

# Shen: A Sufficiently Advanced Lisp

Aditya Siram (@deech), Intoximeters, Inc.

September 18, 2014

# Outline

## 1 Shen

- No arguments ...

```
(define hello -> (output "Hello world"))  
(hello) => "Hello world"
```

- Single argument ...

```
(define hello  
  X -> (output "Hello ~A" X))  
(hello "Strange Loop") => "Hello Strange Loop"
```

- Multiple Arguments

```
(define hello  
  X Y -> (output "Hello ~A and ~A" X Y))  
(hello "Strange Loop" "Lambda Lounge")  
  => "Hello Strange Loop and Lambda Lounge"
```

# Typed Functions

- Turn on typechecking, type signatures are required ...

```
(tc +)
```

- Simple Types

```
(define is-positive  
  { number --> boolean }  
  X -> (and (number? X) (>= X 0)))  
(is-positive -1) => false : boolean
```

# Typed Functions

- Polymorphic Types

```
(define map
  { (A --> B) --> (list A) --> (list B) }
  F [] -> []
  F [ X | XS ] -> [(F X) | (map F XS)])
```

```
(map (function (+ 1)) [1 2 3]) =>
  [2 3 4] : (list number)
```

- Note the currying above ...

# Untyped Functions

- Turn off typechecking, type signatures are optional ...

```
(tc -)
(define is-positive
  X -> (and (number? X) (>= X 0)))
(define map
  { (A --> B) --> (list A) --> (list B) }
  F [] -> []
  F [ X | XS ] -> [(F X) | (map F XS)])
```

```
(is-positive 10) => true
(map (/ . X (+ X 1)) [1 2 3]) => [2 3 4]
```

- Notice no types in results & the alternate lambda syntax

- Optional typechecking
- Very flexible type specification
- (Prolog) Pattern matching

```
(define same  
  X X -> true  
  _ _ -> false)
```

- Full Prolog engine
- Full parsing engine
- Decompiles to KLambda (Scheme-like)
- Your Shen = A Scheme interpreter + 45 primitive functions
  - if, and, or, cons ...
- "A unifying meta-language ..."



# Web Application Demo

- Backend in Ruby (Sinatra)
- Frontend in JavaScript (Qooxdoo)
- Common code in Shen
- This is the important bit.

- Simple Login Screen
- Check form inputs using identical Shen code.

- Validation data-structure

```
(set *model*  
  (@v (@p "username"  
           (function (parses (function <alphas>))))  
    (@p "password"  
        (function parse-password))  
    <>))
```

- This is a global variable!
- And a function that runs the validator ...

```
(define run-validator  
  nil  Key -> false  
  Value Key ->  
    (let Validator (lookup (value *model*) Key)  
      (Validator Value)))
```

- The lookup function ...

```
(define lookup
  (@v (@p Key Validator) _) Key -> Validator
  (@v _ PS) Key -> (lookup PS Key)
  <> Key -> (error "No validator for ~A." Key))
```

- Notice use of unification!

- The username parser is standard

```
(defcc <alphas>
  <alpha> <alphas> := [<alpha> | <alphas> ];
  <alpha> := [<alpha>])

(compile (function <alphas>)
  (string->bytes "abcdef")) =>
  [97 98 99 100 101 102]
```

# Check Parser Output

- Check the parser runs

```
(define parses
  Parser Input ->
  (trap-error
    (do
      (compile
        (function Parser)
        (shen.string->bytes Input))
      true)
    (/. E false)))
```

- 'do' macro for imperative programming
- 'trap-error' for exceptions

- The password parser is more interesting

```
(defcc <password>
  shen.<digit> <password>
    := [(function add-digit) | <password>];
  X <password>
    := [(function add-special) | <password>]
      where (element? X (shen.string->bytes "_!"));
  shen.<alpha> <password>
    := [(function add-alpha) | <password>];
  <e> := []
)
```

- User input is translated into function!

