

# Concurrent Kittens in Little Boats

Brian Beckman

24 Mar 2023

## Contents

<b>1</b>	<b>Prologue</b>	<b>2</b>
1.1	“What are you doing?” . . . . .	2
1.2	“Why not RhoLang?” . . . . .	2
1.3	“Why not something better?” . . . . .	3
1.4	“Clojure doesn’t have types!” . . . . .	3
1.5	“How did you write this?” . . . . .	3
<b>2</b>	<b>Π Kittens in Little Boats</b>	<b>4</b>
2.1	A Cartoon . . . . .	4
2.2	Free and Bound . . . . .	5
2.3	Binding . . . . .	6
2.4	Substitution . . . . .	6
2.5	Renaming . . . . .	6
2.6	Animated Cartoons . . . . .	6
2.7	Finishing Up . . . . .	7
2.8	All Names are Channels . . . . .	7
2.9	Bail the Boats! . . . . .	7
2.10	Kitten Boat Grammar . . . . .	9
<b>3</b>	<b>Channels and Names</b>	<b>10</b>
3.1	Kit-1 . . . . .	10
3.2	Kit-2 . . . . .	11
3.3	Kit-3 . . . . .	12
3.4	Kitten Zero — the Whisper Boat . . . . .	12
<b>4</b>	<b>Change Log</b>	<b>13</b>

## 1 Prologue

I must confess a mild mental disability: when I see mathematics or programs, I see drowning kittens, and I want to save them.

$1+2*3$ , five little kittens drowning in the sea. Venus-the-boatwright rises on her half-shell, fists full of bows and sterns. She reaches down and builds little boats, each from a bow and a stern. She puts some little kittens in each boat. Sometimes, she puts another little boat in a boat in place of a kitten.

She kisses the first kitten (or boat) in each boat and says “You’re the captain; take care of the others!” She blesses the little boats and sinks back into the sea, leaving  $(+ \ 1 \ (* \ 2 \ 3))$  sailing safely on the waves.

Have you been burnt by C code like  $x \mid y \ \& \ z$  or  $x \mid y \ * \ z$ ? Perhaps you can dream, like me, of saving the drowning kittens. Wright the boats in dry-dock<sup>1</sup> or asea<sup>2</sup>, right the boats so they float,<sup>3</sup> and write  $(\& \ z \ (\mid \ x \ y))$  and  $(\ * \ z \ (/ \ x \ y))$ ! If not, perhaps you can tolerate me. My affliction is not my fault! It is like misophonia<sup>4</sup>: built-in, incurable, inexplicable (except to fellow sufferers).<sup>5</sup>

### 1.1 “What are you doing?”

Let’s implement some fundamental terms from the rho calculus<sup>6</sup> and the pi calculus<sup>7</sup> in a DSL<sup>8</sup> made of little boats, meaning “embedded in Clojure.” Asynchronous behavior is easy to model in Clojure, and we have `clojure.spec`<sup>9</sup> for type checking. Plus Clojure looks like little boats to save the drowning kittens. It’s important for programs to look good. Mathematics and programming *are visual arts*.

### 1.2 “Why not RhoLang?”

You call up your friend Nancy and invite her to a party at your house. She asks “Will Ted be there?” You think to yourself *what fun!* and innocently say, “Yes, I think so! I invited him!” Nancy says “I’m terribly sorry, but I won’t be able to make it to your party. Please have a great time and give my best to everyone!” Nancy hangs up and thinks to herself *except to Ted*.

Ted’s last name is Scala.

Let’s compromise. Nancy won’t ever see Scala again, plus she can’t stand to see kittens drowning, i.e., implicit precedence rules. Otherwise she likes RhoLang.

<sup>1</sup>compiled ahead-of-time (AOT)

<sup>2</sup>interpreted or compiled just-in-time (JIT)

<sup>3</sup>type-check and optimize

<sup>4</sup><https://www.webmd.com/mental-health/what-is-misophonia>

<sup>5</sup>Plus, I don’t like writing parsers: it’s boring.

<sup>6</sup>Meredith, L. G.; Radestock, Mattias (22 December 2005). “A Reflective Higher-Order Calculus”. *Electronic Notes in Theoretical Computer Science*. 141 (5): 49–67. <https://doi.org/10.1016/j.entcs.2005.05.016>.

