInteger > Long > Integer > Short > Byte

3.1.6. Cooperative Scheduling

3.1.6.1. Overview

In order to support massive concurrency efficiently in Java, Orc uses cooperative threading. Orc programs are broken into discrete steps which are executed by a fixed-size thread pool. This approach works for Orc expressions, but the internals of an Orc site written in Java cannot be easily broken down. So Orc has two choices when it needs to call a local Java site:

- Run the site call in a new thread. This means site calls never unnecessarily block the Orc engine, but it also limits the number of concurrent site calls an Orc program can make.
- Run the site call in the same thread as the Orc engine, which conserves thread resources but may block the engine from making further progress until the site call finishes.

Fortunately, in practice most Orc site calls take a short, deterministic amount of time to complete. Of those that don't, it's usually only because they are blocked waiting for some external event, like another site call. In this case, instead of blocking the site can return control to the Orc engine, asking to be run again when the external event occurs. In typical Java applications this kind of non-blocking behavior is implemented using callbacks (or in its most general form, continuation passing). Unfortunately this creates convoluted and verbose code: what if you need to block in the middle of a **`for`** loop, for example?

Kilim [http://kilim.malhar.net/] resolves this issue by allowing programmers to write code in a natural blocking style and then manipulating the bytecode to perform a form of CPS conversion, so that instead of blocking the code actually returns control to a scheduler. Orc incorporates Kilim to allow site authors to write sites with internal concurrency and blocking behavior which don't use Java threads and cooperate with the Orc engine.

3.1.6.2. Kilim Tutorial

For a full introduction to Kilim, see A Thread of One's Own [http://www.malhar.net/sriram/kilim/thread_of_ones_own.pdf], by Sriram Srinivasan. In the following, we will cover just enough to get you started writing Orc sites using Kilim.

Kilim introduces three key primitives:

`Pausable` This checked exception marks methods which may block [1]. You should never catch this exception, and you should always declare it even if you have already declared **`throws`** `Exception` or **`throws`** `Throwable`. In all other respects it follows the normal rules for checked exceptions: an override can only throw it if the superclass method throws it, and you must throw it if you call any method which throws it.

`Mailbox` A multiple-producer, single-consumer blocking queue used to communicate with tasks.

`Task` Cooperative analogue of a Thread. Creating and running a task looks like this:

```
new Task() {
    public void execute() throws Pausable {
        // do some stuff
```

```
            }
        }.start();
```

When writing Orc sites you rarely need to use tasks explicitly because every Orc site call is implicitly run inside a Kilim task. So all you need to do is mark `Pausable` methods. Here's a short example of a buffer site written using Kilim `Mailbox`es:

```
public class KilimBuffer<V> {
 private LinkedList<Mailbox<V>> waiters =
  new LinkedList<Mailbox<V>>();
 private LinkedList<V> buffer = new LinkedList<V>();
 public synchronized void put(V o) throws Pausable {
  Mailbox<V> waiter = waiters.poll();
  if (waiter != null) waiter.put(o);
  else buffer.add(o);
 }
 public synchronized V get() throws Pausable {
  V out = buffer.poll();
  if (out != null) return out;
  else {
   Mailbox<V> waiter = new Mailbox<V>();
   waiters.add(waiter);
   return waiter.get();
  }
 }
}
```

Classes using Kilim can be imported like other Java classes using the Orc **`class`** declaration. For example, we can use the `KilimBuffer` class defined above:

```
class Buffer = orc.lib.state.KilimBuffer
val b = Buffer()
  Rtimer(1000) >> b.put("1 second later") >> null
| b.get()
```

3.1.6.3. When you must block

Sometimes blocking is unavoidable, for example if a site must perform blocking IO. For such cases, Orc provides a utility method `orc.runtime.Kilim#runThreaded(Callable)` which farms work out to a thread pool. The advantage of doing this over spawning your own thread is that there is no chance of using too many threads; if no thread is available, the method pauses until one becomes available. The disadvantage is that if you have too many Java methods which must communicate with each other concurrently and can't use Mailboxes, there won't be enough threads for them all and execution will deadlock. This situation is sufficiently rare that `runThreaded` is usually the correct approach.

3.1.6.4. Compiling Kilim Sites

Code which uses Kilim annotations must be processed after compiling and before running, with the Kilim "weaver". For example, if your `.class` files are in `./build`, you would run the weaver like this:

```
java -cp orc-0.9.3.jar:lib/kilimex.jar:./build \
  kilim.tools.Weaver -d ./build ./build
```

The Orc source distribution includes an ant **kilim** task in `build.xml` to do exactly this.

