Orc User Guide v2.0.0

Orc User Guide v2.0.0

Publication date 2011-04-14

License and Grant Information

This material is based upon work supported by the National Science Foundation under Grant No. CCF-0811536. Any opinions, findings, and conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Table of Contents

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/`.

Unless otherwise noted, all material in this document pertains to the Orc language implementation version 2.0.0 .

Chapter 1. The Orc Programming Language

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.

1.1. 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.

1.1.1. 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 |"`

1.1.2. 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"`.

Numbers with no decimal part, such as `3`, are treated as integers. Arithmetic operators with two integer arguments will perform an integer operation and return an integer result; for example, `5 / 2` performs integer division and evaluates to `2`. However, if either argument to an operator has a decimal part (even if it is trivial, as in `3.0`), the other argument will be promoted, and a decimal operation will be peformed. For example, `5 / 2.0` and `5.0 / 2` both perform decimal division and evaluate to `2.5`.

1.1.2.1. 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.

Warning

Silent expressions can produce side effects. For example, on encountering a type error, Orc will print an error message to the console. However the expression containing the type error will not publish a value, and in this respect it is silent.

1.1.3. 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`.

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.

1.1.4. 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.

1.1.5. Data Structures

Cor supports three basic data structures, *tuples*, *lists*, and *records*.

1.1.5.1. Tuples

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.

1.1.5.2. Lists

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.

1.1.5.3. Records

A *record expression* is a comma-separated sequence of elements of the form *f* `=` E, enclosed by record braces `{.` and `.}`, where each f is a field name and each E is an expression. Records may have any number of fields, including zero. Each expression is evaluated; the value of the whole record expression is a record containing an element for each field with its associated value. Order is irrelevant. If any of the expressions is silent, then the whole record expression is silent.

Examples

- `{. zero = 3 - 3, one = 0 + 1 .}` evaluates to `{. zero = 0, one = 1 .}`.
- `{. .}` evaluates to `{. .}`, the empty record.

Elements of records are accessed using a dot (`.`) syntax, as in most object oriented languages. The expression `r.f` evaluates to the value associated with field `f` in record `r`. If `f` is not present in `r`, the expression is silent.

Suppose `r = {. x = 0, y = 1 .}`

Examples

- `r.x` evaluates to `0`.
- `r.y` evaluates to `1`.
- `r.z` is silent.

Records can also be extended using the + operator. An expression `r + s`, where r and s are records, creates a new record which has all of the elements of `r` whose field names do not appear in `s`, and all of the elements of `s`. In other words, `s` overrides `r`. This use of `+` is left-associative and does not commute.

Examples

- `{. x = 0 .} + {. y = 1 .}` evaluates to `{. x = 0, y = 1 .}`
- `{. x = 0, y = 1 .} + {. y = 2, z = 3 .}` evaluates to `{. x = 0, y = 2, z = 3 .}`

1.1.6. 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 except a record. 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)`.

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.

1.1.7. 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

{-
OUTPUT:
10
-}
```

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
```

1.1.7.1. 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.

1.1.7.2. 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.

1.1.7.3. 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.

Since a function defined using `lambda` has no name, it is not possible to define a recursive function in this way. Only `def` can create a recursive function.

1.1.7.4. 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) =
  val (x,y) = H(n-1)
  (y,x+y)

def fib(n) =
  if (n < 0) then 0
  else (
    val (x,_) = H(n)
    x
  )
```

As a more complex example of matching, consider the following function which takes a list argument and returns a new list containing only the first `n` elements of the argument list.

```
def take(0,_) = []
def take(n,h:t) = if (n > 0) then h:(take(n-1,t)) else []
```

Mutual recursion and clausal definitions are allowed to occur together. Here are two functions, `stutter` and `mutter`, which are each mutually recursive, and each have multiple clauses. `stutter(l)` returns `l` with every odd element repeated. `mutter(l)` returns `l` with every even element 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)
```

1.1.8. 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
```

1.2. 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.

1.2.1. 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.

1.2.1.1. 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!")
```

1.2.2. 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.

1.2.2.1. 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. The resulting publications of F `|` G may be published in arbitrary order.

```
{- Publish 1 and 2 in parallel -}

1 | 1+1

{-
OUTPUT:PERMUTABLE
1
2
-}
```

```
{- 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")
```

```
{- Publish 1, 2, and 3 in parallel -}

1+0 | 1+1 | 1+2

{-
OUTPUT:PERMUTABLE
1
2
3
-}
```

For more information, see the Reference Manual (Parallel Combinator) .

1.2.2.2. 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 instance of G is executed in parallel with F (and with any previous copies of G); in that instance 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

{-
OUTPUT:PERMUTABLE
1
2
-}
```

```
{- Publish 3 and 4 in parallel -}

2 >n> (n+1 | n+2)

{-
OUTPUT:PERMUTABLE
3
4
-}


{- Publish 0, 1, 2 and 3 in parallel -}

(0 | 2) >n> (n | n+1)

{-
OUTPUT:PERMUTABLE
0
1
2
3
-}


{- 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 instance of G is started with all of the bindings from the match. Otherwise, the published value is simply ignored, and no new instance of G is executed.

```
{- Publish 3, 6, and 9 in arbitrary order. -}

(3,6,9)  >(x,y,z)>  ( x | y | z )

{-
OUTPUT:PERMUTABLE
3
6
9
-}


{- Filter out values of the form (_,false) -}

( (4,true) | (5,false) | (6,true) )  >(x,true)> x

{-
OUTPUT:PERMUTABLE
4
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

{-
OUTPUT:
goodbye
world
-}
```

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 | 2) >x> (3 | 4) >y> (x,y)

{-
OUTPUT:PERMUTABLE
(1,3)
(1,4)
(2,3)
(2,4)
-}
```

1.2.2.3. 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 *killed*. A killed 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)

{-
OUTPUT:
5
-}
{-
OUTPUT:
6
-}
```

```
{- 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")

{-
OUTPUT:PERMUTABLE
true
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) )

{-
OUTPUT:
9
-}
{-
OUTPUT:
25
-}
```

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)`.

1.2.2.4. 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 *halts*, 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.

1.2.3. Revisiting Cor expressions

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

1.2.3.1. 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-)`.

1.2.3.2. Conditionals

The conditional expression `if` E `then` F `else` G is actually a derived form based on two different sites named `If` and `Iff`. The sites take a boolean argument. `If` returns a signal if that argument is `true`, or remains silent if the argument is `false`. `Iff` returns a signal if that argument is `false` or remains silent if the argument is `true`.

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

1.2.3.3. `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`).

1.2.3.4. 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?

