Swearjure

Gary Fredericks

```
(#((% (+(*))) %)
 [(+ (*)(*)(*)(*)(*)(*)(*)) ; input == 7
  #(({(+) (% (+(*)(*)))} (% (+))
     (% (+(*)(*)(*)))) %)
  #({} % (+(*)))
  #(* ((% (+(*)))
       [(- (% (+)) (+(*)))
        (% (+(*)))
        (% (+(*)(*)))
        (% (+(*)(*)(*)))])
      (% (+)))]) ; => 5040
```

Primary Researchers

Tim McCormack

Jean Niklas L'orange

Constants

Positive Integers

```
(*) ;; => 1

(+ (*) (*)) ;; => 2

(* (+ (*)(*))
   (+ (*)(*)(*))
   (+ (*)(*)(*)(*)(*)(*)(*)))
;; => 42
```

Negative Integers and Ratios

```
(- (+ (*) (*) (*))) ;; => -3

(/ (+ (*) (*))
   (+ (*) (*) (*))) ;; => 2/3
```

Floats?

No known solution.

`true` / `false` / `nil`

```
(= :+)      ;; => true
(= :+ :-)   ;; => false
({} :+)     ;; => nil
```

Data Structures

```
1:  (def x 42) (def xs '(0 1 2))
2:  ;; construct a list:
3:  `(~x ~@xs) ;; => (42 0 1 2)
4:  ;; concat two lists:
5:  `(~@xs ~@xs) ;; => (0 1 2 0 1 2)
6:  ;; convert to a vector:
7:  `[~@xs] ;; => [0 1 2]
8:  ;; nth
9:  (`[~@xs] (+ (*) (*))) ;; => 2
```

Functions

Function Literal Toolbox

`#(...)`

- Can't use `fn`, `defn`, etc
- Can only refer to `%` and `%&`
  - (not `%2`, `%3`, etc)
- Can't nest expressions

The Literal Problem

How can a function return a data structure literal?

```
1:  #([1 2 %])           ;; doesn't work
2:
3:  (fn [%] [1 2 %])     ;; works, but illegal
4:
5:  #(identity [1 2 %]) ;; works, but illegal
6:
7:  #([[1 2 %]] (+))     ;; works, and legal!
```

Common `clojure.core` Functions

```
 1:  (def inc #(+ % (*)))
 2:  (def dec #(- % (*)))
 3:  (def identity #([%] (+)))
 4:
 5:  (def boolean #({false false nil false} % true))
 6:  (def boolean #({(= :+ :-)
 7:                  (= :+ :-),
 8:                  (:+ :-)
 9:                  (= :+ :-)}
10:                 % (= :+)))
```

Control Flow, Recursion, etc.

Use a function table

```
 1:  ((fn [ft n]
 2:     ((ft :fact) ft n)) ; Call factorial with all fns and n
 3:
 4:   {:fact (fn [ft n]
 5:             (let [f-to-call ({0 :is-zero} n :recurse)]
 6:               ((ft f-to-call) ft n)))
 7:    :is-zero (fn [ft n] 1) ; return 1
 8:    :recurse (fn [ft n]    ; recurse by calling :fact
 9:               (* n ((ft :fact) ft (- n 1))))}
10:    6)
```

Control Flow, Recursion, etc. 2

Simulating `if`

```
 1:  ;; Equivalent to #(if (< % 5) (+ % 7) (- % 5))
 2:
 3:  ((fn [ft n]
 4:     ((ft :main) ft n))
 5:
 6:   {:main (fn [ft n] ((ft ({false :falsy true :truey} (< n 5)))
 7:                      ft n))
 8:    :truey (fn [ft n] (+ n 7))
 9:    :falsy (fn [ft n] (- n 5))}
10:    6) ;; => 1
```

Control Flow, Recursion, etc. 3

The function table tactic eats the stack.

Curiosities

hash-map

```
;; NB: Does not accept 0 args
(def hash-map
  #(`[{~% ~@%&}] (+)))
```

Getting a Gensym

Always returns the same symbol

```
(#('%& '%& '%&))
;; => rest__1206#
```

rest

```
(#(((% (+(*))) (+)) %)
 [[:a :b :c :d :e] ;; input
  [#(({[] ((% (+(*))) (+(*)))}
      (% (+))
      ((% (+(*))) (+(*)(*))))
    [(% (+)) (% (+(*)))])
   #({} %& [])
   #(((% (+(*))) (+(*)(*)(*)))
     [(% (+)) (% (+(*))) [(+(*)) []]])
   #(({(% (+)) ((% (+(*))) (+(*)(*)(*)(*)))}
      `[~((% (+)) (+))
        ~@((% (+(*)(*))) (+(*)))]
      ((% (+(*))) (+(*)(*)(*)(*)(*))))
     [(% (+)) (% (+(*))) (% (+(*)(*)))])
   #((% (+(*)(*))) (+(*)))
   #(((% (+(*))) (+(*)(*)(*)))
     [(% (+)) (% (+(*))) [(+ (+(*)) ((% (+(*)(*))) (+)))
                    `[~@((% (+(*)(*))) (+(*)))
                      ~((% (+)) ((% (+(*)(*))) (+)))]]])]])
;; => [:b :c :d :e]
```

Completeness

Turing Complete?

Reduction to lambda calculus is difficult because we effectively don't have closures.

Turing Complete: The SKI Calculus

This slide has been left as an exercise for the reader.

Clojure Complete?

Hardly. Missing:

- Expressions for:
  - Arbitrary strings, symbols, keywords
  - Floating point numbers
- Interop of any sort
- `partial` and `apply`
- `def`, `recur`, `throw`, `catch`, …
- Nearly everything really

Swearjure

"Inescapeably useless, surprisingly powerful."

Future Work

- Actually implement SKI Calculus
- Efficient algorithm for finding smallest representation of an integer
- Smallest alphanumeric preamble that would make Swearjure clojure-complete