Part I

Values

A ***value*** is the result of an ***expression***

- Expression: `1 + 2`
- Value: `3`

A value can be be
the argument to a function,
the right-hand side of a `let`,
...

Functions as Values?

Is a function a value in Moe?

No

You can define a function

```
fun double(x): x + x
```

You can call a function

```
double(10)
```

You *cannot* use a function name without calling it

You *cannot* pass a function to another function

Functions as Values?

Is a function a value in Shplait?

Yes

An expression can produce a function result

```
fun double(x): x + x
double
```

```
[min, max]
```

```
fun (x): x + x
```

You can pass a function to a function:

```
map(fun (x): x + x,
    [1, 2, 3])
```

Why Functions as Values

Abstraction is easier with functions as values

- `filter`, `map`, `foldl`, etc.

Separate `fun` definition form becomes unnecessary

```
fun f(x): 1 + x
f(10)
```

⇒

```
let f = (fun (x): 1 + x):
  f(10)
```

Why Functions as Values

Abstraction is easier with functions as values

- `filter`, `map`, `foldl`, etc.

Separate `fun` definition form becomes unnecessary

```
fun f(x): 1 + x
f(10)
```

⇒

Historical name: **`lambda`** or **λ**

```
let f = (fun (x): 1 + x):
  f(10)
```

Part 2

New Moe Grammar, Almost

```
<Exp>  ::=  <Int>
        |   <Symbol>
        |   <Exp> + <Exp>
        |   <Exp> * <Exp>
        |   let <Symbol> = <Exp>: <Exp>
        |   <Symbol>(<Exp>)                *
        |   fun (<Symbol>): <Exp>          NEW
```

Evaluation

`10` ⇒ `10`

`y` ⇒ *free variable*

`1 + 2` ⇒ `3`

`2 * 3` ⇒ `6`

`let x = 7: x + 2` ⇒ `7 + 2` ⇒ `9`

`fun (x): 1 + x` ⇒ `fun (x): 1 + x`

Result is not always a number!

~~`interp :: (Exp, ....) -> Int`~~

`interp :: (Exp, ....) -> Value`

Evaluation

`10` ⇒ `10`

`y` ⇒ *free variable*

`1 + 2` ⇒ `3`

`2 * 3` ⇒ `6`

`let x = 7: x + 2` ⇒ `7 + 2` ⇒ `9`

`fun (x): 1 + x` ⇒ `fun (x): 1 + x`

`let y = 10: fun (x): y + x`
⇒ `fun (x): 10 + x`

`let f = (fun (x): 1 + x): f(3)`
⇒ `(fun (x): 1 + x)(3)`

Doesn't match the grammar for **`<Exp>`**

New Moe Grammar

```
<Exp>  ::=  <Int>
        |   <Symbol>
        |   <Exp> + <Exp>
        |   <Exp> * <Exp>
        |   let <Symbol> = <Exp>: <Exp>
~~      |   <Symbol>(<Exp>)~~
        |   fun (<Symbol>): <Exp>                NEW
        |   <Exp>(<Exp>)                         NEW
```

Evaluation

`let f = (fun (x): 1 + x): f(3)`
⇒ `(fun (x): 1 + x)(3)`
⇒ `1 + 3` ⇒ `4`

`(fun (x): 1 + x)(3)` ⇒ `1 + 3` ⇒ `4`

`1(2)` ⇒ *not a function*

`1 + (fun (x): 10)` ⇒ *not a number*

Part 3

Expression Datatype

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
        r :: Exp)
| multE(l :: Exp,
        r :: Exp)
| letE(n :: Symbol,
       rhs :: Exp,
       body :: Exp)
| funE(n :: Symbol,
       body :: Exp)
| appE(fn :: Exp,
       arg :: Exp)
```

```
check: parse('fun (x): x + 1')
       ~is funE(#'x, plusE(idE(#'x), intE(1)))
```

Expression Datatype

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
        r :: Exp)
| multE(l :: Exp,
        r :: Exp)
| letE(n :: Symbol,
       rhs :: Exp,
       body :: Exp)
| funE(n :: Symbol,
       body :: Exp)
| appE(fn :: Exp,
       arg :: Exp)
```

```
check: parse('f(10)')
       ~is appE(idE(#'f), intE(10))
```

