Orc User Guide

Orc User Guide

Table of Contents

1. The Orc Programming Language 1
    1.1. Introduction 1
    1.2. Cor: A Functional Subset 1
        1.2.1. Constants 2
        1.2.2. Operators 2
        1.2.3. Conditionals 3
        1.2.4. Variables 4
        1.2.5. Data Structures 4
        1.2.6. Patterns 5
        1.2.7. Functions 6
        1.2.8. Comments 9
    1.3. Orc: Orchestrating services 9
        1.3.1. Communicating with external services 10
        1.3.2. The concurrency combinators of Orc 11
        1.3.3. Revisiting Cor expressions 14
        1.3.4. Time 15
    1.4. Advanced Features of Orc 16
        1.4.1. Special call forms 17
        1.4.2. Extensions to pattern matching 18
        1.4.3. New forms of declarations 18
        1.4.4. The after combinator 19
2. Programming Methodology 21
3. Accessing and Creating External Services 22
A. Complete Syntax of Orc 23
B. Standard Library 24
C. FAQ 25

List of Tables

1.1. Syntax of the Functional Subset of Orc (Cor) 2
1.2. Operators of Cor 3
1.3. Basic Syntax of Orc 10
1.4. Advanced Syntax of Orc 16
A.1. Complete Syntax of Orc 23

Chapter 1. The Orc Programming Language

1.1. 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/`.

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.

Next, 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.

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

1.2. Cor: A Functional Subset

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

A Cor program is called 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.

The following figure describes the syntax of Cor. Each part of the syntax is explained in a subsequent section.

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

E ::=		*Expression*	
	C		*constant value*
|	E *op* E		*operator*
|	X		*variable*
|	**if** E **then** E **else** E		*conditional*
|	X(E , ... , E)		*function call*
|	(E , ... , E)		*tuple*
|	[E , ... , E]		*list*
|	D E		*scoped declaration*
C ::=	`true` | `false` | *integer* | *string*	*Constant*	
X ::=	*identifier*	*Variable*	
D ::=		*Declaration*	
	val P = E		*value declaration*
|	**def** X(P , ... , P) = E		*function declaration*
P ::=		*Pattern*	
	X		*variable*
|	C		*constant*
|	_		*wildcard*
|	(P , ... , P)		*tuple*
|	[P , ... , P]		*list*

1.2.1. Constants

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

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

- Integers: `... -1, 0, 1 ...`
- Booleans: `true` and `false`
- Strings: `"orc"`, `"ceci n'est pas une |"`

1.2.2. Operators

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

Examples

- `1+2` evaluates to `3`.
- `(98+2)*17` evaluates to `1700`.
- `4 = 20 / 5` evaluates to `true`.

- `3-5 >= 5-3` evaluates to `false`.
- `true && (false || true)` evaluates to `true`.
- `"leap" + "frog"` evaluates to `"leapfrog"`.

Here is the full set of operators that Cor supports:

Table 1.2. Operators of Cor

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

The `=` operator can compare values of any type.

Examples

- `10 = true` evaluates to `false`.

Sometimes, there are situations where an expression is stuck, because it is attempting to perform some impossible operation and cannot reach a value. If this is the case, we say that the expression is *silent*. An expression is also silent if it depends on the result of a silent subexpression.

Examples

- `10 / 0` is silent.
- `6 + false` is silent.
- `3 + 1/0` is silent.
- `4 + true = 5` is silent.

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

1.2.3. Conditionals

A conditional expression in Cor is of the form **`if`** E **`then`** F **`else`** G. Its meaning is similar to that in other languages: the value of the expression is the value of F if and only if E evaluates to true, or the value of G if and only if E evaluates to false. Note that G is not evaluated at all if E evaluates to true, and similarly F is not evaluated at all of E evaluates to false. Thus, for example, evaluation of **`if`** `3 = 3` **`then`** `5` **`else`** `1/0` does not cause any error.

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.

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

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

Examples

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

1.2.4. Variables

A *variable* names the value of some expression so that we can use it later. Expression `x`, where x is some variable name, evaluates to the result *bound* to the variable x.