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)

{-
OUTPUT:
5
-}
{-
OUTPUT:
6
-}
```

```
{- Publish exactly one of 0, 1, 2 or 3 -}

(0 | 2) + (0 | 1)

{-
OUTPUT:
0
-}
{-
OUTPUT:
1
-}
{-
OUTPUT:
2
-}
{-
OUTPUT:
3
-}
```

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.

1.2.3.5. 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 binds the values of its arguments to the function's parameters, and then executes the function body in parallel with the computation of the bindings. Whenever the function body publishes a value, the function call publishes that value. Thus, unlike a site call, a 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)

{-
OUTPUT:PERMUTABLE
1
2
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 killed, and the corresponding parameter is bound to that value in the execution of the function body. Any part of the function body which uses a parameter that has not yet been bound blocks until that parameter is bound to a value.

1.2.4. 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 `Rwait`.

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

```
{- Print "red", wait for 3 seconds (3000 ms), and then print "green" -}

Println("red") >> Rwait(3000) >> Println("green") >> stop

{-
OUTPUT:
red
green
-}
```

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

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

We can also use `Rwait` 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")
           | Rwait(5000) >> "Search timed out.")
```

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

1.3. Advanced Features of Orc

1.3.1. 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*.

1.3.2. 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(_,_,_) | Empty()
{- 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.

1.3.3. Classes

A class is an abstraction mechanism in Orc, much like the `def` construct. It extends the `def` construct, by allowing us to convert an Orc program fragment into a site. Specifically, a class can be used to: (1) Define object classes with methods, (2) Create new sites and extend behaviors of existing sites, (3) Allow concurrent method invocation on objects, and (4) Create active objects, whose executions may be based on time or other external stimulus, not necessarily explicitly called methods.

A class has the following properties:

1. Encapsulation: Just like objects in an object-oriented language, classes also provide the encapsulation facility for the programmer. The data defined inside the class can only be accessed and modified through methods defined in class. Therefore, it hides the representation of the data and implementation of functions and methods which work on the data and manage the state.
2. Methods instead of functions: These are the gateways to access and manipulate the data represented by class. They also enable the user of the class to access the service provided by that. Unlike functions in Orc, methods defined in a class can only publish one value.
3. Termination protection: The execution of a class and its methods are protected from termination of the rest of the program. In other words, the execution of class cannot be interrupted. In the pruning combinator as soon as the right hand side expression publishes a value, all the on going executions in the right hand side will be terminated. However, if there is any call to a class in the right hand side, that call will proceed until completion. For example, in the following Orc expression, suppose that c is a class.

   ```
   y <y< g() | c()
   ```

   If g() publishes a value before c() does, the execution of c() will continue. Although, the published value of c() (if any) will never be used. On the other hand, if c() publishes a value first, then the execution of g() will be interrupted since its execution is not protected. This property of classes is very important to prevent the data corruption and invalid state of an object. We will see using an example the significance of this property later on in the guide.
4. Strict calls: unlike calls to ordinary function definitions in Orc, calls to class methods are strict. This means that the call will not happen unless all the parameters are bound.

A class definition can be translated to pure Orc calculus, making use of the site `makesite`. (See ??? for details of `makesite`.)

1.3.3.1. Object Definition

To motivate the notion of class, we start with a simple example, defining a stack with methods `push` and `pop`. Below, parameter `n` defines the maximum length of each instance of stack. We store the stack elements in an array. Our implementation blocks an illegal operation (a `push` on a full stack or a `pop` on an empty stack).

```
{- Define a simple stack in Orc -}

def class Stack(n) =
  val store = Table(n, Ref)
```

```
  val len = Ref (0) -- len is the current stack length

  def push(x) = Ift(len? <: n) >> store(len?) := x >> len := len? + 1

  def pop() = Ift(len? :> 0) >> len := len? - 1 >> store(len?)?

  {- class goal -} stop

{- Test the stack -}
val st = Stack(5)
st.push(3) >> st.push(5) >> st.pop() >> st.pop()

{-
OUTPUT:
3
-}
```

1.3.3.2. Class Syntax and Semantics

A class is defined much like a a function. The keyword `class` is used after `def`. A class may have parameters (including other classes), as in a function definition. The body of a class may include function and class definitions. The name of every definition in the body of a class is exported.

A class definition obeys all the rules of function definition except:

1. A class definition must include at least one definition (which could be a class definition),
2. A class's goal expression must not publish,
3. Each function defined within a class publishes at most once.

A class call creates and publishes a site (recall that the goal expression of a class must not publish). Thus,

```
val st = Stack(5)
```

instantiates site `st`. Multiple instances of a class may be created by calling the class multiple times. The functions (and classes) defined in the body of the class are externally accessed as dot methods.

1.3.3.3. Notes

- Clausal definition: A class may be defined as a set of clauses, exactly as a function definition. In this case all clauses must define the same names to be exported.
- Concurrent calls to methods: Methods of a class instance may be invoked concurrently, as in functions. It is the obligation of the programmer to ensure that concurrent calls do not interfere. Calling the methods of stack `st` concurrently may result in unintended outcomes. For example, in

  ```
    st.push(3) >> st.pop() >> Rwait(1000) >> st.pop()
  | st.push(4) >> stop
  ```

  the last `st.pop()` (in the first line) often does not succeed, because just one value was stored in the stack though there were two concurrent `st.push()` operations.
- Prune on class call: Like site calls, running class method invocations are not terminated by the pruning combinator. In the following program, `x` is assigned value 3 because execution of

`testprune().run()` never publishes. However, `testprune().run()` is treated as a site call which continues execution even after x is assigned a value. Eventually, the line `done` is printed.

```
def class testprune() =
  def run() =  Rwait(1000) >> Println("done") >> stop
  stop

  val x =  Rwait(50) >> 3 |  testprune().run()
  x
```

1.3.3.4. Class Example

The following shows the code for this sequence number generator in Orc:

```
{- Create a sequence number generator -}

{- mutable integer defined with seed zero -}
val seq_num = Ref(0)

{- gen_seq will increase the seq_num and return the new value -}
def gen_seq() =
  seq_num := seq_num?+1 >> seq_num?

gen_seq() >s> Println(s) >> gen_seq() >s> Println(s) >> stop

{-
OUTPUT:
1
2
-}
```

What is the problem with the above code? The first problem is that the seq_num is exposed to all the expressions which come after it. This means that all those expressions can possibly read and write the value of seq_num. This is not always desirable and makes the reasoning about the program more difficult. The second issue with this code is that it is not flexible enough. What if we want to have different sequence number generator with a different initial seed? For example, we may want a sequence number starting at 0 and another one starting at 1000.