Expression Datatype

```
type Exp
| intE(n :: Int)
| idE(s :: Symbol)
| plusE(l :: Exp,
        r :: Exp)
| multE(l :: Exp,
        r :: Exp)
| letE(n :: Symbol,
       rhs :: Exp,
       body :: Exp)
| funE(n :: Symbol,
       body :: Exp)
| appE(fn :: Exp,
       arg :: Exp)
```

```
check: parse('(fun (x): x + 1)(10)')
         ~is appE(funE(#'x, plusE(idE(#'x), intE(1))),
                  intE(10))
```

Part 4

Functions with Substitutions

```
interp( let y = 10:
          fun (x): y + x )
```

Functions with Substitutions

```
interp( let y = 10:
          fun (x): y + x )
```

Functions with Substitutions

```
interp(let y = 10:
         fun (x): y + x)
```

⇒

```
fun (x): 10 + x
```

Functions with Substitutions

```
interp(let y = 10: fun (x): y + x)
```

⇒

```
fun (x): 10 + x
```

Functions with Deferred Substitution

```
interp(let y = 10: fun (x): y + x)
```

⇒

y = 10

```
interp(fun (x): y + x)
```

Functions with Deferred Substitution

```
interp( (let y = 10: fun (x): y + x)(let y = 7: y) )
```

Argument expression:

```
interp( let y = 7: y )
```

⇒

```
        y = 7
interp( y )  ⇒  7
```

Function expression:

```
interp( let y = 10: fun (x): y + x )
```

⇒

```
                          y = 10
interp( fun (x): y + x )  ⇒  ?
```

Functions with Deferred Substitution

```
interp((let y = 10: fun (x): y + x)(let y = 7: y))
```

Argument expression:

```
interp(let y = 7: y)
```

⇒

```
       y = 7
interp(y)  ⇒  7
```

Function expression:

```
interp(let y = 10: fun (x): y + x)
```

⇒

```
                    y = 10
interp(fun (x): y + x)  ⇒  ?
```

A ***closure*** combines an expression with an environment

Representing Values

```
type Value
| intV(n :: Int)
| closV(arg :: Symbol,
        body :: Exp,
        env :: Env)

type Binding
| bind(name :: Symbol,
       val :: Value)
```

```
check: interp(let y = 10: fun (x): y + x,
              mt_env)
         ~is closV(#'x, y + x,
                   extend_env(bind(#'y, intV(10)),
                              mt_env))
```

Continuing Evaluation

```
                     y = intV(7)
Argument: interp( y )
            ⇒ intV(7)
                                        y = intV(10)
Function: interp( fun (x): y + x )
           ⇒ closV(#'x, y + x ,
                     extend_env(bind(#'y, intV(10)),
                                mt_env))
```

To apply, interpret the function body with the given argument:

```
                     x = intV(7)   y = intV(10)
      interp( y + x )
```

Part 5

Interpreter

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | intE(n): intV(n)
  | idE(s): lookup(s, env)
  | plusE(l, r): num_plus(interp(l, env), interp(r, env))
  | multE(l, r): ....
  | letE(n, rhs, body):
      ....
  | funE(n, body): ....
  | appE(fn, arg):
      ....
```

Add and Multiply

```
fun num_plus(l :: Value, r :: Value) :: Value:
  cond
  | l is_a intV && r is_a intV:
      intV(intV.n(l) + intV.n(r))
  | ~else:
      error(#'interp, "not a number")

fun num_mult(l :: Value, r :: Value) :: Value:
  cond
  | l is_a intV && r is_a intV:
      intV(intV.n(l) * intV.n(r))
  | ~else:
      error(#'interp, "not a number")
```

Add and Multiply

```
fun num_op(l, r, op):
  cond
  | l is_a intV && r is_a intV:
      intV(op(intV.n(l), intV.n(r)))
  | ~else:
      error(#'interp, "not a number")

fun num_plus(l :: Value, r :: Value) :: Value:
  num_op(l, r, fun (ln, rn): ln + rn)

fun num_mult(l :: Value, r :: Value) :: Value:
  num_op(l, r, fun (ln, rn): ln * rn)
```

Interpreter

```
fun interp(a :: Exp, env :: Env) :: Value:
  match a
  | intE(n): intV(n)
  | idE(s): lookup(s, env)
  | plusE(l, r): num_plus(interp(l, env), interp(r, env))
  | multE(l, r): num_mult(interp(l, env), interp(r, env))
  | letE(n, rhs, body):
      interp(body, extend_env(bind(n, interp(rhs, env)),
                              env))
  | funE(n, body): closV(n, body, env)
  | appE(fn, arg):
      match interp(fn, env)
      | closV(n, body, c_env):
          interp(body,
                 extend_env(bind(n, interp(arg, env)),
                            c_env))
      | ~else: error(#'interp, "not a function")
```