# Encore Programming Language Documentation

Version 0.45

Stephan Brandauer,
Elias Castegren,
Dave Clarke,
Kiko Fernandez,
Henrik Sommerland,
Tobias Wrigstad,
and Albert Yang

January 19, 2017

# 1 Introduction

Note that this document is a work in progress and may not be up to date.

Welcome to the Encore programming language! Please be aware that Encore is in its early development phase and is subject to change without notice.

For instructions about building encore please see the *Getting Up and Running* document.

# 2 Basics

Encore uses active classes which are classes with asynchronous method call semantics in order to achieve parallelism-by-default. This section gives a concise overview of encore's classes. The §3 "Syntax" and §4 "Semantics" sections will describe the language in much greater detail.

## 2.1 Hello, World!

The obligatory hello-world program, `hello.enc`, looks like this:

```
class Main
  def main() : void {
    print("Hello, World!")
  }
```

To compile and run the program just run `encorec -run hello.enc`.

```
$ encorec -run hello.enc
Hello, World
```

It is important to know that every legal `encore` program needs to have an active class called `Main`, with a method called `main`. The runtime will allocate one object of class `Main` and begin execution in its `main` method.

## 2.2 Compiler options

Running `encorec foo.enc` will typecheck the source and produce the executable `foo`. The following options are supported:

- `-c` – Keep intermediate C-files
- `-tc` – Typecheck only (don't produce an executable)
- `-o [file]` – Specify output file
- `-run` – Run the program and remove the executable
- `-clang` – Use clang to build the executable (default)
- `-AST` – Output the parsed AST as text to foo.AST
- `-TypedAST` – Output the typechecked AST as text to foo.TAST

- `-nogc` – Disable the garbage collection of passive objects.
- `-help` – Prints a help message and then exits.
- `-I path1:path2:...` – Directories in which to look for modules. (Not needed for modules which are in the current folder.)
- `-F [flags]` – Provides additional flags to the C compiler. Nessesary when using some C libraries. Use quatitionmarks to add more than one.

For instance, we might want to keep the intermediate C-files:

```
encorec -c foo.enc
```

## 2.3 Active/Passive Classes

`Encore` implements active and passive classes. Allocating an object from those classes gives active or passive objects, respectively. To make a class active, no special keyword is necessary, as classes are active by default: `class A`. To make a class passive, use `passive class P`.

Calling a method on an active object will have that method execute concurrently (also in parallel, if enough cores are available). The return type of an active method is a future (see §4.2 "Futures") and not the return type in the method declaration. Active classes are the default in `encore`.

In contrast to active classes passive classes executes methods synchronously, in the calling thread. Passive objects are similar to what you'd expect in a Java-like language.

Note that all fields in an active class are private but all fields are public in a passive class.

A class may have a method called `init` which is used as the constructor. It is called when appending arguments to object construction (`new Foo(42)`) and can not be called on its own. Note that there (currently) is a difference between `new Foo` and `new Foo()`. The first one only creates the object, while the latter one creates the object and immediately calls the `init` method on it.

### 2.3.1 Example: Active Class

We write an active class `Foo`. A foo has an ID string, calling the `printID` method will print that string 5 times to `stdout`. The main method creates two `Foo` instances, and have them print their IDs

```
class Main
```

```
  def main() : void
    let obj1 = new Foo
        obj2 = new Foo
    in {
      obj1.setID("obj1");
      obj2.setID("obj2");
      obj1.printID();
      obj2.printID();
      ()
    }

class Foo
  id : string

  def setID(new_id : string) : void {
    this.id = new_id;
    ()
  }

  def printID() : void {
    let i = 0 in {
      while i < 5 {
        i = i + 1;
        print(this.id)
      }
    }
  }
```

Executing this program gives nondeterministic output:

```
$ encorec -run ex_active.enc
obj2
obj1
obj2
obj1
obj2
obj1
obj2
obj1
obj2
obj1
```

The behavior of this program can be made deterministic using `get`. See §4.2 "Futures" for more information.

### 2.3.2 Example: Passive Class

A passive class is declared using the `passive` keyword. Note the use of a constructor (the `init` method):

```
#! /usr/bin/env encorec -run
--ex_passive.enc
passive class Location
  x : int
  y : int
  label : string

  def init(new_x:int, new_y:int, new_label:string) : void {
    this.x = new_x;
    this.y = new_y;
    this.label = new_label
  }

class Main
  loc : Location

  def main() : void {
    this.loc = new Location(1, 2, "a place");
    print this.loc.x;
    print this.loc.y;
    print this.loc.label
  }
```

Running:

```
$ ./ex_passive.enc
1
2
a place
```

# 3 Syntax

This section contains information about the grammar, keywords and literals of the Encore language.

## 3.1 Grammar

Please note that this grammar tree is out of date.

This section introduces the Encore grammar by using the BNF-grammar notation and show examples on how to build syntactically valid Encore programs.