<sup>7</sup><https://en.wikipedia.org/wiki/%CE%A0-calculus>

<sup>8</sup>[https://en.wikipedia.org/wiki/Domain-specific\\_language](https://en.wikipedia.org/wiki/Domain-specific_language)

<sup>9</sup><https://clojure.org/guides/spec>

Let's create something with the same semantics as RhoLang embedded in Clojure, and call it CrowLang.

Perhaps CrowLang can inter-operate with RhoLang. They're both on the Java Virtual Machine, after all. Or perhaps we'll write new lcompilers<sup>10</sup> [*sic*] for CrowLang and RhoLang (you can write an lcompiler for Scala, Nancy doesn't care). Lcompilers are fast, flexible, modularized, and easy to write. In fact, some day, lcompilers will use rho and pi for internal type-checking and other formalisms!

### 1.3 “Why not something better?”

Considered and rejected:

**Agda** — too obscure

**Haskell or Mathematica** — I think Mathematica is my all-time favorite programming language. Sadly, no one else will use it. Haskell is in my top-five favorites, rejected for the same reason.

**Python** — well, ok, umm, never mind, no

**Common Lisp** — not modern, otherwise fantastic!

**Racket** — designed for DSLs after all, but no one else uses it

**Coq** — full of rabbit holes, otherwise lovely!

**C++** — I would drive us both crazy.

There *is* a path of least resistance for me, considering all things.

### 1.4 “Clojure doesn't have types!”

Not so. Clojure.spec<sup>11</sup> is at least as strong as types. It's not static, that's true, but `s/conform` *could* be static.

Static or not hardly matters in our case. We can build little boats in drydock before setting sail, or Venus-the-boatwright will build them at sea for us. We can check types, we can do rewriting, we can prove theorems. It's good enough.

Also, Clojure is already in our toolchain. We use it for abstract interpretation and test generation<sup>12</sup> for lfortran and lpython.<sup>10</sup>

### 1.5 “How did you write this?”

This is an executable document. When I produce a PDF from it, all code blocks are executed and results reported. I like this better than Jupyter notebooks for many reasons. This document is an instance of Knuth's literate programming<sup>13</sup> in org-babel.<sup>14</sup>

---

<sup>10</sup><https://github.com/lcompilers>

<sup>11</sup><https://clojure.org/guides/spec>

<sup>12</sup><https://github.com/rebcabin/asr-tester>

<sup>13</sup>[https://en.wikipedia.org/wiki/Literate\\_programming](https://en.wikipedia.org/wiki/Literate_programming)

<sup>14</sup><https://orgmode.org/worg/org-contrib/babel/>

## 2 $\Pi$ Kittens in Little Boats

From the wiki page:<sup>7</sup>

$P, Q ::=$		
$0$	napping kitten	Do nothing; halt.
$x(y). P$	listening kitten	Listen on channel $x$ for channel $y$ .
$\bar{x}(y). P$	chatting kitten	Say " $y$ " on channel $x$ ; don't wait.
$P \mid Q$	two kittens	Run $P$ and $Q$ in parallel.
$(\nu x) P$	whispering kitten	fresh channel name $x$ ; Use it in $P$ .
$!P$	mama cat	Run copies of $P$ forever.

(1)

This pi calculus is low-level, like  $\lambda$  calculus, only with concurrency added. We'd have to build up numbers (like Church numerals), Booleans, sets, functions, conditionals, loops, everything. We'll do a little better, later. First, let's save some drowning kittens!

### 2.1 A Cartoon

Here is a tiny calculation cartoon, again from the wiki page, showing a reduction similar to an  $\eta$ -reduction in  $\lambda$  calculus:

$$\begin{aligned}
 (\nu x) \quad ( & \bar{x}(z).0 \\
 & | \quad x(y). \bar{y}(x). x(y).0 \\
 & | \quad z(v). \bar{v}(v).0 \\
 & )
 \end{aligned}
 \tag{2}$$

We have four drowning kittens. Kitten Zero is a whispering kitten  $(\nu x) P$ . She whispers " $x$ " to the other kittens, meaning "let's talk on channel  $x$ !" One might write:

$$(\nu x) \left( \begin{array}{l} \bar{x}(z).0 \\ | \quad x(y). \bar{y}(x). x(y).0 \\ | \quad z(v). \bar{v}(v).0 \end{array} \right)
 \tag{3}$$

Kitten Zero is obviously going to be the captain of a boat containing the remaining kittens, but we've only just started to wright boats.

