The jank programming language

A Clojure dialect on LLVM with gradual typing, a native runtime, and C++ interop

Jeaye Wilkerson | @jeayewilkerson | Github: jeaye | jank: website

The broad strokes

- Interactive programming with Clojure (live demo with HN API)
- Clojure's interactive compilation model
- The road to interactive programming with jank

A lot of Clojure!

Live demo x2!

Interactive Programming with Clojure

Hopefully a practical explanation of why Lispers talk so darn much about REPLs

What's a REPL to you?

On a surface level, REPLs are handy. But, to Lispers, they're **much more** than a readline loop. Here's what most people think when they hear REPL.

JavaScript

```
❯ node
> 1 + 2
3
>
```

Ruby

```
❯ irb
irb(main):001:0> 1 + 2
=> 3
irb(main):002:0>
```

Clojure

```
❯ clj
user=> (+ 1 2)
3
user=>
```

With Cling, we might also think of Jupyter!

This page is intentionally left blank.

Clojure's Interactive Compilation Model

… and thus jank's …

How does Clojure work?

Let's compile something!

```
(ns hn.query
  (:require [jsonista.core :as j]))

; Top-level effect!
(def config (-> (slurp "resources/config.json")
                j/read-value))
(def base-url (:base-url config))
(def post-id (:post-id config))

(defn find-word [comments word]
  (filter (fn [comm]
            (clojure.string/includes? (:text comm "") word))
          comments))
```

How does Clojure work?

Let's compile something!

```
(ns hn.query
  (:require [jsonista.core :as j]))
```

```
{:namespaces {'hn.query {}
              'jsonista.core {:vars {'read-value …}}}
 :namespace-aliases {'j 'jsonista.core}}
```

How does Clojure work?

Let's compile something!

```
(ns hn.query
  (:require [jsonista.core :as j]))

; Top-level effect!
(def config (-> (slurp "resources/config.json")
                j/read-value))
(def base-url (:base-url config))
(def post-id (:post-id config))
```

```
{:namespaces {'hn.query {:vars {'config …
                                'base-url …
                                'post-id …}}
              'jsonista.core {:vars {'read-value …}}}
 :namespace-aliases {'j 'jsonista.core}}
```

How does Clojure work?

Let's compile something!

```
(ns hn.query
  (:require [jsonista.core :as j]))

; Top-level effect!
(def config (-> (slurp "resources/config.json")
                j/read-value))
(def base-url (:base-url config))
(def post-id (:post-id config))

(defn find-word [comments word]
  …)
```

```
{:namespaces {'hn.query {:vars {'config …
                                'base-url …
                                'post-id …
                                'find-word …}}
              'jsonista.core {:vars {'read-value …}}}
 :namespace-aliases {'j 'jsonista.core}}
```

Cool, so we evaluated that.

Wait, I thought we were compiling…

What's the difference?

When compiling:

- Only top-level forms are evaluated
- No entrypoint is called
- Generated bytecode for each file is persisted to disk

Otherwise, it's indistinguishable from evaluating.

Turning our gaze to jank

- Started research in 2015; has been through many iterations
- jank is now a Clojure dialect with LLVM as its host (via Cling!)
- On top of that, it aims to provide gradual, structural typing (not covering this today)

The focus today: how do we make jank as interactive as Clojure?

- The road to interactive programming with jank
  - Step 1: Codegen
  - Step 2: JIT compilation
  - Step 3: REPL support

https://jank-lang.org/

Step 1: Codegen with jank

Turning jank into C++

What does the codegen look like?

```
(def a 222)
(println a)
```

```
namespace jank::generated {
  struct gen3 : jank::runtime::behavior::callable {



    gen3(jank::runtime::context &rt_ctx)



    { }

    jank::runtime::object_ptr call() const override {


    }
  };
}
```

What does the codegen look like?