⟨*Program*⟩ ::= [⟨*BundleDecl*⟩] ⟨*Imports*⟩* [⟨*EmbedTL*⟩] ⟨*TypeDef*⟩* ⟨*ClassDecl*⟩ | ⟨*eps*⟩
⟨*BundleDecl*⟩ ::= `bundle` ⟨*QName*⟩ `where`
⟨*TypeDef*⟩ ::= `typedef` ⟨*Name*⟩ [⟨*TypeParams*⟩] `=` ⟨*Type*⟩
⟨*TypeParams*⟩ ::= `<` ⟨*TypeVar*⟩ {`,` ⟨*TypeVar*⟩}* `>`
⟨*TypeVar*⟩ ::= [a-z] [a-zA-Z0-9_]*
⟨*ClassDecl*⟩ ::= [`passive`] `class` ⟨*Name*⟩ `{` ⟨*FieldDecls*⟩ ⟨*MethodDecls*⟩ `}`
⟨*Imports*⟩ ::= `import` [`qualified`] ⟨*QName*⟩ [`(`⟨*Name*⟩, ...`)`] [`as` ⟨*Name*⟩]
⟨*EmbedTL*⟩ ::= `embed` .* body .* `end` | `embed` .* `end`
⟨*FieldDecls*⟩ ::= ⟨*Name*⟩ `:` ⟨*Type*⟩ ⟨*FieldDecls*⟩ | ⟨*eps*⟩
⟨*ParamDecls*⟩ ::= ⟨*Name*⟩ `:` ⟨*Type*⟩ ⟨*ParamDecls*⟩ | ⟨*eps*⟩
⟨*MethodDecls*⟩ ::= `def` ⟨*Name*⟩ `(` ⟨*ParamDecls*⟩ `)` `:` ⟨*Type*⟩ ⟨*Expr*⟩
⟨*Sequence*⟩ ::= ⟨*Expr*⟩ ⟨*Seq*⟩ | ⟨*eps*⟩
⟨*Seq*⟩ ::= `;` ⟨*Expr*⟩ ⟨*Seq*⟩ | `;` | ⟨*eps*⟩
⟨*Arguments*⟩ ::= ⟨*Expr*⟩ ⟨*Args*⟩ | ⟨*eps*⟩
⟨*Args*⟩ ::= `;` ⟨*Expr*⟩ ⟨*Args*⟩ | ⟨*eps*⟩
⟨*LetDecls*⟩ ::= ⟨*Name*⟩ `=` ⟨*Expr*⟩ ⟨*LetDecls*⟩ | ⟨*eps*⟩
⟨*Expr*⟩ ::= `--` this is a comment
| `{-` this is a block comment `-}`
| `(` `)`
| `embed` ⟨*Type*⟩ .* `end`
| ⟨*Expr*⟩ `.` ⟨*Name*⟩
| ⟨*Expr*⟩ `.` ⟨*Name*⟩ `(` ⟨*Arguments*⟩ `)`
| ⟨*Expr*⟩ `!` ⟨*Name*⟩ `(` ⟨*Arguments*⟩ `)`
| `print` ⟨*Expr*⟩
| ⟨*Name*⟩ `(` ⟨*Arguments*⟩ `)`
| `(` ⟨*Expr*⟩ `)`
| ⟨*Name*⟩
| `liftv` ⟨*Expr*⟩
| `liftf` ⟨*Expr*⟩
| `join` ⟨*Expr*⟩
| `each` ⟨*Expr*⟩
| ⟨*Expr*⟩ `>>` ⟨*Arrow*⟩
| ⟨*Expr*⟩ `||` ⟨*Expr*⟩
| `let` ⟨*LetDecls*⟩ `in` ⟨*Expr*⟩

```
                | repeat ⟨Name⟩ <- ⟨Expr⟩ ⟨Expr⟩
                | for ⟨Name⟩ in ⟨Expr⟩ ⟨Expr⟩
                | for ⟨Name⟩ in ⟨Expr⟩ by ⟨Expr⟩ ⟨Expr⟩
                | ⟨Expr⟩ = ⟨Expr⟩
                | { ⟨Sequence⟩ }
                | if ⟨Expr⟩ then ⟨Expr⟩ else ⟨Expr⟩
                | if ⟨Expr⟩ then ⟨Expr⟩
                | unless ⟨Expr⟩ then ⟨Expr⟩
                | while ⟨Expr⟩ ⟨Expr⟩
                | match ⟨Expr⟩ with ⟨MatchClause⟩*
                | get ⟨Expr⟩
                | new ⟨Type⟩ ( ⟨Arguments⟩ )
                | new ⟨Type⟩
                | new [ ⟨Expr⟩ .. ⟨Expr⟩ ]
                | new [ ⟨Expr⟩ .. ⟨Expr⟩ by ⟨Expr⟩ ]
                | ⟨Expr⟩ [ ⟨Expr⟩ ]
                | [ ⟨Expr⟩ , ... ]
                | | ⟨Expr⟩ |
                | null
                | true
                | false
                | ⟨Char⟩
                | "String"
                | ⟨Int⟩
                | ⟨Real⟩
                | ⟨Option⟩
                | ⟨Expr⟩ ⟨Op⟩ ⟨Expr⟩
                | not ⟨Expr⟩
                | \ ( ⟨ParamDecls⟩ ) -> ⟨Expr⟩
⟨Op⟩          ::= <  |  >  |  ==  |  !=  |  +  |  -  |  *  |  /  |  %  |  and  |  or
⟨Option⟩      ::= Just ⟨Expr⟩
                | Nothing
⟨Name⟩        ::= [a-zA-Z][a-zA-Z0-9]*
⟨QName⟩       ::= ⟨Name⟩ [.⟨QName⟩]
⟨MatchClause⟩ ::= ⟨Expr⟩ [when⟨Expr⟩] => ⟨Expr⟩
⟨Int⟩         ::= [0-9]+
⟨Real⟩        ::= ⟨Int⟩ . ⟨Int⟩
⟨Char⟩        ::= [^']  |  \'
⟨String⟩      ::= ([^"]  |  \")*
⟨Type⟩        ::= ⟨Arrow⟩  |  ⟨NonArrow⟩
⟨Arrow⟩       ::= ( ⟨Types⟩ ) -> ⟨NonArrow⟩  |  ⟨NonArrow⟩ -> ⟨NonArrow⟩
⟨NonArrow⟩    ::= char
                | int
                | bool
                | void
                | ⟨RefType⟩
```