There is another issue associated with the above code from a concurrent programming viewpoint. It is not thread safe. This means that if we make two parallel calls to gen_seq(), we may get the same result. For example, the following piece of code:

```
gen_seq() | gen_seq()
```

The above code can generate "1" as the publication for both expressions, and in most of the runs it will. This is a famous phenomenon in parallel programming known as a race condition. In fact, access to the shared variable seq_num is subject to race condition between the two threads running the two calls to gen_seq(). To solve the race issue we need to add a semaphore to the above code in order to regulate the access to the shared variable, seq_num. Therefore, every call to gen_seq needs to acquire the semaphore in order to update the variable and release the semaphore at the end. The new safe code is as follows:

```
{- Create a sequence number generator with a semaphore -}
{- the mutable integer defined with seed zero -}
val seq_num = Ref(0)

{- a semaphore to regulate access to shared variable seq_num -}
val seq_num_sem = Semaphore(1)

{- gen_seq will increase the seq_num and return the new value -}
def gen_seq() =
  seq_num_sem.acquire() >> seq_num := seq_num?+1 >>
  seq_num? >x> seq_num_sem.release() >> x

gen_seq() | gen_seq()

{-
OUTPUT:
1
2
-}
```

The above code, besides the issue of encapsulation and the complexity of having more than one sequence number generator, looks fine. However, there is a subtle problem with this code. The problem is data corruption and invalid state which can be caused by abrupt termination of the call to gen_seq(). Consider the following piece of code and corresponding execution order:

```
y <y< 1+1 | gen_seq()
```

```
1- 1+1
2- seq_num_sem.acquire()
3- y is bound to 2
4- terminate gen_seq()
```

As illustrated in the above execution sequence, the semaphore seq_num_sem got acquired but never released because of abrupt termination. Therefore consecutive calls to gen_seq will never proceed because they cannot acquire the semaphore. So, we need to somehow protect the execution of gen_seq. Classes address all the above issues in a succinct way.

The class solution to the sequence number generator is the following code:

```
{- Create a sequence number generator as a class -}

def class gen_seq(init) =
 {- the mutable integer defined with seed init -}
 val seq_num = Ref(init)

 {- the next function will increase the seq_num and return the new value -}
 def next() =
  seq_num := seq_num?+1 >> seq_num?
 signal
 val g = gen_seq(1000)

 g.next() >y> Println(y) >> g.next() >y> Println(y) >> stop

{-
OUTPUT:
1001
1002
-}
```

It is very easy to instantiate a new sequence number that starts with a different seed. We just need to create a new instance of gen_seq with a different seed. The following code shows an example of creating two sequence number generator with two different seeds.

```
val g0 = gen_seq(0)
val g1 = gen_seq(1000)

g0.next() >y> Println(y) >> g1.next() >y> Println(y) >> stop
```

Notice how easy it is to have a new sequence number generator. Comparatively, in a pure functional language where we just have functions, we need to declare new declarations of seq_num to accomplish the job.

Now, in order to solve the issue of thread safety, we want to add a semaphore to the class. Note that since the execution of the class is protected we will not run into the problem of invalid state or data corruption caused by abrupt termination. The following code shows the thread safe version of the sequence number generator.

```
{- Create a sequence number generator as a class with semaphore -}

def class gen_seq(init) =
 {- the mutable integer defined with seed init -}
 val seq_num = Ref(init)

 {- a semaphore to regulate access to shared variable seq_num -}
 val seq_num_sem = Semaphore(1)

 {- gen_seq will increase the seq_num and return the new value -}
 def next() =
  seq_num_sem.acquire() >> seq_num := seq_num?+1 >>
  seq_num? >x> seq_num_sem.release() >> x
 signal

{-
OUTPUT:
-}
```

1.3.3.5. More Class Examples

Here we show a few more examples of class usage.

1.3.3.5.1. Active Classes

Class is an active entity. Active objects, unlike passive ones, have their own thread of control. They can initiate a computation without receiving a method call (in procedural languages) or a message (in languages with message passing model). Note that this definition of active objects subsumes reactive objects and actors. We said that the execution of class is protected from the rest of the program. Here we want to clarify what do we mean by creation of the class and when the class actually exists.

A class exists as soon as all the parameters and free variables in the class are bound to their values. Thereafter, any method on the class can be called. This means that the creation process would not wait for goal expression of the class to finish. Therefore, the creation of the class would return immediately. But the goal expression continues to execute. The goal expression will not publish anything. However, it can have side effects. For example, it could print something on the screen or it can initiate some other processes and send and receive data to and/or from channels.

The following example explains the notion of activeness of classes. We want to design a clock that ticks every n (user specified) milliseconds. We want to be able to get the number of ticks passed since the process started. The following example shows a class implementation of this clock.

```
{- Define a class to track passage of time -}

def class nclock(n) = -- tick every n milliseconds
 {- tick_cnt counts the number of ticks -}
 val tick_cnt = Ref(0)

 {- getTick returns the number of ticks passed so far -}
 def getTick() = tick_cnt?
 metronome(n) >> tick_cnt:=tick_cnt?+1

 val ticker = nclock(250)
 {- prints a mutiple of 4 every one second -}
 Rwait(1000) >> metronome(1000) >> ticker.getTick()

{-
OUTPUT:
-}
```

In this example, the class nclock has a tick counter tick_cnt that records the number of ticks passed so far. The method getTick returns the number of ticks. The interesting point about this class is its goal expression. The goal expression is a metronome that publish a signal every n milliseconds. Each time a signal is issued it causes the tick_cnt to increase by one. As can be seen, the goal expression would never stop. It keeps increasing the tick counter for ever and a thread is always active in the nclock class instance. This program keeps publishing the multiples of 4 starting from 4 upward.

1.3.3.5.2. Multi-dimensional Matrix

We declare a two dimensional matrix whose index ranges may span any finite interval of integers. The same technique can be used to declare any multi-dimensional matrix.

```
{- Define a multi-dimensional matrix -}

def class Matrix((lo1, up1), (lo2, up2)) =
  val mat = Array((up1 - lo1 + 1) * (up2 - lo2 + 1))
  def access(i, j) = mat((i - lo1) * (up2 - lo2 + 1) + j)
  stop

val A = Matrix((-2, 0), (-1, 3)).access
A(-1,2) := 5 >> A(-1, 2) := 3 >> A(-1, 2)?

{-
OUTPUT:
3
-}
```

Note that we define matrix `A` to be the sole method, `access`, of the class; this enables us to refer to matrix elements in the traditional style.