- Parsing forms a chain of functions

```
"Hi123!_"
```

```
=>
```

```
[
```

```
  (function add-alpha)
```

```
  (function add-alpha)
```

```
  (function add-digit)
```

```
  (function add-digit)
```

```
  (function add-digit)
```

```
  (function add-special)
```

```
  (function add-special)
```

```
]
```



- Add functions

```
(define add-alpha
  (@p Alpha Digit Special) ->
  (@p (+ Alpha 1) Digit Special))
(define add-digit
  (@p Alpha Digit Special) ->
  (@p Alpha (+ 1 Digit) Special))
(define add-special
  (@p Alpha Digit Special) ->
  (@p Alpha Digit (+ 1 Special)))
```

- Compose them!

```
(shen.compose  
  [  
    (function add-alpha)  
    (function add-digit)  
    ...  
  ]  
  (@p 0 0 0))
```

```
"Hi123_!" => (@p 2 3 2)
```

```
(define validate-password  
  (@p Alpha Digit Special) ->  
    (and (= Alpha Digit)  
          (not (= Special 0))))
```

- Ruby/JavaScript hookup

```
Shen.call_by_name(      shen.send(  
  "run-validator",      "run-validator",  
  [value, "password"]  password, "password"  
)                      )
```

- Increment a counter over HTTP
- Make it type safe.
- Shen functions ...

```
(define string->positive-number
  { (readable string) --> positive-number }
  X -> (make-positive (toNumber X)))

(define add-positive
  { positive-number --> positive-number -->
    positive-number }
  X Y -> (+ X Y))

(define is-positive
  { number --> boolean }
  X -> (and (number? X) (>= X 0)))
```

- Type specification

```
(datatype positive-number
  X : positive-number;
  Y : positive-number;

  -----
  (+ X Y) : positive-number;

  ...
)
```

- To show '(+ X Y)' inhabits 'positive-number'
  - Show 'X' inhabits 'positive-number' **and**
  - Show 'Y' inhabits 'positive-number'

- So how do you make a 'positive-number'? Verified types ...

```
(datatype positive-number
```

```
...
```

```
-----  
(is-positive X) : verified >> X : positive-number;  
)
```

- Note nothing above the bar
- If '(is-positive X)' is in the list of assumptions,
  - 'X : positive-number' is proven!

- Basically include a '(is-positive X)' guard ...

```
(define make-positive
  { number --> positive-number }
  X -> X where (is-positive X)
  X -> (error "~A is not a positive number" X))
```

- Typechecker 'sees' the (is-positive X) and rubber-stamps 'make-positive'.

- Parsing a string to a number using Shen's own parser ...

```
(defcc <digits>
  <digit> <digits> := [<digit> | <digits> ];
  <digit> := [<digit>])

(defcc <trimmed-digits>
  <whitespaces> <digits> <whitespaces> := <digits>;
  <digits> <whitespaces> := <digits>;
  <whitespaces> <digits> := <digits>;
  <digits> := <digits>)

(compile (function <trimmed-digits>)
  (string->bytes " 123 ")) =>
  [1 2 3]
```



- Run the parser

```
(define number-string-p
  X -> (let Result
        (trap-error
         (compile
          (function <trimmed-digits>
            (shen.string->bytes X))
            (/. E (do (pr E) (fail))))
         (not (= Result (fail))))))

(declare number-string-p [string --> boolean])
```

- Note the type signature after the fact.
- Tell the typechecker the type of dynamic code!

- Now add the final rule to the datatype ...

```
(datatype positive-number
  ....

  -----
  (number-string-p X) : verified >>
  X : (readable string);
)

(define make-readable
  { string --> (readable string) }
  X -> X where (number-string-p X)
  X -> (error "~A is not a number" X))
```

- Everything along the chain is typechecked!

# JSON Typing

- JSON -> Internal Representation

```
{                                [object
  auth:                          (@p "auth"
    {                             [object
      user: "someuser",          (@p "user" "someuser")
      pass: "somepass"           (@p "pass" "somepass")
    }                             ]])
  first: "Joe",                  (@p "first" "Joe")
  last: "User",                  (@p "last" "User")
  phone: [314 555 5555]          (@p "phone" [314 555 5555])
}
```

- Internal representation -> type

[object	(object
(@p "auth"	((kv (auth *
[object	(object
(@p "user" "someuser")	((kv (user * string))
(@p "pass" "somepass"))]	(kv (pass * string))))))
(@p "first" "Joe")	((kv (first * string))
(@p "last" "User")	((kv (last * string))
(@p "phone" [314 555 5555]))]	(kv (phone * (list number))))))