| ⟨*Fut*⟩ ⟨*Type*⟩
| ⟨*Par*⟩ ⟨*Type*⟩
| `(` ⟨*Type*⟩ `)`
| `[` ⟨*Type*⟩ `]`

⟨*Types*⟩ ::= ⟨*Type*⟩ ⟨*Tys*⟩ | ⟨*eps*⟩
⟨*Tys*⟩ ::= `,` ⟨*Type*⟩ ⟨*Tys*⟩ | ⟨*eps*⟩
⟨*RefType*⟩ ::= [A-Z] [a-zA-Z0-9_]*

## 3.2 Module system

As of now encore supports a rudimentary module system. The keyword used to import modules is `import`.

Here follows a trivial example of usage of the module system.

File `Lib.enc`:

```
bundle Lib where

class Foo:
  def boo():void {
              print "^_^"
  }
```

Line `bundle Lib where` declares the module name. This line is optional, though desirable for library code.

File `Bar.enc`:

```
import Lib

class Main:
  def main():void {
        let
              f = new Foo
        in{
              f.boo();
        }
  }
```

Here the file `Bar.enc` imports `Lib.enc` and can thus access the class `Foo`.

To import files from different directories one needs to use the `-I path` argument for the compiler.

Modules are hierarchical. Module `A.B.C` (in some directory `A/B/C.enc` in the include path) is declared using `bundle A.B.C where` and imported using `import A.B.C`.

As of now the module system has no notion of name spaces so all imported objects needs to have unique names. There is also no support for cyclic imports and no "include guards" so it's up to the programmer to ensure that each file is only imported once.

### 3.2.1 Standard Library

Encore supports a standard library, which is currently stored in the `bundles` directory of the `git` hierarchy, but in the future will be available in the directory `.encore/bundles` in a user's home directory.

The `bundles` directory contains three subdirectories. Directory `standard` includes stable library functionality. Directory `prototype` contains experimental and unstable libraries. These libraries should all be sub-bundles of the `Proto` bundle to remind the programmer of their status. Finally, directory `joy` contains bundles obtained through the `enjoy` package manager (functionality to be implemented).

## 3.3 Operators

Operators are special tokens that apply operations on expressions.

The following tokens are operators:

| | | | | | | |
|---|---|---|---|---|---|---|
| not | and | or | < | > | <= | >= |
| == | != | + | - | * | / | % |

## 3.4 Type Synonyms

Type synonyms allow abbreviations to be used for types. These take one of the two forms, where a single name can be used to abbreviate another type, or where a name has parameters.

```
typedef Fint = Foo<int,int>
```

or

```
typedef F<x,y> = Foo<y,x>
```

The types `Fint` and `F<type1,type2>` can be used in code as synonyms for `Foo<int,int>` and `Foo<type2,type1>`, respectively.

Type synonyms cannot be recursive.

## 3.5 Comments

In Encore you have one-liner comments and block comments. One-liner comments are written by using `--` while block comments are written starting with {- followed by the text of the comment and closing the comment with -}. For instance:

```
-- this expression prints numbers from 0 to 5
let i = 0 in
  {-
     this loop iterates 5 times
   -}
  while i < 5 {
    print("i={}\n",i)
    i = i+1
  }
```

## 3.6 Primitive Types

The available primitive types and example literals for them are:

```
char     'a'
real     1.234, -3.141592
int      1, -12
uint     42
bool     true, false
void     ()
```

While not technically a primitive type, there is special support for the `String` type. Strings can be created using the normal syntax `"Hello world!"` (which is just short-hand for `new String("Hello world!")`). The `String` class in `bundles/standard` provides basic operations on strings.

## 3.7 Loops

There are `while`, `repeat` and `for` loops.

A while loop takes a boolean loop condition, and evaluates its body expression repeatedly – as long as the loop condition evaluates to true:

```
let i = 0 in
  while i < 5 {
    print("i={}\n",i)
    i = i+1
  }
```

This prints:

```
i=0
i=1
i=2
i=3
i=4
```

The `repeat` look is syntax sugar that makes iterating over integers simpler. The following example is equivalent to the `while` loop above:

```
repeat i <- 5
  print("i={}\n",i)
```

Encore for loops can iterate over ranges of integers with uniform strides and arrays. They generalise `repeat` loops. To use `for` loops to iterate over the values 1 through 10 (inclusive), we can write:

```
for i in [1..10]
  print i
```

This prints:

```
1
2
3
...
10
```

For loops can be given an optional stride length using the `by` keyword. To modify the example above to iterate in strides of 3, we can write:

```
for i in [1..10] by 3
  print i
```