Variables are bound using a *declaration*. A declaration is a statement that has no value of its own but instead binds one or more variables. 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 + 3 + 4 + 5
val y = x + x
```

These declarations bind variable `x` to 15 and variable `y` to 30.

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 can continue. If an evaluated expression depends on that variable, that expression is silent.

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

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

1.2.5. Data Structures

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

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

Examples

- `(1+2, 7)` evaluates to `(3,7)`.
- `("true" + "false", true || false, true && false)` evaluates to `("truefalse", true, false)`.
- `(2/2, 2/1, 2/0)` is silent.

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

Examples

- `[1,2+3]` evaluates to `[1,5]`.
- `[true && true]` evaluates to `[true]`.
- `[]` evaluates trivially 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.

Examples

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

1.2.6. Patterns

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

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

Examples

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

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

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

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

Examples

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

1.2.7. Functions

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

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

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

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

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

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

Examples

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

Notice that the evaluation strategy of functions allows to call to proceed even if some of the actual parameters to be 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 definition or a call may have zero arguments, in which case we write `()` for the arguments.

```
def zero() = 0
```

1.2.7.1. Recursion

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

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

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

Mutual recursion is also supported.

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

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

1.2.7.2. Closure

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

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

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

```
def onetwosum(f) = f(1) + f(2)
def triple(x) = x * 3
```

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

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

1.2.7.3. Lambda

Sometimes one would like to create a closure directly, without bothering to give it a name. There is a special keyword **`lambda`** for this purpose. By writing a function definition with the keyword **`lambda`** instead of a function name, that definition becomes an expression which evaluates to a closure.

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

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

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

1.2.7.4. Clauses

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

Here's an example.

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

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

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

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

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

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

```
{- Take up to the first n elements from a list -}
def take(0,_) = []
def take(_,[]) = []
def take(n,h:t) = h:(take(n-1,t))
```

Mutual recursion and clausal definitions are allowed to occur together. For example, this function takes a list and evaluates to a new list with every other 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)
```

1.2.8. Comments

Cor has two kinds of comments.

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

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

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

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

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

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

1.3. Orc: Orchestrating services

Cor is a pure declarative language. It has no state, since variables are bound at most once and cannot be reassigned. Evaluation of an expression results in at most one value. It cannot communicate with the outside world except by producing a value. Clearly, the full Orc language must transcend these limitations, because the orchestration of external services is critical to Orc's purpose.

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.

If an execution of an expression publishes no value, it is called *silent*. If no execution of an expression can possibly publish a value, then that expression is slient. However, unlike silent expressions in Cor, a silent expression in Orc might perform some meaningful computation or communication; silence does not necessarily indicate an error.

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

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

Table 1.3. Basic Syntax of Orc

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

1.3.1. Communicating with external services

We introduce the term *site* to denote an external service which Orc may call. Sites are called using the same syntax as a function call, but with a slightly different meaning. Sites are introduced and bound to variables by a special declaration.

1.3.1.1. Calling a site

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

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

Once the Google search service determines the top result, it sends a response. The site call then publishes that response. Note that the service might never respond: Google's servers might be down, the network might be down, or the search might yield no result URL. If the service fails to respond, the site call remains silent.

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

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

1.3.1.2. Declaring a site

A **`site`** declaration makes some service available as a site and binds it to a variable. The service might be an object in the host language (e.g. a class instance in Java), or an external service on the web which is accessed through some protocol like SOAP or REST, or even a primitive operation like addition. We will discuss the particulars of these declarations, and the guidelines for accessing web-based services or creating one's own services, in Chapter 3. Also, many useful sites are already defined in the Orc standard library, documented in Appendix B. For now, we present a simple type of site declaration: using an object in the host language as a site.

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

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

1.3.2. The concurrency combinators of Orc

Orc has three *combinators*: parallel, sequential, and asymmetric. A combinator forms an expression from two component expressions. Each combinator captures a different aspect of concurrency.

1.3.2.1. The parallel combinator

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

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


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

  Publish any results they return.

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

The parallel combinator is fully associative: (F **|** G) **|** H, F **|** (G **|** H), and F **|** G **|** H are all equivalent. It is also commutative: F **|** G is equivalent to G **|** F.

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

1.3.2.2. The sequential combinator

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

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

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


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


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


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

The sequential combinator may be written as F **>P>** G, where P is a pattern instead of just a variable name. Any value published by F is matched against the pattern P. If this match is successful, a new copy

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

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

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

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

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

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

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

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

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

1.3.2.3. The asymmetric combinator

The asymmetric 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 execuion of G is immediately *terminated*. A terminated expression cannot call any sites or publish any values.

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

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

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

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

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

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

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

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

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

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

1.3.3. Revisiting Cor expressions

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

1.3.3.1. `val`

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

1.3.3.2. `if`

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

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

When the **`else`** branch of a conditional is unneeded, we can use the **`if`** site and the sequential combinator to write the simpler expression **`if`**`(`E`)` **`>>`** F.