1.3.3.5.3. Create a new site

We create a new site, a bounded channel, using `Channel` and `Semaphore` sites. The channel stores the data items and the semaphores are used to ensure proper blocking. Below, `n` is the maximum channel size, and `p` and `g` are semaphores whose values are the number of empty and full positions, respectively. A `put` operation is allowed only if `p > 0` and a `get` if `g > 0`.

```
def class BChannel(n) =
  val b = Channel()
  val (p, g) = (Semaphore(n), Semaphore(0))
  def put(x) = p.acquire() >> b.put(x) >> g.release()
  def get() = g.acquire() >> b.get() >x> p.release() >> x
  stop
```

Note that setting `n = 1` lets us define a 1-place channel in which the executions of `put` and `get` operations have to alternate.

1.3.3.5.4. Extend functionality of existing site

We add a `length` function to the `Channel` site, that returns its current length.

```
def class Channel() =
  val ch = Channel()
  val chlen = Counter(0)

  def put(x) = ch.put(x) >> chlen.inc()
  def get() = ch.get() >x> chlen.dec() >> x
  def length() = chlen.value()
  stop
```

1.3.3.5.5. A Communication protocol; Rendezvous

A set of senders and receivers communicate in the following manner. A sender executes `send(v)` and a receiver `recv()`. The `send(v)` remains blocked until some `recv()` operation is executed; similarly a `recv()` is blocked until there is a corresponding `send(v)`. When both `send(v)` and `recv()` operations are ready for execution, the `recv()` operation receives data `v`, `send` receives a signal, and both can then proceed.

We employ semaphore `s` on which the send operation blocks; `s` is released by the receive operation. The sender then puts its data in a channel and the receiver reads from the channel.

```
def class Rendezvous() =
  val (s,data) = (Semaphore(0), Channel())
  def send(x) = s.acquire() >> data.put(x)
  def recv() = s.release() >> data.get()
  stop
```

The following code fragment shows three threads that are forced to execute their codes in a nearly sequential manner due to the restrictions imposed by rendezvous.

```
val group1 = Rendezvous()
val group2 = Rendezvous()

  group1.send(3)
| Rwait(1000) >> group2.recv()
| group2.send(5) >> group1.recv()
```

1.3.3.5.6. A class operating in real time

We create a class to mimic a stopwatch. A stopwatch allows the following operations:

- `start()`: (re)starts the stopwatch and publishes a signal
- `halt()`: stops and publishes current value on the stopwatch

We implement an instance of a stopwatch by assigning a new clock to it (created by calling site `Clock()` that returns a new clock with value 0). Additionally, two mutable variables are used with the following meaning.

- `timeshown`: clock value when the stopwatch was last stopped,
- `laststart`: clock value when the stopwatch was last started.

Initially, both variable values are 0.

```
def class Stopwatch() =
  val clk = Clock()
  val (timeshown, laststart) = (Ref(0), Ref(0))

  def start() = laststart := clk()

  def halt() = timeshown := timeshown? + (clk() - laststart?) >> timeshown?
  stop
```

A useable implementation of stopwatch requires a more general interface. It should allow `start` operation in the state where the stopwatch is already running, and `halt` in a state where the stopwatch is already halted. Additionally, an operation to determine the staus of the stopwatch (running or not) should be provided. Such an implementation is given for the library site, `Stopwatch`; see the `Stopwatch` site in ???.

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 expressions to be combined are small, write them all on one line.

```
F | G | H
```

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

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

When the expressions to be combined are individually long 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 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 expressions to be combined 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

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
```

2.1.4.1. Ambiguous Declarations

The following situation could introduce syntactic ambiguity: the end of a declaration (def or val) is followed by an expression that starts with a non-alphanumeric symbol. Consider these example programs:

```
def f() =
  def g() = h
  (x,y)
```

```
def f() =
  val t = h
  (x,y)
```

```
def f() =
  val t = u
  -3
```

`(x,y)` may be interpreted as the parameter list of `h`, and `-3` as continuation of `u`, or they may be regarded as completely separate expressions (in this case, the goal expression of `def f`). To avoid this ambiguity, Orc imposes the following syntactic constraint:

An expression that follows a declaration begins with an alphanumeric symbol

To circumvent this restriction, if (x,y) is an expression that follows a declaration, write it as `signal >> (x,y)`. Similarly, write `signal >> -3`, in case `-3` is the goal expression in the above example. Note that there are many solutions to this problem; for example using `stop | (x,y)` is also valid.

2.2. Programming Idioms

In this section we give Orc implementations of some standard idioms from concurrent and functional programming. Despite the austerity of Orc's four combinators, we are able to encode a variety of idioms straightforwardly.

2.2.1. Channels

Orc has no communication primitives like pi-calculus channels[1] or Erlang mailboxes[2]. Instead, it makes use of sites to create channels of communication.

The most frequently used of these sites is `Channel`. When called, it publishes a new asynchronous FIFO channel. That channel is a site with two methods: `get` and `put`. The call `c.get()` takes the first value from channel `c` and publishes it, or blocks waiting for a value if none is available. The call `c.put(v)` puts `v` as the last item of `c` and publishes a signal.

A channel may be closed to indicate that it will not be sent any more values. If the channel `c` is closed, `c.put(v)` always halts (without modifying the state of the channel), and `c.get()` halts once `c` becomes empty. The channel `c` may be closed by calling either `c.close()`, which returns a signal once `c` becomes empty, or `c.closeD()`, which returns a signal immediately.

2.2.2. 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.

2.2.3. 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.

[1]R. Milner. *Communicating and Mobile Systems: the #-Calculus*. Cambridge University Press, May 1999.
[2]J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

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)
```

2.2.4. Mutable References

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`. It also provides the site `Array` to create mutable arrays.

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.

2.2.4.1. Rewritable references

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() | Rwait(1000) >> r.write(1) >> stop

{-
OUTPUT:
1
-}
```

2.2.4.2. Write-once references

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 r = Cell()
  Rwait(1000) >> r.write(2) >> println("Wrote 2") >> stop
| Rwait(1000) >> r.write(3) >> println("Wrote 3") >> stop
| r.read()

{-
OUTPUT:PERMUTABLE
2
Wrote 2
-}
```

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].

2.2.4.3. Syntax for manipulating references

Orc provides syntactic sugar for reading and writing mutable storage:

- `x?` is equivalent to `x.read()`. This operator is of equal precedence with the dot operator and function application, so you can write things like `x.y?.v?`. This operator is very similar to the C languages's `*` operator, but is postfix instead of prefix.
- `x := y` is equivalent to `x.write(y)`. This operator has higher precedence than the concurrency combinators and if/then/else, but lower precedence than any of the other operators.

Here is a previous example rewritten using this syntactic sugar:

```
{- 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 r = Cell()
  Rwait(1000) >> r := 2 >> println("Wrote 2") >> stop
| Rwait(1000) >> r := 3 >> println("Wrote 3") >> stop
| r?

{-
OUTPUT:PERMUTABLE
2
Wrote 2
-}
```

2.2.4.4. Arrays

While lists are a very useful data structure, they are not mutable, and they are not indexed. However, these properties are often needed in practice, so the Orc standard library provides a function `Array` to create mutable arrays.