This prints:

```
1
4
7
10
```

Arrays can be sources of values for `for` loops. In the code below, let `arr` be an array `[1,2,3,4,5,6,7,8,9,10]`. The following two code snippets are equivalent to the two code snippets above. So, this prints 1 through 10 on the terminal:

```
for i in arr
  print i
```

And this prints 1, 3, 7, and 10 on the terminal:

```
for i in arr by 3
  print i
```

It is possible to loop over non-literal ranges as well. Notably, if `rng` is the range `[1..10 by 3]` then

```
for i in rng
  print i
```

prints

```
1
3
7
10
```

and

```
for i in rng by 2
  print i
```

prints

```
1
7
```

## 3.8 Arrays

For each type `T` there is a corresponding array type `[T]`. For example, `[int]` is the type of arrays of integers. You create a new array by writing `new [T](n)`, where `n` is the length of the array. An array has a fixed size and can not be dynamically extended or shrunk.

You access an array by using standard bracket notation `a[i]`. This is also how assignment into arrays is written. You get the size of an array by putting the array within bars `|a|`. You can also create array literals by writing a comma separated list of expressions within brackets `[1, 2, 1+2]`. This short example uses all features of arrays:

```
class Main
  def bump(arr : [int]) : void
      repeat i <- |arr|
        arr[i] = arr[i] + 1

  def main() : void{
    let a = [1,2,3] in {
      this.bump(a);
      repeat i <- |a|
        print a[i];
      let b = new [int](3) in {
        b[0] = 0;
        b[1] = a[0];
        b[2] = 42 - 19;
      }
    }
  }
```

The expected output is

```
2
3
4
```

## 3.9 Ranges

Ranges are Encore objects which are currently only useful for iterating over using `for` loops. There are currently no surface-level operations on ranges. Ranges may be passed around like normal objects. Pretending that ranges can be turned into their array equivalents, we can explain ranges by examples thus (ranges left, arrays right):

```
[1..5] == [1,2,3,4,5]
[0..4] == [0,1,2,3,4]
[2..6] == [2,3,4,5,6]
[0..100 by 30] == [0,30,60,90]
[0..4, by 2] == [0,2,4]
```

When ranges are used in `for` loops for iteration, they are optimised out, meaning there is no memory allocation due to a temporary range object.

# 4 Semantics

This section describes the semantics of the language.

## 4.1 Classes

Classes in `encore` have fields and methods. There is no class inheritance (but there are traits, see below). A class is either active or passive; all objects of an active class are active objects with asynchronous method calls and all objects of a passive class are passive objects with synchronous method calls.

Every encore program needs a class `Main` with a method `main`, which is the starting point of a program. The `main` method may take an array of strings as arguments. This array contains the command line arguments to the program.

### 4.1.1 Active Objects

Accessing fields of active objects is only possible via the `this` reference, which means that all fields are private. Passive objects have public fields.

```
#!/usr/bin/env encorec -run
class ActiveFoo
  sum : int

  def get_sum1() : int
    this.sum -- legal

  def get_sum2() : int
    let x = this in
      x.sum -- not legal

...
```

### 4.1.2 Passive Objects

Passive objects have public fields:

```
#!/usr/bin/env encorec -run

passive class Foo
  sum : int
```

```
  def init(sum_ : int) : void
    this.sum = sum_

class Main
  def main() : void
    let f = new Foo(12) in
      print("f.sum={}\n", f.sum) -- prints "f.sum=12"
```

### 4.1.3 Traits

Passive objects may implement one or several *traits*. A trait is like an interface with default implementations. It `require`s fields and methods and provides implemented methods. The class that implements a trait must provide all the fields and methods required by the trait (methods may have a more specialized return type), and will have its own interface extended with the methods provided by the trait:

```
trait Introduce
  require name : string
  require greeting() : string
  def introduce() : void
    print("{}! My name is {}\n", this.greeting(), this.name)

trait HasName
  require name : string

passive class Person : HasName + Introduce
  name : string
  def init(name : string) : void
    this.name = name

  def greeting() : string
    "Hello"

passive class Dog : HasName + Introduce
  name : string
  def init(name : string) : void
    this.name = name

  def greeting() : string
    "Woof"

class Main
  def meeting(x : Introduce, y : Introduce) : void{
    x.introduce();
```

```
    y.introduce();
  }

  def main() : void
    let p = new Person("Bob")
        d = new Dog("Fido")
    in
      this.meeting(p, d)
```

In this example, both `Person` and `Dog` are subtypes of the traits `Introduce` (and `HasName`) and can use the method `introduce`. Traits are (currently) only available for passive classes.

### 4.1.4 Parametric Class Polymorphism

Classes can take type parameters. This allows to implement, for example, 2-tuples:

```
passive class Pair<a, b>
  fst : a
  snd : b
  def init(fst_ : a, snd_ : b) : void{
    this.fst = fst_;
    this.snd = snd_
  }
```

We can now use the class like this:

```
class Main
  def main() : void
    let pair = new (Pair<int, String>)(65, "a") in
      print("({},{})\n", pair.fst, pair.snd)
```

### 4.1.5 Methods

As methods on active objects run asynchronously, they will not return the declared return type `a` but `Fut a` when called on any variable other than `this`. When calling a method on `this`, it will run synchronously and therefore not return a future but the result directly.