Appendix A. Complete Syntax of Orc

Table A.1. Complete Syntax of Orc

E	::=	*Expression*	
	C		*constant value*
	| X		*variable*
	| **stop**		*silent expression*
	| (E , ... , E)		*tuple*
	| [E , ... , E]		*list*
	| E G+		*call*
	| E *op* E		*operator*
	| E **>**P**>** E		*sequential combinator*
	| E **|** E		*parallel combinator*
	| E **<**P**<** E		*pruning combinator*
	| E **;** E		*otherwise combinator*
	| **lambda** (P , ... , P) = E		*closure*
	| **if** E **then** E **else** E		*conditional*
	| D E		*scoped declaration*
G	::=	*Argument group*	
	(E , ... , E)		*arguments*
	| . *field*		*field access*
C	::=	*Boolean* | *Number* | *String* | **signal**	*Constant*
X	::=	*identifier*	*Variable*
D	::=	*Declaration*	
	val P = E		*value declaration*
	| **def** X(P , ... , P) = E		*function declaration*
	| **site** X = *address*		*site declaration*
	| **class** X = *classname*		*class declaration*
	| **include** " *filename* "		*inclusion*
P	::=	*Pattern*	
	X		*variable*
	| C		*constant*
	| _		*wildcard*
	| (P , ... , P)		*tuple*
	| [P , ... , P]		*list*
	| !P		*publish pattern*
	| P **as** X		*as pattern*

Where relevant, syntactic constructs are ordered by precedence, from highest to lowest. For example, among expressions, calls have higher precedence than any of the combinators, which in turn have higher precedence than conditionals.

Appendix B. Standard Library

B.1. Overview

The standard library is a set of declarations implicitly available to all Orc programs. In this section we give an informal description of the standard library, including the type of each declaration and a short explanation of its use.

Orc programs are expected to rely on the host language and environment for all but the most essential sites. For example, in the Java implementation of Orc, the entire Java standard library is available to Orc programs via **`class`** declarations. Therefore the Orc standard library aims only to provide convenience for the most common Orc idioms, not the complete set of features needed for general-purpose programming.

B.2. Notation

Each declaration in the standard library includes a type signature as part of its documentation. The notation for type signatures, summarized here, is based on a formal type system for Orc currently under preparation.

A type signature consists of: a declaration keyword, a declaration name, argument types, the return type, and finally any subtyping constraints. For example **`def`** `min(A, A) :: A, A <: Comparable` gives the type of the **`def`** declaration for `min`. According to this signature, `min` is a function which takes two arguments of type `A` and returns a value of the same type, where `A` is any subtype of `Comparable`.

Declaration keywords include **`site`**, **`def`**, **`val`**, and **`type`**. The first three keywords are described in previous chapters. The last, **`type`**, declares an algebraic data type with one or more constructors. The constructors may be used both as sites and as destructuring pattern forms. For example, the declaration **`type`** `Either<A> = Some(A)` **`|`** `None()` implicitly declares two sites: **`site`** `Some(A) :: Either<A>` and **`site`** `None() :: Either<A>`. It also implicitly declares two pattern forms `Some(_)` and `None()`: `Some(y)` **`>`**`Some(x)`**`>`** `x` publishes the value of `y`, and `Some(y)` **`>`**`None()`**`>`** **`signal`** halts (failing to match).

Object members (methods and fields) are each declared separately. The binding name of a member is written in the form Type.member, e.g. `Foo.get` refers to the `get` member of an object of type `Foo`. The object type can include type variables which are referenced by the member type, so for example **`site`** `Buffer<A>.get() :: A` means that when the `get` method is called on a `Buffer` holding an arbitrary element type `A`, it will return a value of the same type.

Argument and return types are written as follows:

- Primitive types are given descriptive names based on the names of the corresponding Java classes. For example, `Number` is any number, and `Comparable` is any value which supports a total order.
- Type variables in polymorphic declarations are written using the letters `A` ... `Z`. For example, the signature **`site`** `let(A) :: A` means that `let` takes one argument of any type, and returns a value of the same type.
- The Top type (of which all types are subtypes) is written `Top`. The Bottom type (which is a subtype of all types) is written `Bot`.
- A list with element type `A` is written `[A]`.
- A tuple with element types `A, B, ...` is written `(A, B, ...)`.

- Any other parameterized type is written as a type name followed by type parameters in angle brackets. For example, `Array<Integer>` is the type of arrays of integers.
- A function type (when given as an argument or return type) is written with the keyword **`lambda`** followed by the argument types and return type as in a declaration type signature. For example, **`lambda`** `(Integer) :: Integer` is the type of functions mapping integers to integers.