*We'll say "kitten" and mean "an actual kitten, or a little boat containing kittens or more little boats." Each little boat contains zero-or-more kittens and zero-or-more more littler boats.*

One of the three remaining kittens is chatting on channel  $x$  and the other two are listening, one on  $x$  and the other on  $z$ . These three are doing their things two at a time,  $P \mid (Q \mid R)$  or  $(P \mid Q) \mid R$ , it doesn't matter how you think about it (*associativity of par*).<sup>15</sup>

<sup>15</sup>A better *par* boat could hold any number of kittens, in any order. We'll get there.

Kitten One,  $\bar{x}\langle z \rangle.0$ , chats on  $x$ , “Hey, let’s continue chatting on  $z$ !” Then she takes a nap. Only one of the other kittens, Kitten Two,  $x(y).\bar{y}\langle x \rangle.x(y).0$ , listens on  $x$ .<sup>16</sup> She thinks, “Oy! Here I am waiting on  $x$  for someone to tell me where (on what channel) to continue, and I just heard ‘continue on  $z$ ,’ so I’ll switch to  $z$ . After I switch, I’ll have something to say, but just let me switch, first!”

$$(\nu x) \left( \begin{array}{c} \bar{x}\langle z \rangle.0 \\ \parallel \downarrow \\ | \quad x(y).\bar{y}\langle x \rangle.x(y).0 \\ \quad \downarrow \downarrow \\ | \quad x(z).\bar{z}\langle x \rangle.x(y).0 \\ | \quad z(\nu).\bar{\nu}\langle \nu \rangle.0 \end{array} \right) \quad (4)$$

The out-channel  $\bar{x}$  of Kitten One matches the in-channel  $x$  of Kitten Two; Kitten One said something and Kitten Two heard it. The *say-prefix*,  $\bar{x}\langle z \rangle$ , before the dot of Kitten One,  $\bar{x}\langle z \rangle.0$ , gets gobbled up, and Kitten One takes a nap. The *hear-prefix*,  $x(y)$ , before Kitten Two’s first dot, the first dot of  $x(y).\bar{y}\langle x \rangle.x(y).0$ , also gets gobbled up. Plus,  $y$  changes to  $z$  in the next say-prefix,  $\bar{y}\langle x \rangle$ , of Kitten Two’s first suffix,  $\bar{y}\langle x \rangle.x(y).0$ :

$$(\nu x) \left( \begin{array}{c} 0 \\ | \quad \bar{z}\langle x \rangle.x(y).0 \\ | \quad z(\nu).\bar{\nu}\langle \nu \rangle.0 \end{array} \right) \quad (5)$$

This is rather like the substitution of actual arguments for formal parameters in a function call in an ordinary programming language.

## 2.2 Free and Bound

Before this substitution of  $z$  for  $y$ , the  $y$  in Kitten Two’s next say-prefix,  $\bar{y}\langle x \rangle$ , is *free*. That means it must change to  $z$ . However, the second  $y$  in Kitten Two’s first suffix,  $\bar{y}\langle x \rangle.x(y).0$ , isn’t free. The  $y$  in  $x(y)$  doesn’t change to  $z$  because that  $y$  is *local* to the final suffix,  $0$ , of  $x(y).0$ . If that that suffix had more to do, the name  $y$  would be unconnected to the earlier appearances of  $y$ . In this case, the suffix takes a nap, so we don’t have to worry.

*Bound* is a synonym for *not free*. A variable  $y$  is either free or bound in a prefix,  $(\nu x)$ ,  $\bar{y}\langle x \rangle$ , or  $x(y)$ . It can’t be both except in the one strange case  $x(x)$ . Once it’s bound, it’s bound in all the suffixes to the right;  $y$  would be bound in the suffixes,  $P.Q.\dots$ , of  $x(y).P.Q.\dots$ , at least up until the next binding. Any re-bindings of  $y$  in a long suffix  $P.Q.\dots$  pertain to the closest binding, looking leftward. That closest binding must be a whispering kitten or a listening kitten.

<sup>16</sup>If more than one kitten listens on the same channel, we have a classic race condition. A compiler can detect this directly from the syntax of the program! At run time, only one will hear and the other will starve, at least for a while.