1.3.3.3. Treating Orc expressions functionally

We let Orc expressions with concurrency combinators occur as subexpressions of Cor expressions, for example as operands to arithmetic operations, or as arguments to function calls. The execution of an Orc

expression may publish many values and perform many site calls. What does such an expression mean in a purely functional context, where only one value is expected? For example, what does `2 + (3 | 4)` publish?

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

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

These are the contexts where only one value is expected from E. Whenever an Orc expression appears in such a context, it executes until it publishes its first value, and then it is terminated. The published value is used in the context as if it were the result of evaluating the expression.

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

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

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

1.3.3.4. Functions

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

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

```
-- Publish all integers in the interval 1..n, in arbitrary order.
def range(n) = if (n > 0) then (n | range(n-1)) else stop

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

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

1.3.4. Time

Orc is designed to communicate with the external world, and one of the most important characteristics of the external world is the passage of time. Orc implicitly interacts with the passage of time by calling

external services which take time to respond. However, Orc can also explicitly wait for some amount of time, using the special site `Rtimer`.

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

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

The following example defines a metronome, which publishes a signal once per second, indefinitely.

```
def metronome() = signal | Rtimer(1000) >> metronome()
```

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

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

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

1.4. Advanced Features of Orc

In this section we introduce some advanced features of Orc. These include curried function definitions and curried calls, a special syntax for writing calls in an object-oriented style, extensions to pattern matching, new forms of declarations, and a new concurrency combinator.

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

Table 1.4. Advanced Syntax of Orc

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

1.4.1. Special call forms

1.4.1.1. The . notation

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

There is a special kind of site call in Orc which serves a similar purpose. One may write `x.msg`, for any identifiers `x` and `msg`. This attempts to treat the value bound to `x` as a site, and call it with a special *message* value. 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 the value cannot be interpreted as a site, no call is made.

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

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

1.4.1.2. Currying

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

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

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

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

When defining a curried function, we have abstracted it in two steps, and when applying it we have written two separate calls. However, this is verbose and not very clear. Orc has a special syntax for curried definitions and curried applications that will simplify both of these steps. Here is the previous program, written in this simplified syntax:

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

Definitions may have multiple argument sequences; they are enclosed in parentheses and concatenated. Curried function calls chain together multiple applications in a similar way.

1.4.2. Extensions to pattern matching

1.4.2.1. Publish pattern

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

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

This publishes `2` and `3`.

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

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

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

1.4.2.2. As pattern

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

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

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

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

1.4.3. New forms of declarations

1.4.3.1. `class` declaration

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

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

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

This instance of String is a Java object; its methods are called and its fields are accessed using the `.` notation, just as one would expect in Java.

1.4.3.2. `include` declaration

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

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

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


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

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

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

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

1.4.4. The after combinator

Orc has a fourth concurrency combinator: the *after* combinator, written F **`;`** G. The after combinator executes its left side, publishing each of its publications as they occur. When the left side is done executing, then the right side starts executing.

We determine when an expression is done executing using the following rules.

- Any fully evaluated or silent Cor expression is done.
- A site call is done when it has published a value. A site call is also done if the site *refuses* the call, as explained below.
- F **`|`** G is done when its subexpressions F and G are done.
- F **`>x>`** G is done when F is done and all copies of G are done.
- F **`<x<`** G is done when F is done, and G is either done or has been terminated after publishing a value.
- F **`;`** G is done when its subexpressions F and G are done.
- **`stop`** is done.

A site refuses a call when it knows that it will never respond to that call. This may happen for a variety of reasons. A site may be specifically designed to never respond under certain circumstances; for example, **`if`**`(false)` will never respond, so **`if`** refuses that call. A site may depend on some connection that has failed in a detectable way; for example, `Google("five tons of flax")` if it is running on a computer which has no internet connection. A site may require some resource which is unavailable; for example, a site call based on a website may fail if the website itself is down or has moved (a 404 error). Generally speaking, any underlying exception or error which prevents the site call from ever continuing

will trigger a refusal. Refusal is used only to implement the after combinator; it is not used elsewhere in the language.

The after combinator is intended to capture as closely as possible the notion of sequential processing, as denoted by `;` in other languages. It was not present in the original formulation of the Orc concurrency calculus; it has been added to support computation and iteration over strictly finite data. Sequential programs conflate the concept of producing a value with the concept of completion. Orc separates these two concepts; variable binding combinators like `>x>` and `<x<` handle values, whereas `;` detects the completion of an execution.

Chapter 2. Programming Methodology

Chapter 3. Accessing and Creating External Services

Appendix A. Complete Syntax of Orc

Table A.1. Complete Syntax of Orc

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

Appendix B. Standard Library

Appendix C. FAQ