`Array(n)` creates an array of size `n` whose elements are all initially `null`. The array is used like a function; the call `A(i)` returns the `i`th element of the array `A`, which is then treated as a reference, just like the references created by `Ref`. A call with an out-of-bounds index halts, possibly reporting an error.

The following program creates an array of size 10, and initializes each index i with the ith power of 2. It then reads the array values at indices 3, 6, and 10. The read at index 10 halts because it is out of bounds (arrays are indexed from 0).

```
{- Create and initialize an array, then halt on out of bounds read -}
val a = Array(10)
def initialize(i) =
  if (i <: 10)
    then a(i) := 2 ** i  >>  initialize(i+1)
    else signal
initialize(0) >> (a(3)? | a(6)? | a(10)?)
```

The standard library also provides a helper function `fillArray` which makes array initialization easier. `fillArray(a, f)` initializes array `a` using function `f` by setting element `a(i)` to the first value published by `f(i)`. When the array is fully initialized, `fillArray` returns the array `a` that was passed (which makes it easier to simultaneously create and initialize an array). Here are a few examples:

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

```
{- Create an array of 5 elements; each element is a newly created channel -}
fillArray(Array(5), lambda(_) = Channel())
```

```
{- Create an array of 2 channels -}
val A = fillArray(Array(2), lambda(_) = Channel())

{- Send true on channel 0,
   listen for a value on channel 0 and forward it to channel 1,
   and listen for a value on channel 1 and publish it.
-}

  A(0)?.put(true) >> stop
| A(0)?.get() >x> A(1)?.put(x) >> stop
| A(1)?.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)
```

2.2.5. Tables

Orc programs occasionally require a data structure that supports constant-time indexing but need not also provide mutable storage. For this purpose, an `Array` is too powerful. The standard library provides another structure called a `Table` to fill this role.

The call `Table(n,f)`, where `n` is a natural number and `f` a total function over natural numbers, creates and returns an immutable array of size `n` (indexed from 0), whose `i`th element is initialized to `f(i)`. A table can also be thought of as a partially memoized version of `f` restricted to the range [0, `n`). `Table` does not return a value until all calls to `f` have completed, and will halt if any call halts. Given a table `T`, the call `T(i)` returns the `i`th element of `T`. Notice that unlike array access, the `?` is not needed to subsequently dereference the return value, since it is not a mutable reference.

Tables are useful when writing algorithms which used a fixed mapping of indexes to some resources, such as a shared table of communication channels. Such a table could be constructed using the following code:

```
val size = 10
val channels = Table(size, lambda (_) = Channel())
```

Notice that the `lambda` ignores its argument, since each channel is identical. Here is another example, which memoizes the cubes of the first 30 natural numbers:

```
val cubes = Table(30, lambda(i) = i*i*i)
```

2.2.6. 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.

2.2.7. 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)>
  ( Ift(x = 3) >> signal
  | Ift(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 parallel matching technique 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`.

2.2.8. 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
   signal >> (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)
```

2.2.8.1. 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.

2.2.8.2. 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))
```

Note that all bidders are called simultaneously. Also note that if some bidder fails to return a bid, then the auction will never complete. Later we will see a different solution that addresses the issue of non-termination.

2.2.8.3. Example: Barrier synchronization

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

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

Suppose we would like to *synchronize* F and G, so that both start executing at the same time, after both `M()` and `N()` respond. This is easily done using the fork-join idiom. In the following, we assume that `x` does not occur free in G, nor `y` in F.

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

2.2.9. Sequential Fork-Join

Previous sections illustrate how Orc can use the fork-join idiom to process a fixed set of expressions or a list of values. Suppose that instead we wish to process all the publications of an expression F, and once this processing is complete, execute some expression G. For example, F publishes the contents of a text file, one line at a time, and we wish to print each line to the console using the site `println`, then publish a signal after all lines have been printed.

Sequential composition alone is not sufficient, because we have no way to detect when all of the lines have been processed. A recursive fork-join solution would require that the lines be stored in a traversable data structure like a list, rather than streamed as publications from F. A better solution uses the `;` combinator to detect when processing is complete:

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

Since `;` only evaluates its right side if the left side does not publish, we suppress the publications on the left side using `stop`. Here, we assume that we can detect when F halts. If, for example, F is publishing the lines of the file as it receives them over a socket, and the sending party never closes the socket, then F never halts and no signal is published.

2.2.10. Priority Poll

The otherwise combinator is also useful for trying alternatives in sequence. Consider an expression of the form F_0 ; F_1 ; F_2 ; .... If F_i does not publish and halts, then F_{i+1} is executed. We can think of the F_i's as a series of alternatives that are explored until a publication occurs.

Suppose that we would like to poll a list of channels for available data. The list of channels is ordered by priority. The first channel in the list has the highest priority, so it is polled first. If it has no data, then the next channel is polled, and so on.

Here is a function which polls a prioritized list of channels in this way. It publishes the first item that it finds, removing it from the originating channel. If all channels are empty, the function halts. We use the `getnb` ("get non-blocking") method of the channel, which retrieves the first available item if there is one, and halts otherwise.

```
def priorityPoll([]) = stop
def priorityPoll(b:bs) = b.getD() ; priorityPoll(bs)
```

2.2.11. Parallel Or

``Parallel or'' is a classic idiom of parallel programming. The ``parallel or'' operation executes two expressions F and G in parallel, each of which may publish a single boolean, and returns the disjunction of their publications as soon as possible. If one of the expressions publishes `true`, then the disjunction is `true`, so it is not necessary to wait for the other expression to publish a value. This holds even if one of the expressions is silent.

The ``parallel or'' of expressions F and G may be expressed in Orc as follows:

```
val result =
  val a = F
  val b = G
  Ift(a)  >>  true | Ift(b)  >>  true | (a || b)

result
```

The expression `(a || b)` waits for both `a` and `b` to become available and then publishes their disjunction. However if either `a` or `b` is true we can publish `true` immediately regardless of whether the other variable is available. Therefore we run `Ift(a) >> true` and `Ift(b) >> true` in parallel to wait for either variable to become `true` and immediately publish the result `true`. Since more than one of these expressions may publish `true`, the surrounding `val` is necessary to select and publish only the first result.