## 2.3 Binding

There are only two ways to bind a name; only two *binding prefix forms*:

**whispering** —  $(\nu x) P$  binds  $x$  in its suffixes  $P$ , up until the next binding.

**listening** —  $x(y)$ .  $Q$  binds  $y$  in its suffixes  $Q$ , up until the next binding.

**Definition 1.** *binding, scope*: Each binding of a given name, say  $y$ , pertains to the entire suffix of its binding form, up until the next binding of  $y$ . That new binding *shadows* the prior binding. This is like the *environment model* or *lexical binding* of an ordinary programming language. A sequence of binding prefixes describes a right-hugging nest of *scopes* in which the bound variable have known values. ■

We don't yet know *bound to what*? We can only find out later when a hear-prefix lines up with a say-prefix again. So this usage of the word *bound* means *eventually bound to something*. The term *bound* by itself can be ambiguous, because we might also say *bound* when we *do* know *bound to what*.

## 2.4 Substitution

We can state a general rule for *substitution*, with some terminology to be clarified:

**Definition 2.** *substitution*: When the channel  $x$  of a left-most say-prefix,  $\bar{x}\langle z \rangle$ , equals the channel  $x$  of a left-most hear-prefix,  $x(y)$ , the prefixes are gobbled up and all free occurrences of  $y$  on the right of the hear-prefix suffer substitution of  $z$  for  $y$ . ■

## 2.5 Renaming

What if there were already some  $z$ 's amongst the free  $y$ 's? The listening kitten would have to patch that up first. It doesn't matter what temporary bound name she gives to a channel, so long as the same channel has the same bound name everywhere to the right of some dot. We might rename preexisting  $z$ 's something like  $z_1$ . That's *alpha renaming*. We don't have that problem here, but we might later. Our kittens always remember their sailorly duty to clean up messes in their boats.

**Definition 3.** *renaming*: Prior to substitution of  $z$  for a free variable  $y$  in the suffixes of a hear-prefix, any bound occurrences of  $z$  to the right of the hear-prefix must be renamed consistently lest they collide with the incoming  $z$  that replaces  $y$ . ■

Renaming in the suffixes happens first, then substitution happens to the free variables.

## 2.6 Animated Cartoons

I can't animate cartoons in a paper, but I visualize calculations as symbols moving around in an animated cartoon (please forgive my synaesthetic mental affliction). It saves me mistakes. I animate calculations with pen and paper.

## 2.7 Finishing Up

Kitten Two,  $\bar{z}\langle x \rangle. x(z). 0$ , now says on  $z$  “Switch to  $x$ , will you?” to whomever is listening. Then she waits and listens on  $x$ . Kitten Three,  $z(v). \bar{v}\langle v \rangle. 0$ , is listening on  $z$  for a channel. She temporarily calls that channel  $v$ , but now she knows that  $v$  is really  $x$ :

$$(\nu x) \left( \begin{array}{c|c} 0 & \\ \hline x(z). 0 & \\ \hline \bar{x}\langle x \rangle. 0 & \end{array} \right) \quad (6)$$

See how the  $z$  chat-listen pair got gobbled up and how  $x$  got substituted for  $v$ ? If not, do an animation on paper. Kitten Three didn’t have to patch up any lingering  $x$ ’s, but she remembers to check.

Kitten Three says “ $x$ ” on  $x$  and takes a nap without waiting. Kitten Two hears on  $x$  that her temporary channel name  $z$  really should be  $x$  again. She changes her  $z$  to  $x$ , notices she doesn’t have any patching up or anything else to do, and takes a nap. If you don’t see it in your mind’s eye, animate it on paper.

$$(\nu x) \left( \begin{array}{c|c} 0 & \\ \hline | & 0 \\ \hline | & 0 \end{array} \right) \quad (7)$$

All the kittens are napping safely in the whisperer’s boat.

## 2.8 All Names are Channels

Every variable,  $x$ ,  $y$ ,  $z$ ,  $v$ , stands in for a communication channel. Sometimes we know what channel a variable stands for, say after substitution. Other times, a variable stands for a channel we’ll find out about later. That’s all we have so far: channels, known or unknown.