Multiple sets of argument types are syntactic sugar for currying. For example, **`site`** `Foo()() :: Signal` is equivalent to **`site`** `Foo() :: ` **`lambda`** `() :: Signal`.

Subtyping constraints are used to constrain the types of polymorphic type variables, and are written `A <: X`, meaning `A` must be a subtype of `X`.

B.3. Reference

B.3.1. 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(A) :: A`

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

`let` **`site`** `let(A, ...) :: (A, ...)`

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

`if` **`site`** **`if`**`(Boolean) :: Signal`

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.

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 bth 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 (<)(Comparable, Comparable) :: Boolean

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

site (<=)(Comparable, Comparable) :: Boolean

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

Comparable) ::
Boolean

site (>)
(Comparable,
Comparable) ::
Boolean

site (>)(Comparable, Comparable) :: Boolean

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

site (>=)
(Comparable,
Comparable) ::
Boolean

site (>=)(Comparable, Comparable) :: Boolean

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

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

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

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

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

site (~)
(Boolean) ::
Boolean

site (~)(Boolean) :: Boolean

Return the logical negation of the argument.

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

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

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.

[A]

type [A] = (:)(A, [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 `Array`s and Java `Iterable`s, 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.

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

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

`min` `def min(A,A) :: A, A <: Comparable`

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

Recall that the type constraint `A <: Comparable` means that `min` can only be applied to arguments which are subtypes of `Comparable`; in other words, they must have a total order.

`max` `def max(A,A) :: A, A <: Comparable`

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

B.3.2. data.inc: General-purpose supplemental data structures.

General-purpose supplemental data structures.

`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() :: Signal`

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() :: Signal`

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() :: Signal`

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() :: Signal`

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

`snoopnb` **site** `Semaphore.snoopnb() :: Signal`

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

`Buffer` **site** `Buffer() :: Buffer<A>`

Create a new buffer (FIFO channel) of unlimited size.

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 no items are available, block until one becomes available.

Recall that the type signature **site** `Buffer<A>.get() :: A` means that when the `get` method is called on a buffer holding an arbitrary element type `A`, it will return a value of the same type.

`getnb` **site** `Buffer<A>.getnb() :: A`

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

`put` **site** `Buffer<A>.put(A) :: Signal`

Put an item in the buffer.

`close` **site** `Buffer<A>.close() :: Signal`

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() :: Signal`

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() :: [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(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 removing an item opens a slot. A buffer with zero slots is equivalent to a synchronous channel [55].

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 no items are available, block until one becomes available.

`getnb` `site BoundedBuffer<A>.getnb() :: A`

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

`put` `site BoundedBuffer<A>.put(A) :: Signal`

Put an item in the buffer. If no slots are open, block until one becomes open.

`putnb` `site BoundedBuffer<A>.putnb(A) :: Signal`

Put an item in the buffer. If no slots are open, halt.

`close` **site** `BoundedBuffer<A>.close() :: Signal`

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() :: Signal`

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() :: 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) :: Signal`

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

`Cell` `site Cell() :: 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() :: Signal`

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

`Ref` `site Ref() :: Ref<A>`

Create a rewritable storage location without an initial value.

Example:

```
val r = Ref()
Rtimer(1000) >> r.write(5) >> stop
| println(r.read()) >>
  r.write(10) >>
  println(r.read()) >>
  stop
```

`Ref` `site Ref(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. The syntactic sugar `x?` is equivalent to `x.read()`.

`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) :: Signal`

Write a value to the ref. The syntactic sugar `x := y` is equivalent to `x.write(y)`.

`Array` **site** `Array(Integer) :: Array<A>`

Create a new native array of the given size.

Example:

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

`Array` **site** `Array(Integer, String) :: Array<A>`

Create a new primitive array of the given size with the given primitive type. The primitive type should match the element type of the array, 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.

`set` **site** `Array<A>.set(Integer, A) :: Signal`

Set the element of the array given by the index, counting from 0.

`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) :: Signal`

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(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)
```

`Option<A>` `type Option<A> = Some(A) | None()`

An optional value which is either provided or not.

Example:

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

`Either<A,B>` `type Either<A,B> = Left(A) | Right(B)`

A union which may be tagged as either "left" or "right".

Example:

```
-- Publishes: "left"
Left(3) >x> (
   x >Right(_)> "right"
 | x >Left(_)> "left"
)
```

`Set` **site** `Set() :: 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() :: Signal`

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() :: 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() :: Signal`

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() :: Signal`

Increment the counter.

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

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

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

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

Example:

```
-- Publishes five signals
val c = Counter(4)
repeat(c.dec)
```

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

Create a new dictionary (mutable record), initially empty. The first time each field of the dictionary is accessed (using dot notation), the record creates and returns a new empty Ref [56] which will also be returned on subsequent accesses of the same field. Dictionaries allow you to easily create simple data structures.

Example:

```
-- Publishes: 1 2
```

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

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

Return the first element of a pair.

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

Return the second element of a pair.

`swap` `def swap((A,B)) :: (B,A)`

Swap the elements of a pair.

B.3.3. 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(lambda (A, ...) :: B, [A]) :: B`

Apply a function to a list of arguments.

`curry` `def curry(lambda (A,B) :: C)(A)(B) :: C`

Curry a function of two arguments.

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

Curry a function of three arguments.

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

Uncurry a function of two arguments.

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

Uncurry a function of three arguments.

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

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

`constant` `def constant(A)() :: A`

Create a function which returns a constant value.

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

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

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

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

ignore

```
def ignore(lambda () :: B)(A) :: B
```

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

ignore2

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

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

compose

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

Compose two single-argument functions.

while

```
def while(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)
```

repeat

```
def repeat(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() >!_> f() >!_> f() >!_> ...`

fork

```
def fork([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()`

sequence

```
def sequence([lambda () :: A]) :: Signal
```

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

join

```
def join([lambda () :: A]) :: Signal
```

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() >> stop | g() >> stop | h() >> stop ; signal`

`por` **`def`** `por(`**`lambda`**` () :: Boolean, `**`lambda`**` () :: Boolean) :: Boolean`

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

`pand` **`def`** `pand(`**`lambda`**` () :: Boolean, `**`lambda`**` () :: Boolean) :: Boolean`

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

`collect` **`def`** `collect(`**`lambda`**` () :: A) :: [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))
```

B.3.4. 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]) :: A`

Publish every value in a list, simultaneously.

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

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

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

Return the reverse of the given list.

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

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

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

Return the first element of a list.

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

Return all but the first element of a list.

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

Return all but the last element of a list.

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

Return the last element of a list.

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

Is the list empty?

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

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

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

Return the first list concatenated with the second.

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

`foldl1` `def foldl1(lambda (A, A) :: A, [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.

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

`foldr1` `def foldr1(lambda (A, A) :: A, [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.

`afold` `def afold(lambda (A, A) :: A, [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`.

`cfold` `def cfold(lambda (A, A) :: A, [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`.

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

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

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

Split a list of pairs into a pair of lists.

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

Return the number of elements in a list.

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

Given a number `n` and a list `l`, return the first `n` elements of `l`.

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

Given a number `n` and a list `l`, return the elements of `l` after the first `n`.

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

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

`merge` `def merge([A], [A]) :: [A], A <: Comparable`

Merge two sorted lists.

Example:

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

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

Merge two lists using the given less-than relation.

`sort` `def sort([A]) :: [A], A <: Comparable`

Sort a list.

Example:

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

sortBy

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

Sort a list using the given less-than relation.

mergeUnique

```
def mergeUnique([A], [A]) :: [A], A <: Comparable
```

Merge two sorted lists, discarding duplicates.

Example:

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

mergeUniqueBy

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

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

sortUnique

```
def sortUnique([A]) :: [A], A <: Comparable
```

Sort a list, discarding duplicates.

Example:

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

sortUniqueBy

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

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

group

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

groupBy

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

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

`range` **def** `range(Integer, Integer) :: [Integer]`

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

`any` **def** `any(`**lambda** `(A) :: Boolean, [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.

`all` **def** `all(`**lambda** `(A) :: Boolean, [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.

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

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

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

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

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

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

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

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

`minimum` **def** `minimum([A]) :: A, A <: Comparable`

Return the minimum element of a non-empty list.

`maximum` **def** `maximum([A]) :: A, A <: Comparable`

Return the maximum element of a non-empty list.

B.3.5. 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, ...) :: Signal`

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, ...) :: Signal`

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(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"
```

`lines` **def** `lines(String) :: [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.

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

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

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

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

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

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

B.3.6. time.inc: Real and logical time.

Real and logical time.

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

Publish a signal after the given number of milliseconds.

`Clock` **site** `Clock()() :: Integer`

A call to `Clock` creates a new relative 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) :: Signal`

Publish a signal after the given number of logical timesteps. A logical timestep is complete as soon as all outstanding site calls (other than calls to `Ltimer`) have published.

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

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

B.3.7. 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() :: Double`

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(Site) :: Site`

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) :: Signal`

Publish the given number of signals, simultaneously.

Example:

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

`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)
```

`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)
```

`fillArray` `def fillArray(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)
```

`takePubs` `def takePubs(Integer, lambda () :: A) :: A`

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