#### Expression Sequences

Syntactically, method bodies are a single expression:

```
def single() : void
  print "a single expression needs no curly braces"
```

```
def curly() : void {
  print ".. but it CAN use them!"
}
```

If you desire to run several expressions in sequence, you must use a <*Sequence*> expression:

```
def multiple() : int {
    print "multiple";
    print "expressions";
    print "are wrapped by { ... }";
    print "and separated by ';'";
    2
}
```

A sequence expression is a number of expressions, separated by semicolons and wrapped in curly braces. It evaluates to whatever the last subexpression evaluates. To simplify the introduction of variables, you can use the following syntax instead of normal let-expressions:

```
def variables() : String {
  print "Variables can be introduced anywhere in a sequence";
  let x = 42;
  let s = "Foo";
  print("x is {}", x);
  s;
}
```

## 4.2 Futures

As mentioned before (§2.3 "Active/Passive Classes"), a method call on an active class is executed asynchronously. Futures are produced when you call a method on an active class.

You can think about futures as a small data structure that will contain, eventually, a value when the asynchronous calculation finishes.

`Futures` are considered first class citizens. This allows us to pass `futures` to functions, return them from functions and use them as any other data type.

A `future` is considered to be **fulfilled** when the asynchronous operation has been finished and the `future` contains the returned value.

In the following subsections you will find a set of operations that can be performed on futures.

**Limitations**

The current implementation of futures does not allow more than 16 active objects blocking on a `future`.

### 4.2.1 `get` a value

As mentioned in the introduction to §4.2 "Futures", a future data type is returned whenever you call a method on an active class. Given the following code:

```
1 class Item
2   price: int
3
4   ...
5
6   def get_price(): int
7     this.price
8
9
10 class Main
11   item: Item
12
13   def init(i: Item): Main{
14     this.item = i;
15     this;
16   }
17
18   def execute(): void {
19     let fut = item.get_price() in {
20       print(get fut);
21       print("End payment!");
22     }
23   }
```

line 19 returns a `future` data type.

In order to get the value of the `future`, you need to do a call to the `get` method (line 20). `get` checks if the value of the future has been fulfilled. If it is, then returns a value of the expected type, in our case, an `int` (`get_price` returns an `int`, line 6). If the `future` is not fulfilled, then it blocks the active object and the `Main` actor will not do any progress until the future is fulfilled.

In this case, the statement `"End payment!"` in line 21 will always be printed after the print out of the price.

### 4.2.2 `await` execution

One of the problems of calling `get` on a future is the blocking behaviour of the actor. As we explained in §4.2.1 "`get` a value", this will block the actor until the future is fulfilled. If the method on which we get the future does a time-consuming calculation, the actor will be waiting until it finishes and the future is fulfilled.

A better alternative would be to let the actor continue running some other messages from its message queue and resume the actor when the future is fulfilled. This is exactly what the `await` on a future does.

Let's take a look at the example below:

```
1 class Main
2   def main() : void
3     let t = new Test() in
4     {
5       t ! run1();
6       t ! run2();
7     }
8
9 class Producer
10  def foo() : int
11    17
12
13 class Test
14  p:Producer
15  def init() : void
16    this.p = new Producer
17
18  def run2() : void {
19    print "While awaiting";
20  }
21
22  def run1() : void
23    let f = this.p.foo() in
24    {
25      print "Before await";
26      await f;
27      print "After await";
28      print get f
29    }
```

In this example, the output is:

```
Before await
```

```
While awaiting
After await
17
```

As you can see in the program, `Main` executes `run1` before `run2`. In the middle of the execution (line 26) it decides to `await` until the future is fulfilled and therefore, it starts executing `run2`. After finishing `run2`, the future has been fulfilled and `run1` continues from where it left off.

**NOTE**

In the current implementation, `await` would cause the actor to save the context and process other messages in the mailbox if the future is not fulfilled. Otherwise, `await` would behave like a no-op. When the future is fulfilled, the producer (the actor who fulfills the future) would send a message to awaited actor, who would resume the save context on processing this message.

### 4.2.3 `chain` on a `future`

The semantics of `chain` allows you to run a callback as soon as the future that you chain on is fulfilled. The result of a 'chain' operation is another 'Future' that will contain the result of the chained 'lambda'. For instance:

```
1  #!/usr/bin/env encorec -run
2
3  class Producer
4    def produce() : int
5      42 + 1
6
7  class Main
8    def main(): void {
9      let p = new Producer
10         l = \ (x:int) -> { print x; x + 1 } in {
11       print get p.produce() ~~> l;
12     }
13   }
```

In the example above, the lambda `lambda_fun` will be executed as soon as the future from `p.produce()` (line `10`) is fulfilled.

In the current implementation, we do not handle cyclic dependencies and they causes deadlock! We are working on it!