2.2.12. Timeout

Timeout, the ability to execute an expression for at most a specified amount of time, is an essential ingredient of fault-tolerant and distributed programming. Orc accomplishes timeout using pruning and the `Rwait` site. The following program runs F for at most one second, publishing its result if available and the value `0` otherwise.

```
Let( F | Rwait(1000) >> 0 )
```

2.2.12.1. Auction with timeout

In the auction example given previously, 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() | Rwait(8000) >> 0
  max(bid, auction(bs))
```

This version of the auction is guaranteed to complete within 8 seconds.

2.2.12.2. 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 result of the original expression with `true` and the result of the timer with `false`. Thus, if the expression does time out, then we can distinguish that case using the boolean value.

Here, we run expression F with a time limit `t`. If it publishes within the time limit, we bind its result to `r` and execute G. Otherwise, we execute H.

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

Instead of using a boolean and conditional, we could use pattern matching:

```
val s = Some(F) | Rwait(t) >> None()
  s >Some(r)> G
| s >None()>  H
```

It is even possible to encapsulate timeout as a function.

```
def timeout(x, t) = Let(Some(x) | Rwait(t) >> None())
```

`timeout(F, t)` waits `t` milliseconds for F to publish a value. If F publishes `v` within the time limit, `timeout` returns `Some(v)`. Otherwise, it returns `None()` when the time limit is reached.

2.2.12.2.1. Timeout streams

We can also apply timeout to streams. Let's define a modified version of the `repeat` function as follows:

```
def repeatWithTimeout(f, t) =
  timeout(f(), t)
    >Some(x)>
  (x | repeatWithTimeout(f, t))
```

We call `f()` as before, but apply a timeout of `t` to the call. If a value becomes available from `f` before the timeout, then the call to `timeout` publishes `Some(x)`, which we match, and then publish `x` and recursively wait for further values from the stream.

However, if no value is available from `f` within the timeout, the call to `timeout` publishes `None()`. Since `None()` does not match the pattern, the entire expression halts, indicating that the end of the stream has been reached.

It is also possible to achieve this behavior with the existing `repeat` function, simply by changing the function passed to `repeat`:

```
def f'() = timeout(f(), t) >Some(x)> x
repeat(f')
```

2.2.13. Priority

We can use a timer to give a window of priority to one computation over another. In this example, we run expressions F and G concurrently. For one second, F has priority; F's result is published immediately, but G's result is held until the time interval has elapsed. If neither F nor G publishes a result within one second, then the first result from either is published.

```
val x = F
val y = G
Let( y | Rwait(1000) >> x )
```

2.2.14. Metronome

A timer can be used to execute an expression repeatedly at regular intervals, for example to poll a service. Recall the definition of `metronome` from the previous chapter:

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

The following example publishes "tick" once per second and "tock" once per second after an initial half-second delay. The publications alternate: "tick tock tick tock ...". Note that this program is not defined recursively; the recursion is entirely contained within `metronome`.

```
  metronome(1000) >> "tick"
| Rwait(500) >> metronome(1000) >> "tock"
```

2.2.15. Routing

The Orc combinators restrict the passing of values among their component expressions. However, some programs will require greater flexibility. For example, `F <x< G` provides F with the first publication of G, but what if F needs the first n publications of G? In cases like this we use channels or other stateful sites to redirect or store publications. We call this technique *routing* because it involves routing values from one execution to another.

2.2.15.1. 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.

2.2.15.1.1. 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 = Channel()
repeat(c.get) <<
    F >x> c.put(x) >> stop
  | Rwait(1000) >> c.closeD()
```

This program allows F to execute for one second and then terminates it. Each value published by F is routed through channel `c` so that it does not terminate F. After one second, `Rwait(1000)` responds, triggering the call `c.closeD()`. The call `c.closeD()` closes `c` and publishes a signal, terminating F. The library function `repeat` is used to repeatedly take and publish values from `c` until it is closed.

2.2.15.1.2. 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 = Channel()
repeat(c.get) <<
  F >x>
    (if x >= 0
        then c.put(x) >> stop
        else c.closeD())
```

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, publishing a signal and causing the termination of F.

2.2.15.1.3. Interrupt

We can use routing to interrupt an expression based on a signal from elsewhere in the program. We set up the expression like a timeout, but instead of waiting for a timer, we wait for the semaphore `done` to be released. Any call to `done.release` will terminate the expression (because it will cause `done.acquire()` to publish), but otherwise F executes as normal and may publish any number of values.

```
val c = Channel()
val done = Semaphore(0)
repeat(c.get) <<
    F >x> c.put(x) >> stop
  | done.acquire() >> c.closeD()
```

2.2.15.1.4. 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 = Channel()
val done = Semaphore(0)
def allow(0) = done.release() >> stop
def allow(n) = c.get() >x> (x | allow(n-1))
allow(5) <<
    F >x> c.put(x) >> stop
  | done.acquire() >> c.closeD()
```

We use the auxiliary function `allow` to get only the first 5 publications from the channel `c`. When no more publications are allowed, `allow` uses the interrupt idiom to halt F and close `c`.

2.2.15.2. Non-Terminating Pruning

We can use routing to create a modified version of the pruning combinator. As in `F <x< G`, we'll run F and G in parallel and make the first value published by G available to F. However instead of terminating G after it publishes a value, we will continue running it, ignoring its remaining publications.

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

2.2.15.3. Sequencing Otherwise

We can also use routing to create a modified version of the otherwise combinator. We'll run F until it halts, and then run G, regardless of whether F published any values or not.

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

We use `c.close()` instead of the more common `c.closeD()` to ensure that G does not execute until all the publications of F have been routed. Recall that `c.close()` does not return until `c` is empty.

2.2.16. Interruption

We can write a function `interruptible` that implements the interrupt idiom to execute any function in an interruptible way. `interruptible(g)` calls the function `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`. For simplicity, we assume that `g` itself publishes no values.

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

```
def interruptible(f) =
  val done = Semaphore(0)
  done.release
    << f() >> stop
     | done.acquire() >> c.closeD()

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

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 `interruptible(g)`, including the execution of `g` itself.

The solution is to bind the variable differently:

```
def interruptible(f) =
  val done = Semaphore(0)
  done.release
    << f() >> stop
     | done.acquire() >> c.closeD()

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

This idiom, wherein 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.

2.2.17. Fold

We consider various concurrent implementations of the classic "list fold" function from functional programming:

```
def fold(_, [x])  = x
def fold(f, x:xs) = f(x, fold(xs))
```

This is a seedless fold (sometimes called `fold1`) which requires that the list be nonempty and uses its first element as a seed. This implementation is short-circuiting --- it may finish early if the reduction operator `f` does not use its second argument --- but it is not concurrent; no two calls to `f` can proceed in parallel. However, if `f` is associative, we can overcome this restriction and implement fold concurrently. If `f` is also commutative, we can further increase concurrency.