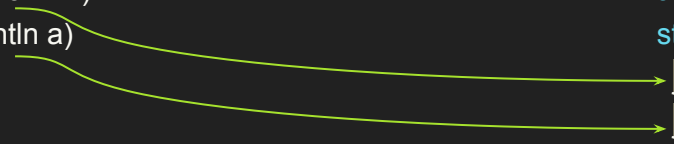
```
(def a 222)
(println a)
```

1. Lift constants

```
namespace jank::generated {
  struct gen3 : jank::runtime::behavior::callable {

    jank::runtime::object_ptr const const2;

    gen3(jank::runtime::context &rt_ctx)

        const2{jank::runtime::make_box<jank::runtime::obj::integer>(222)}
    { }

    jank::runtime::object_ptr call() const override {


    }
  };
}
```

What does the codegen look like?

```
(def a 222)
(println a)
```



```
namespace jank::generated {
  struct gen3 : jank::runtime::behavior::callable {
    jank::runtime::var_ptr const var1;
    jank::runtime::var_ptr const var4;
    jank::runtime::object_ptr const const2;

    gen3(jank::runtime::context &rt_ctx)
      : var1{rt_ctx.intern_var("user", "a").expect_ok()},
        var4{rt_ctx.intern_var("clojure.core", "println").expect_ok()},
        const2{jank::runtime::make_box<jank::runtime::obj::integer>(222)}
    { }

    jank::runtime::object_ptr call() const override {


    }
  };
}
```

1. Lift constants
2. Lift vars

What does the codegen look like?

```
(def a 222)
(println a)
```

1. Lift constants
2. Lift vars
3. Add remaining code

```
namespace jank::generated {
  struct gen3 : jank::runtime::behavior::callable {
    jank::runtime::var_ptr const var1;
    jank::runtime::var_ptr const var4;
    jank::runtime::object_ptr const const2;

    gen3(jank::runtime::context &rt_ctx)
        : var1{rt_ctx.intern_var("user", "a").expect_ok()},
          var4{rt_ctx.intern_var("clojure.core", "println").expect_ok()},
          const2{jank::runtime::make_box<jank::runtime::obj::integer>(222)}
    { }

    jank::runtime::object_ptr call() const override {
      var1->set_root(const2);
      return var4->get_root()->as_callable()->call(var1->get_root());
    }
  };
}
```

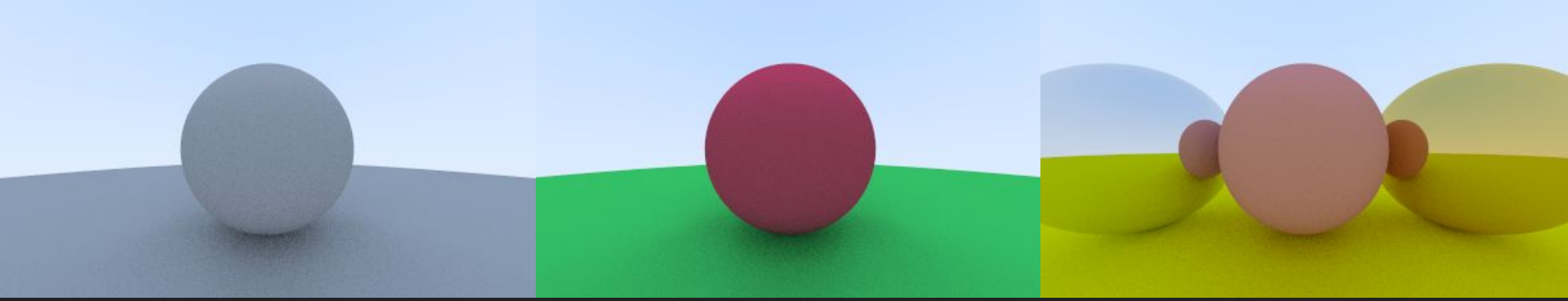
Step 2: JIT compilation with jank

Turning C++ into machine code

This page is intentionally left blank.

Step 3: REPL support for jank

The final goal



Q&A

Thanks for having me!

https://jank-lang.org/