## 4.3 Tasks

### 4.3.1 async

Tasks allows the developer to execute a function asynchronously, not bound to the actor that calls it. For instance, in the following code, an actor calls a global function ‘long_computation‘, which is executed by the actor synchronously. This means that the actor will not be able to continue until the computation has finished.

```
1 def repeat(max_iterations: int, fn: int -> int): void {
2   repeat i <- max_iterations
3     print fn(i)
4 }
5
6 def inc(x: int): int
7   x+1
8
9 class Main
10  def main(): void
11    repeat(300, inc);
```

Sometimes you don’t care who executes the function, you just want to schedule the function and let someone run it. This is when tasks come in handy, for situations such as `pmap`, `foreach`, etc. The next example re-writes the previous code and makes it run by different actors, which doesn’t hog the actor that calls the global function.

```
1 def p_repeat(max_iterations: int, fn: int -> int): void
2   repeat i <- max_iterations
3     async(print fn(i))
4
5 def inc(x: int): int
6   x+1
7
8 class Main
9   def main(): void
10    p_repeat(300, inc)
```

Instead of calling a ‘async function‘, you can write statements inside the ‘async‘ construct that will be performed as a block of code. The following example, gets numbers from 0 to 4 and runs in parallel and asyncronously the quadruple of the given number. In this case, we do not want to block on the future, but execute the side effects (printing).

```
1 def square(x: int): int
2   x * x
3
```

```
class Main
  def main(): void {
    repeat i <- 5
      async {
        let s = square(i) in
          print square(s)
      }
  }
```

### 4.3.2 Finish

There's another feature that can be used with tasks and allows you to use your typical fork-join parallel construct. By using the keyword `finish` and tasks in its body you are guaranteed that the tasks will be finished before you leave the body of it.

E.g.

```
class Main
  def main():void {
    finish {
      async {
        -- perform asynchronous computation
      };
    };
    -- at this point, the task is fulfilled
    print "Finish"
  }
```

In this case, the asynchronous computation is performed before the program prints the word `Finish`.

In this other example (`async_finish.enc` in tests):

```
class Main
  def main(): void {
    let f = async{print "Task declared outside finish"} in {
      finish {
        async(print "Running inside finish");
        print 23;
        f
      };
      print "OUT";
    }
  }
```

the program returns an output similar to this one (due to non-determinism the order can change):

```
23
Running inside finish
OUT
Task declared outside finish
```

the important point to notice here is that `Task declared outside finish` is declared outside of the `finish` building block and therefore, it's not guaranteed to be finished before printing `OUT`.

### 4.3.3 foreach

`foreach` allows the developer to iterate over an array and execute the body of the foreach in parallel. It's used for performing parallel computations with side-effects.

E.g.

```
1 class Main
2   def main(): void
3     let master = new Master()
4         a = [1, 2, 3, 4, 5]
5     in
6       foreach item in a {
7         master.add(item)
8       };
```

In each iteration (line 7) `item` is replaced by the value `a[iteration]`. Line 7 is executed in parallel with other iterations and therefore, uses `tasks` behind the scenes. Please remember that this does not implies that by the end of the `foreach` construct all the tasks have finished!

## 4.4 Fire and forget`!`

In some occasions, when working on an active class, you might want to perform the side-effects of a method call and forget about the returned value. You could just throw away the future, but more performant way is not to generate the future in the first place. The bang`!` operator sends a message without creating a future. The following snippet shows how to perform this action:

```
1 class ShoppingCart
2   item: Item
```

```
3
4   def add_item(item: Item): Item {
5     this.item = item;
6     item;
7   }
8
9
10 class Main
11   def main(): void {
12     ...
13     let cart = ShoppingCart in {
14       cart!add_item(item);
15       ...
16     }
17    ...
18   }
```

In line 14, we use the symbol `!` to asynchronously execute the method call to the object represented by `item` without caring about the returned value.

## 4.5 `suspend` execution

The `suspend` operator is a cooperative multitasking abstraction that suspends the current running method on an agent and schedules the message to be resumed after the current messages in the inbox have been processed.

```
1 class Pi
2   def calculate_decimals(decimals: int): double {
3     -- perform initial calculations
4     ...
5     suspend;
6     -- continue performing more calculations
7     ...
8   }
9
10
11 class Main
12   def main(): void {
13     let pi = Pi in {
14       pi.calculate_decimals(100000000000);
15     }
16   }
```

Let's assume from the example above that we want to calculate a large number of decimals of $\pi$. The `Pi` active object does a method call to `calculate_decimals` (line `15`). This method starts performing some initial calculations and calls on `suspend`. As a result of this call, `Pi` will suspend the execution of the current running method call, place a new message in its message queue (to resume the execution of the running method) and continue processing other messages. Upon reaching the message that resumes the execution of the method `calculate_decimals`, it will continue from where he left off.

## 4.6 Parallel Combinators

Parallel combinators provide high- and low-level coordination of parallel computations. There are different operators that lift values and futures into a parallel collection. Other combinators are in charge of performing low-level coordination of this parallel collection.

### 4.6.1 liftv

The `liftv` combinator lifts a value to a parallel collection. Its signature is: `liftv :: t -> Par t`.

```
1 let p = liftv 42 in
2   p
```

### 4.6.2 liftf

The `liftf` combinator lifts a value to a parallel collection. Its signature is: `liftf :: Fut t -> Par t`.

```
1 class Card
2   def valid(): bool
3     return true
4
5 class Main
6   def main(): void
7     let card = new Card in
8       liftf(card.valid())
```

### 4.6.3 ||

The `||` combinator (named `par`) merges two parallel computations into a single parallel collection. Its type signature is: `|| :: Par t -> Par t -> Par t`.

```
class Card
  card: int

  def init(card: int): void
    this.card = card

  def valid(): bool
    return true

class Main
  def main(): void
    let card1 = new Card(1234)
        card2 = new Card(4567)
    in
      liftf(card1.valid()) || liftf(card2.valid())
```