## 2.9 Bail the Boats!

For now, we’ve got all kittens safely napping in the “whisper” boat. But they’re not *dry*. They had to bail out a *lot* of water to keep from drowning whilst Venus-the-boatwright was working. Venus will fix that with some little boats *inside* other boats, including the biggest “whisper” boat.

Venus first bails out most of the water, leaving little skeletal, boats-in-progress — ordinary mathematical function notation:

$$(\nu x) \left( \begin{array}{c|c} \text{say}(x, z, 0) & \\ \hline | & \text{hear}(x, y, \text{say}(y, x, \text{hear}(x, y, 0))) \\ \hline | & \text{hear}(z, v, \text{say}(v, v, 0)) \end{array} \right) \quad (8)$$

There is still too much water (syntactic noise), and some kittens still aren’t inside boats! Venus! Finish the boats:

$$(v\ x) \left( \begin{array}{l} (say\ x\ z\ 0) \\ | \ (hear\ x\ y\ (say\ y\ x\ (hear\ x\ y\ 0))) \\ | \ (hear\ z\ v\ (say\ v\ v\ 0)) \end{array} \right) \quad (9)$$

Venus! You're not done! Everything must be a kitten or a boat!

```
(channel x
  (par (say x z 0)
    (par (hear x y
      (say y x
        (hear x y 0)))
      (hear z v
        (say v v 0))))))
```

Hooray, all the kittens are safe and dry! But they can't nap, yet. Venus! Rearrange the boats so kittens can chat and then nap!

```
(channel x
  (par (par (say x z 0) ;; Oooh!, x's line up!
    (hear x y
      (say y x
        (hear x y 0))))
    (hear z v
      (say v v 0))))
```

This is great because we have a rule that says whenever a `say` and a `hear` line up their channels, we can rename, substitute and gobble up one `say` and its matching `hear`:

```
(channel x
  (par (par 0
    (say z x
      (hear x z 0)))
    (hear z v
      (say v v 0))))
```

Darn it! Venus! Rearrange the `par`'s again, (it's always OK to do that):

```
(channel x
  (par 0
    (par (say z x (hear x z 0))
      (hear z v (say v v 0)))))
```

Substitute and gobble:

```
(channel x
  (par 0
    (par (hear x z 0)
      (say x x 0))))
```



One more time:

```
(channel x
  (par 0 (par 0 0)))
```

Inside a `par` boat, it doesn't matter whether you write `hear` before `say` or `say` before `hear` — `par` is commutative. Also, because any number of napping kittens in `par`'s is equivalent to a big napping kitten, we see

```
(channel x 0)
```

Finally, because there is nothing to do with channel `x`, The whispering kitten can nap, too.

```
0
```

Thanks, Venus!

## 2.10 Kitten Boat Grammar

This is what Venus-the-boatwright had in mind whilst she built:

$K, L ::=$		
	(nap)	napping kitten      Do nothing; halt.
	(hear <code>x y</code> <code>K</code> )	listening kitten    Listen on channel <code>x</code> for channel <code>y</code> .
	(say <code>x y</code> <code>K</code> )	chatting kitten    Say " <code>y</code> " on channel <code>x</code> ; don't wait.
	(par <code>K L</code> )	two kittens        Run <code>K</code> and <code>L</code> in parallel.
	(channel <code>x</code> <code>K</code> )	whispering kitten   fresh channel name <code>x</code> ; use it in <code>K</code> .
	(repeat <code>K</code> )	mama cat          run copies of <code>K</code> forever.

(10)

### 3 Channels and Names

Our kittens are Kitten One, Kitten Two, and Kitten Three. These aren't names in kitten-speak, not names for channels like `x` and `y`. These are names in boat-speak, just so we don't have to keep writing out the full boats each time.

Let's run some real code! For technical reasons, we need some punctuation — dots and quote marks here and there — when we write out kittens in Clojure. We'll explain later.