2.2.17.1. Associative Fold

We first consider the case when the reduction operator is associative. We define `afold(f,xs)` where `f` is a binary associative function and `xs` is a non-empty list. The implementation iteratively reduces `xs` to a single value. Each step of the iteration applies the auxiliary function `step`, which halves the size of `xs` by reducing disjoint pairs of adjacent items.

```
def afold(_, [x]) = x
def afold(f, xs) =
  def step([]) = []
  def step([x]) = [x]
  def step(x:y:xs) = f(x,y):step(xs)
  afold(f, step(xs))
```

Notice that `f(x,y):step(xs)` is an implicit fork-join. Thus, the call `f(x,y)` executes in parallel with the recursive call `step(xs)`. As a result, all calls to `f` execute concurrently within each iteration of `afold`.

2.2.17.2. Associative, Commutative Fold

We can make the implementation even more concurrent when the fold operator is both associative and commutative. We define `cfold(f,xs)`, where `f` is a associative and commutative binary function and `xs` is a non-empty list. The implementation initially copies all list items into a channel in arbitrary order using the auxiliary function `xfer`, counting the total number of items copied. The auxiliary function `combine` repeatedly pulls pairs of items from the channel, reduces them, and places the result back in the channel. Each pair of items is reduced in parallel as they become available. The last item in the channel is the result of the overall fold.

```
def cfold(f, xs) =
  val c = Channel()

  def xfer([])    = 0
  def xfer(x:xs)  = c.put(x) >> stop | xfer(xs)+1
  def combine(0) = stop
  def combine(1) =  c.get()
  def combine(m) =  c.get() >x> c.get() >y>
                    ( c.put(f(x,y)) >> stop | combine(m-1))

  xfer(xs) >n> combine(n)
```

2.3. Larger Examples

In this section we show a few larger Orc programs to demonstrate programming techniques. There are many more such examples available at the Orc website, on the community wiki [http://orc.csres.utexas.edu/wiki/Wiki.jsp?page=WikiLab].

2.3.1. Dining Philosophers

The dining philosophers problem is a well known and intensely studied problem in concurrent programming. Five philosophers sit around a circular table. Each philosopher has two forks that she shares with her neighbors (giving five forks in total). Philosophers think until they become hungry. A hungry philosopher picks up both forks, one at a time, eats, puts down both forks, and then resumes thinking. Without further refinement, this scenario allows deadlock; if all philosophers become hungry and pick up their left-hand forks simultaneously, no philosopher will be able to pick up her right-hand fork to eat. Lehmann and Rabin's solution[3], which we implement, requires that each philosopher pick up her forks in a random order. If the second fork is not immediately available, the philosopher must set down both forks and try again. While livelock is still possible if all philosophers take forks in the same order, randomization makes this possibility vanishingly unlikely.

```
{- Dining Philosophers -}

{- Randomly swap order of fork pick-up -}
def shuffle(a,b) = if (Random(2) = 1) then (a,b) else (b,a)

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

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

{- Define a philosopher -}
def phil(n,a,b) =
  def thinking() =
    Println(n + " thinking") >>
    if (Random(10) <: 9)
      then Rwait(Random(1000))
      else stop
  def hungry() = take((a,b))
  def eating() =
    Println(n + " eating") >>
    Rwait(Random(1000)) >>
    Println(n + " done eating") >>
    drop(a,b)
  thinking() >> hungry() >> eating() >> phil(n,a,b)

def philosophers(1,a,b) = phil(1,a,b)
def philosophers(n,a,b) =
  val c = Semaphore(1)
  philosophers(n-1,a,c) | phil(n,c,b)
```

[3]D. J. Lehmann and M. O. Rabin. On the advantages of free choice: A symmetric and fully distributed solution to the dining philosophers problem. In *POPL*, pages 133–138, 1981.

```
{- Test the definitions -}
val fork = Semaphore(1)
philosophers(5,fork,fork)

{-
OUTPUT:EXAMPLE
5 thinking
4 thinking
3 thinking
2 thinking
1 thinking
1 eating
4 eating
...
-}
```

The `phil` function simulates a single philosopher. It takes as arguments two binary semaphores representing the philosopher's forks, and calls the `thinking`, `hungry`, and `eating` functions in a continuous loop. A `thinking` philosopher waits for a random amount of time, with 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 relinquish ownership of her forks.

Calling `take(a,b)` attempts to acquire a pair of forks `(a,b)` in two steps: wait for fork `a` to become available, then immediately attempt to acquire fork `b`. The call `b.acquireD()` either acquires `b` and responds immediately, or halts if `b` is not available. If `b` is acquired, signal success; otherwise, release `a`, and then try again, randomly changing the order in which the forks are acquired using the auxiliary function `shuffle`.

The function call `philosophers(n,a,b)` recursively creates a chain of `n` philosophers, bounded by fork `a` on the left and `b` on the right. The goal expression of the program calls `philosophers` to create a chain of five philosophers bounded on the left and right by the same fork; hence, a ring.

This Orc solution has several nice properties. The overall structure of the program is functional, with each behavior encapsulated in its own function, making the program easy to understand and modify. Mutable state is isolated to the "fork" semaphores and associated `take` and `get` functions, simplifying the implementation of the philosophers. The program never manipulates threads explicitly, but instead expresses relationships between activities using Orc's combinators.

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
-}

{- Define philosopher -}
def philosopher(name, mbox, missing) =
  {- deferred requests for forks -}
  val deferred = Channel()
  {- 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()
    | Rwait(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() >>
    Rwait(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(Table(n, lambda (_) = Table(n, ignore(Channel))))

  {- 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)
  | Rwait(10000)
) >> "HALTED"

{-
OUTPUT:EXAMPLE
(0, 0) thinking
(0, 1) thinking
(0, 2) thinking
(1, 0) thinking
(2, 0) thinking
(1, 1) thinking
(2, 1) thinking
(1, 2) thinking
```

```
(2, 2) thinking
(1, 0) hungry
(0, 0) hungry
(0, 1) hungry
...
"HALTED"
-}
```

Our implementation is based on the actor model [http://en.wikipedia.org/wiki/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 [http://www.erlang.org/] 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 `Channel` 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 [http://en.wikipedia.org/wiki/Tail_call], 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 [http://en.wikipedia.org/wiki/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.