### 4.6.4 >>

The >> combinator (named `sequence`) performs a `pmap` operation on the parallel collection. Its type signature is: `>> :: Par a -> (a -> b) -> Par b`.

```
-- prints if the card is valid
def show(valid: bool): void
  print valid

class Card
  card: int

  def init(card: int): void
    this.card = card

  def valid(): bool
    return true

class Main
  def main(): void
    let card1 = new Card(1234)
        card2 = new Card(4567)
        p = liftf(card1.valid()) || liftf(card2.valid())
    in
      p >> show
```

### 4.6.5 join

The `join` combinator merges two parallel computations into a single one. Its type signature is: `Par Par t -> Par t`.

```
1 class Main
2   def main(): void
3     let par = liftv 42  -- :: Par int
4         par_par = liftv par  -- :: Par Par in
5     in
6       (join par_par)  -- :: Par Par int -> Par int
```

### 4.6.6 each

The `each` combinator lifts an `array` to a parallel collection. Currently, this combinator runs in a single thread. If the collection is too big, this is not performant. As a temporary fix, you can define a flag (`PARTY_ARRAY_PARALLEL`) in `encore.h` to create workers that process a fixed amount of items from the array. Future work will improve this to create and remove workers based on some runtime metrics, instead of on a fixed amount of items. Its type signature is: `[t] -> Par t`.

The following example shows how to calculate the euclidian distance for the K-means algorithm using the `each` combinator:

```
1 class Kmeans
2
3   def _euclidian(xs: (int, int), ks: (int, int)): int
4     let x1 = get_first(xs)
5         x2 = get_second(xs)
6
7         k1 = get_first(ks)
8         k2 = get_first(ks)
9
10        z1 = square(x1-k1)
11        z2 = square(x2-k2)
12    in
13      square_root(z1 + z2)
14
15  def distance(observations: [(int, int)], ks: (int, int)): int
16    each(observations) >> \ (xs: (int, int) -> _euclidian(xs, ks))
```

## 4.7 Embedding of C code

For implementing low level functionality, `encore` allows to embed trusted C code. There are two kinds of embedded C code: expressions, and toplevel `embed` blocks.

We do **not** advocate to rely on the `embed` functionality too much. Code that uses `embed` is highly likely to break with future updates to the language (even more likely than code that doesn't use `embed`).

### 4.7.1 Toplevel Embed Blocks

There can be at most one toplevel embed block. It has to be before the first class definition of a file. It consists of a header section and an implementation section.

```
embed
  // the header section
  int sq(int);
body
  // the implementation section
  int sq(int n) {
    return n*n;
  }
end
```

It is useful to define functions or global variables, include C-header files or to define constants.

The header section will end up in a header file that all the class implementations will include. The implementation section will end up in a separate C-file. Consequently, if the `sq` function would not be included in the header section, it could not be used later.

### 4.7.2 Embedded expressions

When embedding an expression, the programmer needs to assign a type to the expression; `encorec` will assume that this type is correct. The value an embedded expression evaluates to is the return value of the last C-statement in the embedded code.

If you want to access local `encore` variables in an `embed` expression, you'll need to wrap them: `#{x}` for accessing the local variable `x`. If you want to access fields, you don't wrap them, but use C's arrow operator: `this->foo` for `this.foo`.

```
#! /usr/bin/env encorec -run
embed
```

```
  #include<math.h> // for sqrtl
  int64_t sq(int);
body
  int64_t sq(int n) {
    return n*n;
end
  }

class Main
  def main() : void {
    let x = 2 in {
      -- Contrary to the toplevel embed block, an embed expression
      -- also needs to specify a certain type.
      -- In this example, the expression promises to return an int:
      print(embed int sq(#{x}); end);
      -- ..and here it returns a real:
      print(embed real sqrtl(#{x}); end)
  }
}
```

The following table documents how `encore`'s types map to C types:

| **encore** | **C** |
|---|---|
| `char` | `char` |
| `real` | `double` |
| `int` | `int64_t` |
| `uint` | `uint64_t` |
| `bool` | `int64_t` |
| <any active class type> | `pony_actor_t*` |
| <any passive class type> | `CLASSNAME_data*` |
| <a type variable> | `encore_arg_t` |

## 4.8 Formatted printing

The `print` statement allows formatted output:

```
#! /usr/bin/env encorec -run
class Main
  def main() : void {
    let i = 0 in {
      while i < 5 {
        i = i+1;
        print("{} * {} = {}\n", i, i, i*i);
      }
    }
```

```
  }
```

Running:

```
$ ./ex_printing.enc
1 * 1 = 1
2 * 2 = 4
3 * 3 = 9
4 * 4 = 16
5 * 5 = 25
```

## 4.9 Anonymous Functions

Encore has anonymous functions:

```
let f = \ (i : int) -> 10*i in
  print(f(10)) -- prints 100
```

The backslash \ syntax is borrowed from Haskell and is supposed to resemble a lambda. It is followed by a comma separated list of parameter declarations, an arrow `->` and an expression that is the function body.