#### 3.1 Kit-1

```
(def kit-1
  (say. 'x 'z (nap.)))
```

Notice that when `kit-1` takes a `nap`, she's not saying or hearing anything. *The free names of `(nap)`, the names subject to substitution, are the empty set:*

```
(free-names (nap.))
```

```
#{}
```

In fact, the names that `kit-1` will eventually know about while napping, the *bound names, subject to renaming*, are also the empty set:

```
(bound-names (nap.))
```

```
#{}
```

Before she naps, Kitten One says `z` on `x`, so both those names are free for `kit-1`, meaning she just barks them out. They don't stand for anything else:

```
(free-names kit-1)
```

```
class clojure.lang.Compiler$CompilerException
```

`Kit-1` doesn't wait for any names before she's napping, so her *bound names* are the empty set:

```
(bound-names kit-1)
```

```
class clojure.lang.Compiler$CompilerException
```

### 3.2 Kit-2

Kitten Two listens on  $x$  for bound  $y$ , then says, on whatever  $y$  becomes, “ $x$ ”.

```
(def kit-2
  (hear. 'x 'y
    (say. 'y 'x
      (hear. 'x 'y (nap.)))))
```

We know  $y$  as a free variable — subject to substitution — in her immediately-next say-prefix is going to become  $z$ , but *she* doesn’t know so yet. She only knows that she will *eventually* know what  $y$  stands for, so  $y$  is eventually bound, thus bound. Her bound names include  $y$  now, but  $y$  becomes free, at least in her immediately-next say-prefix, after her first hear-prefix is gobbled up during substitution.  $y$  might be subject to renaming (bound) in later activities if she has a big suffix.

```
(bound-names kit-2)
```

```
class clojure.lang.Compiler$CompilerException
```

Kitten Two’s final activity is to listen on  $x$  for whatever- $y$ -becomes. In that final activity, in isolation, she doesn’t know whether she will ever know  $x$ , so the free variables — subject to substitution — of that final activity had better include  $x$ .

```
(do (def kit-2-final (hear. 'x 'y (nap.)))
    (free-names kit-2-final))
```

```
#{x}
```

By nap-time, she’ll know what  $y$  stands for, but she won’t use it while napping;  $y$  is eventually bound thus bound. We saw it’s not free; check that it’s bound:

```
(bound-names kit-2-final)
```

```
#{y}
```

In her next-to-last activity, which includes her last activity, she will know what  $y$  is, so it is bound:

```
(bound-names
  (say. 'y 'x
    kit-2-final))
```

```
#{y}
```

For her whole sailorly mission-plan, Kitten Two will hear  $y$  on  $x$ , so it’s bound, at least until free — subject to substitution.

```
(bound-names
  (hear. 'x 'y
    (say. 'y 'x
      (hear. 'x 'y (nap.)))))
```

```
#{y}
```

But she'll never use  $x$ . She just passes  $x$  along, so it's free:

```
(free-names kit-2)
```

```
class clojure.lang.Compiler$CompilerException
```

### 3.3 Kit-3

Kitten Three listens on  $z$  for  $v$  — a temporary name — then says “ $v$ ” on  $v$

```
(def kit-3
  (hear. 'z 'v
    (say. 'v 'v (nap.))))
```

Her bound names include  $v$ , at least until it becomes free after renaming (which does nothing, here) and before substitution:

```
(bound-names kit-3)
```

```
class clojure.lang.Compiler$CompilerException
```

Her free names — subject to substitution — include  $z$ :

```
(free-names kit-3)
```

```
class clojure.lang.Compiler$CompilerException
```

Can you write down the free and bound names in her last activity, `(say 'v 'v)`? Here are spoilers:

```
(let [kit-3-last (say. 'v 'v (nap.))]
  (println (free-names kit-3-last))
  (println (bound-names kit-3-last)))
```

```
#{v}
```

```
#{}
```

### 3.4 Kitten Zero — the Whisper Boat

The bound names of Kitten Zero, captain of the Whisper Boat, include all the bound names of the other kittens, so had better be  $x$  for her own,  $y$  from Kitten Two, and  $v$  from kitten Three:

```
(do (def whisper-boat
      (channel. 'x
        (par. kit-1
          (par. kit-2 kit-3))))
    (bound-names whisper-boat))
```

```
class clojure.lang.Compiler$CompilerException
```

Can you writ out her free names? Here is a spoiler:

```
(free-names whisper-boat)
```

```
class java.lang.IllegalArgumentException
```

The free names — subject to substitution — include only  $z$  from both Kitten One, who barks them out in  $(\text{say. } 'x' z)$ , and Kitten Three, who listens on  $z$  for a substitution:

```
# { z }
```

## 4 Change Log

2023-26-Mar :: Current version.

2023-22-Mar :: Start.