```
{- A solution to the readers-writers problem -}

{- Queue of lock requests -}
val m = Channel()
{- Count of active readers/writers -}
val c = Counter()

{- Process requests in sequence -}
```

```
def process() =
  {- Grant read request -}
  def grant((false,s)) = c.inc() >> s.release()
  {- Grant write request -}
  def grant((true,s)) =
    c.onZero() >> c.inc() >> s.release() >> c.onZero()
  {- Goal expression of process() -}
  m.get() >r> grant(r) >> process()

{- Acquire the lock: argument is "true" if writing -}
def acquire(write) =
  val s = Semaphore(0)
  m.put((write, s)) >> s.acquire()

{- Release the lock -}
def release() = c.dec()

-------------------------------------------------

{- These definitions are for testing only -}
def reader(start) = Rwait(start) >>
  acquire(false) >> Println("START READ") >>
  Rwait(1000) >> Println("END READ") >>
  release() >> stop
def writer(start) = Rwait(start) >>
  acquire(true) >> Println("START WRITE") >>
  Rwait(1000) >> Println("END WRITE") >>
  release() >> stop

Let(
    process()  {- Output:     -}
  | reader(10) {- START READ  -}
  | reader(20) {- START READ  -}
               {- END READ    -}
               {- END READ    -}
  | writer(30) {- START WRITE -}
               {- END WRITE   -}
  | reader(40) {- START READ  -}
  | reader(50) {- START READ  -}
               {- END READ    -}
               {- END READ    -}
  {- halt after the last reader finishes -}
  | Rwait(60) >> acquire(true)
)

{-
OUTPUT:EXAMPLE
END READ
START WRITE
END WRITE
START READ
START READ
END READ
END READ
signal
-}
```

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.

2.3.4. Quicksort

The original quicksort algorithm [4] was designed for efficient execution on a uniprocessor. Encoding it as a functional program typically ignores its efficient rearrangement of the elements of an array. Further, no known implementation highlights its concurrent aspects. The following program attempts to overcome these two limitations. The program is mostly functional in its structure, though it manipulates the array elements in place. We encode parts of the algorithm as concurrent activities where sequentiality is unneeded.

The following listing gives the implementation of the `quicksort` function which sorts the array `a` in place. The auxiliary function `sort` sorts the subarray given by indices `s` through `t` by calling `part` to partition the subarray and then recursively sorting the partitions.

```
{- Perform Quicksort on a list -}

def quicksort(a) =

  def swap(x, y) = a(x)? >z> a(x) := a(y)? >> a(y) := z

  {- Partition the elements based on pivot point 'p' -}
  def part(p, s, t) =
    def lr(i) = if i <: t && a(i)? <= p then lr(i+1) else i
    def rl(i) = if a(i)? :> p then rl(i-1) else i

    signal >>
      (lr(s), rl(t)) >(s', t')>
      ( Ift (s' + 1 <: t') >> swap(s', t') >> part(p, s'+1, t'-1)
      | Ift (s' + 1 = t') >> swap(s', t') >> s'
      | Ift (s' + 1 :> t') >> t'
      )

  {- Sort the elements -}
  def sort(s, t) =
     if s >= t then signal
     else part(a(s)?, s+1, t) >m>
          swap(m, s) >>
          (sort(s, m-1), sort(m+1, t)) >>
          signal
```

[4] C. A. R. Hoare. Partition: Algorithm 63, Quicksort: Algorithm 64, and Find: Algorithm 65. *Communications of the ACM*, 4(7):321–322, 1961.

```
  sort(0, a.length()-1)

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

The function `part` partitions the subarray given by indices `s` through `t` into two partitions, one containing values less than or equal to `p` and the other containing values > `p`. The last index of the lower partition is returned. The value at `a(s-1)` is assumed to be less than or equal to `p` --- this is satisfied by choosing `p = a(s-1)?` initially. To create the partitions, `part` calls two auxiliary functions `lr` and `rl` concurrently. These functions scan from the left and right of the subarray respectively, looking for out-of-place elements. Once two such elements have been found, they are swapped using the auxiliary function `swap`, and then the unscanned portion of the subarray is partitioned further. Partitioning is complete when the entire subarray has been scanned.

This program uses the syntactic sugar `x?` for `x.read()` and `x := y` for `x.write(y)`. Also note that the expression `a(i)` returns a reference to the element of array `a` at index `i`, counting from 0.

2.3.5. Meeting Scheduler

Orc makes very few assumptions about the behaviors of services it uses. Therefore it is straightforward to write programs which interact with human agents and network services. This makes Orc especially suitable for encoding *workflows*, the coordination of multiple activities involving multiple participants. The following program illustrates a simple workflow for scheduling a business meeting. Given a list of people and a date range, the program asks each person when they are available for a meeting. It then combines all the responses, selects a meeting time which is acceptable to everyone, and notifies everyone of the selected time.

```
include "net.inc"
val during = Interval(LocalDate(2009, 9, 10),
                      LocalDate(2009, 10, 17))
val invitees = ["john@example.com", "jane@example.com"]

def invite(invitee) =
  Form() >f>
  f.addPart(DateTimeRangesField("times",
    "When are you available for a meeting?", during, 9, 17)) >>
  f.addPart(Button("submit", "Submit")) >>
  SendForm(f) >receiver>
  SendMail(invitee, "Meeting Request", receiver.getURL()) >>
  receiver.get() >response>
  response.get("times")

def notify([]) =
  each(invitees) >invitee>
  SendMail(invitee, "Meeting Request Failed",
                    "No meeting time found.")
def notify(first:_) =
  each(invitees) >invitee>
  SendMail(invitee, "Meeting Request Succeeded",
                    first.getStart())

map(invite, invitees) >responses>
afold(lambda (a,b) = a.intersect(b), responses) >times>
```

```
notify(times)
```

This program begins with declarations of `during` (the date range for the proposed meeting) and `invitees` (the list of people to invite represented by email addresses).

The `invite` function obtains possible meeting times from a given invitee, as follows. First it uses library sites (`Form`, `DateTimeRangesField`, `Button`, and `SendForm`) to construct a web form which may be used to submit possible meeting times. Then it emails the URL of this form to the invitee and blocks waiting for a response. When the invitee receives the email, he or she will use a web browser to visit the URL, complete the form, and submit it. The corresponding execution of `invite` receives the response in the variable `response` and extracts the chosen meeting times.

The `notify` function takes a list of possible meeting times, selects the first meeting time in the list, and emails everyone with this time. If the list of possible meeting times is empty, it emails everyone indicating that no meeting time was found.

The goal expression of the program uses the library function `map` to apply `notify` to each invitee and collect the responses in a list. It then uses the library function `afold` to intersect all of the responses. The result is a set of meeting times which are acceptable to everyone. Finally, `notify` is called to select one of these times and notify everyone of the result.

This program may be extended to add more sophisticated features, such as a quorum (to select a meeting as soon as some subset of invitees responds) or timeouts (to remind invitees if they don't respond in a timely manner). These modifications are local and do not affect the overall structure of the program. For complete details, see examples on our website [http://orc.csres.utexas.edu/tryorc.shtml].