The type of a function is declared similarly. The function `f` above has type `int -> int`. Multi-argument functions have types such as `(int, String) -> bool`.

```
#! /usr/bin/env encorec -run
-- ex_lambdas.enc

passive class Point
  x : int
  y : int

  def scale(f : int -> int) : void {
    this.x = f(this.x);
    this.y = f(this.y);
    ()
  }

class Main
  def main() : void {
    let p = new Point in {
      p.x = 1;
      p.y = 2;
      -- scale the point up by 10, using an anonymous function:
```

```
      let f = \ (i : int) -> 10*i in {
        p.scale(f);
      };
      print(p.x);
      print(p.y)
    }
  }
```

## 4.10 Polymorphism

There is limited support for polymorphic methods and functions. Type variables are written with lower case letters and do not need to be declared. If there is a type variable in the return type it must appear somewhere in the types of the arguments. Here is an example program that uses a polymorphic function:

```
class Main
  def main() : void
    let
      apply = \ (f : a -> b, x : a) -> f(x)
    in
      let bump = \ (x : int) -> x + 1 in
        print(apply(bump, 3))
```

## 4.11 Option types

Option types, (sometimes called 'Maybe types') are polymorphic types that represent optional values. Using option types is simple: a value is wrapped inside a `Just` data constructor (e.g. `Just 32`), while `Nothing` presents an absent value.

Option types are useful to substitute the `null` reference to objects. The following example shows the use of option types in a function that creates a string (wrapped inside a `Just`) if the input value is greater than `0` and `Nothing` otherwise.

```
def test_if_greater_than_zero(x: int): Maybe String {
  if x > 0 then Just("Test passes")
  else Nothing
}
```

There exists limited support for type inference when using the `Nothing` data constructor. This means that, sometimes, you will have to cast the type of `Nothing`. In the previous example, you would substitute the `Nothing` (line 3) by `Nothing : Maybe String`.

The section related to pattern matching (below) explains how to extract the value (if it exists) from an option type.

## 4.12 Pattern Matching

Pattern matching allows the programmer to branch on both the value of data, as well as its structure, while binding local names to sub-structures. Encore's match expression looks like this:

```
match theArgument with
  pattern when optionalGuard => handler -- First match clause
  pattern2 => handler2 -- Second match clause
```

The match expressions's argument will be matched against each of the patterns in the order they are written. If both a pattern and its guard matches the handler is executed and no more clauses are checked. The match expression will evaluated to the same value as the selected handler. If no clause matches a runtime error will the thrown and the program will crash.

### 4.12.1 Patterns

In Encore there are 5 different types of patterns. First is the value pattern, it matches whenever the value in the pattern is equal to the match expression's argument. Equality is checked using C's `==` except for on strings when C's `strcmp` function is used. Do not use value patterns to match on objects.

```
match 42 with
  4711 => print "This doesn't match."
  42 => print "But this does."
```

Second we have the variable pattern. It matches any expression and also binds it so that the variable may be used inside both the guard and the handler.

```
match 42 with
  4711 => print "This doesn't match."
  x => print "This matches, and x is now bound to the value 42."
```

Third we have the `Maybe` type patterns: `Just` and `Nothing`. Any number of other patterns can be nested inside the Just pattern. In this example a simple variable pattern have been used.

```
match myMap.lookup(42) with
  Just res => print("myMap is associating 42 with {}.", res)
  Nothing => print "myMap had no association for 42."
```

Fourth is the tuple pattern. It will recursively match all its components against those of the argument, and will only match if all components match.

```
match (42, "foo") with
  (3, "wrong") => print "This doesn't match."
  (x, "foo") => print "But this does."
```

Finally there is the extractor pattern that is used to match against objects. Extractor patterns arise from the definition of extractor methods, which are simply methods with the `void -> Maybe T` type signature, for some type `T`.

In the `Link` class below, the method `link` implicitly (because of its type) defines an extractor pattern with the same name.

```
passive class Link
  assoc : (int, string)
  next : Link

  def link() : Maybe((int, string), Link)
    Just (this.assoc, this.next)
```

When an extractor pattern is used, the corresponding extractor method will be called on the argument of the match. If it returns a `Just`, then the inside of the `Just` will be recursively matched with the argument of the extractor pattern.

```
def loopkup(key : int, current : Link) : Maybe int
  match current with
    null => Nothing : Maybe int
    link((k, v), next) when k == key => Just v
    link(x, next) => this.lookup(key, next)
```

Note that you can define many different extractor methods for the same class, and that they may return `Nothing` under some circumstances, in which case the match for that clause will fail. You are also allowed to call extractor methods outside of a pattern, in which case they will function exactly like any other method.

Regular functions may not be called inside a pattern. Anything that looks like a function call will be interpretted and typechecked as an extractor pattern.

### 4.12.2 Guards

Each match clause may have an optional guard. If there is a guard present, it must evaluate to `true` for the handler to be executed. If it evaluates to `false` the matcher will proceed to check the next clause. The following code demonstrates how to use guards in Encore:

```
match 42 with
    x when x < 0 => print("{} is negative", x)
```

```
    x when x > 0 => print("{} is positive", x)
    x => print("{} is zero", x)
```

### 4.12.3 Multi-headed functions

Encore also allow you to use pattern-matching when defining functions, methods and streams. However, anonymous functions do not support patterns directly in the function head.

```
def myFactorial(0 : int) : int {
  1
}\| myFactorial(n : int) : int {
  n * myFactorial(n-1)
}
```

One limitation of the current implementation is that you have to declare the type of the function in every function head, even though the type has to be the same for all clauses.