- Iterating over '[object ...]'

```
(datatype object-iterator
  ...
  let Separated (shen.cons_form
                  (separate (shen.decons XS)))
  Separated : KVS;

  -----
  [object | XS] : (object KVS);
)

(define separate
  [] -> []
  [X] -> [end X]
  [X | XS] -> [X , | (separate XS)]
)

[object a b c] => [ a , b , end c]
```

# JSON Typing

- Iterating over '[object ...]'
- ```
(datatype object-iterator  
...)
```

```
X : KV;
```

```
XS : KVS;
```

```
-----  
[X , | XS] : (KV KVS);
```

```
X : KV;
```

```
-----  
[end X] : KV;
```

```
)
```

```
[object a b c] => (object (a-type (b-type (c-type))))
```

- Key value datatype:

```
(datatype kv
```

```
  X : (A * B);
```

```
-----
```

```
[kv X] : (kv (A * B));
```

```
)
```

```
[kv (@p "hello" 1)] => (kv (string * number))
```

- Datatype for 'user':

```
(datatype user-type
  X : (string * B);
  X : (user * B) >> P;

  -----
  (= user (string->symbol (fst X))) : verified >> P;)
```

- Function to create 'user':

```
(define make-user
  { (string * A) --> (kv (user * A)) }
  X -> [kv X] where (= user (string->symbol (fst X))))
```



# JSON Typing

- Generate the 'user' key types & functions:

```
(define key-type
  Key -> (string->symbol (make-string "~A-type" Key)))
(define key-maker
  Key -> (string->symbol (make-string "make-~A" Key)))
```

```
(define register-key
  Key ->
    (let Type (key-type Key)
      Func (key-maker Key)
      ...
    )
  )
```

```
(register-key "user"): Type: user-type, Func: make-user
```

# JSON Typing

- Generate the 'user' key types & functions:

```
(define register-key
  Key ->
  (let ...
    (do
      (eval
        [datatype Type
         X : [string * B];
         X : [Key * B] >> P;

         -----
         [= Key [string->symbol [fst X]]] : verified >> P;]
      )
    ...)))
```

# JSON Typing

- Generate the 'user' key types & functions:

```
(define register-key
  Key ->
    (let ...
      (do
        ...
        (eval
          [define Func
            FArg ->
              (shen.cons_form [kv FArg])
              where [= Key [string->symbol [fst FArg]]]]
          ])
      ...)))
```

- Generate the 'user' key types & functions:

```
(define register-key
  Key ->
  (let ...
    (do
      ...
      (declare Func [[string * A] --> [kv [Key * A]]])
      nil)))
```

# JSON Typing

- Generate all the keys

```
[object
  (@p "auth"
    [object
      (@p "user" "someuser")
      (@p "pass" "somepass")])
  (@p "first" "Joe")
  (@p "last" "User")
  (@p "phone" [314 555 5555])]
```

```
(map
  (function register-key)
  [auth user pass first last phone])
```

# JSON Typing

- Need to translate

```
[object  
  (@p "auth"  
    [object  
      (@p "user" "someuser")  
      (@p "pass" "somepass")]))]
```

=>

```
[object  
  (make-auth (@p "auth"  
    [object  
      (make-user (@p "user" "someuser"))  
      (make-pass (@p "pass" "somepass"))])))]
```

# JSON Typing

- Need a macro!

```
(defmacro my-awesome-obj-macro
  [obj-macro [cons object Pairs]] ->
  (shen.cons_form [object |
    (map
      (function obj-macro-helper)
      (shen.decons Pairs))]))
```

```
(define obj-macro-helper
  [@p X Y] ->
  [(key-maker X)
   [@p X (obj-macro-helper Y)]]
  [object PS] -> (obj-macro [object PS])
  XS -> (map (function obj-macro-helper) XS)
        where (cons? XS)
  X -> X)
```

- Invoking

```
(obj-macro [object (@p "auth" ...) ..])
```



# JSON Typing

- Type check at run-time!

```
(define user-input-key-value ->
  (let Key (input)
    Value (input)
    Pair (@p (make-string "~A" Key) Value)
    Form [object | [(helper Pair)]]
    (shen.typecheck (shen.cons_form Form) A)))

> (user-input-key-value)
phone
[314 555 5555]
=>
[object [kv [phone * [list number]]]]
```

- The Shen website - <http://www.shenlanguage.org/>
- The Book Of Shen - <http://www.shenlanguage.org/tbos.html>
- Shen Google Group -  
<https://groups.google.com/group/qilang?hl=en>