Part 1

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 Curly?

No

You can define a function

```
{define {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 Plait?

Yes

An expression can produce a function result

```
 (define (double x) (+ x x))
double

          (list + * - /)

      (lambda (x) (+ x x))
```

You can pass a function to a function:

```
(map (lambda (x) (+ x x))
     (list 1 2 3))
```

Why Functions as Values

Abstraction is easier with functions as values

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

Separate **`define`** form becomes unnecessary

```
{define {f x} {+ 1 x}}
{f 10}
```

⇒

```
{let {[f {lambda {x} {+ 1 x}}]}
  {f 10}}
```

Part 2

New Curly Grammar, Almost

```
<Exp>  ::=  <Number>
         |  <Symbol>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
         |  {let {[<Symbol> <Exp>]} <Exp>}
         |  {<Symbol> <Exp>}                         *
         |  {lambda {<Symbol>} <Exp>}                NEW
```

Evaluation

`10` ⇒ `10`

`y` ⇒ *free variable*

`{+ 1 2}` ⇒ `3`

`{* 2 3}` ⇒ `6`

`{let {[x 7]} {+ x 2}}` ⇒ `{+ 7 2}` ⇒ `9`

`{lambda {x} {+ 1 x}}` ⇒ `{lambda {x} {+ 1 x}}`

Result is not always a number!

~~; interp Exp ... -> Number~~

```
; interp Exp ... -> Value
```

Evaluation

```
10  ⇒  10

y  ⇒  free variable

{+ 1 2}  ⇒  3

{* 2 3}  ⇒  6

{let {[x 7]} {+ x 2}}  ⇒  {+ 7 2}  ⇒  9

{lambda {x} {+ 1 x}}  ⇒  {lambda {x} {+ 1 x}}

{let {[y 10]} {lambda {x} {+ y x}}}
 ⇒  {lambda {x} {+ 10 x}}

{let {[f {lambda {x} {+ 1 x}}]} {f 3}}
 ⇒  {{lambda {x} {+ 1 x}} 3}
```

Doesn’t match the grammar for **<Exp>**

New Curly Grammar

```
<Exp>  ::=  <Number>
        |   <Symbol>
        |   {+ <Exp> <Exp>}
        |   {* <Exp> <Exp>}
        |   {let {[<Symbol> <Exp>]} <Exp>}
~~      |   {<Symbol> <Exp>}~~
        |   {lambda {<Symbol>} <Exp>}         NEW
        |   {<Exp> <Exp>}                     NEW
```

Evaluation

```
{let {[f {lambda {x} {+ 1 x}}]} {f 3}}
 ⇒  {{lambda {x} {+ 1 x}} 3}
 ⇒  {+ 1 3}  ⇒  4
```

```
{{lambda {x} {+ 1 x}} 3}  ⇒  {+ 1 3}
 ⇒  4
```

`{1 2}` ⇒ *not a function*

`{+ 1 {lambda {x} 10}}` ⇒ *not a number*

Part 3

Expression Datatype

```
(define-type Exp
  (numE [n : Number])
  (idE [s : Symbol])
  (plusE [l : Exp]
         [r : Exp])
  (multE [l : Exp]
         [r : Exp])
  (letE [n : Symbol]
        [rhs : Exp]
        [body : Exp])
  (lamE [n : Symbol]
        [body : Exp])
  (appE [fun : Exp]
        [arg : Exp]))
```

```
(test (parse `{lambda {x} {+ x 1}})
      (lamE 'x (plusE (idE 'x) (numE 1))))
```

Expression Datatype

```
(define-type Exp
  (numE [n : Number])
  (idE [s : Symbol])
  (plusE [l : Exp]
         [r : Exp])
  (multE [l : Exp]
         [r : Exp])
  (letE [n : Symbol]
        [rhs : Exp]
        [body : Exp])
  (lamE [n : Symbol]
        [body : Exp])
  (appE [fun : Exp]
        [arg : Exp]))
```

```
(test (parse `{{lambda {x} {+ x 1}} 10})
      (appE (lamE 'x (plusE (idE 'x) (numE 1)))
            (numE 10)))
```

Part 4

Functions with Substitutions

```
(interp {let {[y 10]}
          {lambda {x} {+ y x}}})
```

Functions with Substitutions

```
(interp {let {[y 10]}
           {lambda {x} {+ y x}}} )
```

Functions with Substitutions

```
(interp {let {[y 10]}
           {lambda {x} {+ y x}}})
```

⇒

```
{lambda {x} {+ 10 x}}
```

Functions with Substitutions

```
(interp {let {[y 10]} {lambda {x} {+ y x}}})
```

⇒

```
{lambda {x} {+ 10 x}}
```

Functions with Deferred Substitution

```
(interp {let {[y 10]} {lambda {x} {+ y x}}})
```

⇒

y = 10

```
(interp {lambda {x} {+ y x}})
```

Functions with Deferred Substitution

```
(interp {{let {[y 10]} {lambda {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]} {lambda {x} {+ y x}}})

⇒
                               y = 10
(interp {lambda {x} {+ y x}})  ⇒  ?
```

Functions with Deferred Substitution

```
(interp {{let {[y 10]} {lambda {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]} {lambda {x} {+ y x}}})
```

⇒

```
                                y = 10
(interp {lambda {x} {+ y x}})  ⇒  ?
```

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

Representing Values

```
(define-type Value
  (numV [n : Number])
  (closV [arg : Symbol]
         [body : Exp]
         [env : Env]))

(define-type Binding
  (bind [name : Symbol]
        [val : Value]))
```

```
(test (interp {let {[y 10]} {lambda {x} {+ y x}}}
              mt-env)
      (closV 'x {+ y x}
             (extend-env (bind 'y (numV 10))
                         mt-env)))
```

Continuing Evaluation

```
                       y = (numV 7)
Argument: (interp  y )
               ⇒ (numV 7)
                                        y = (numV 10)
Function: (interp {lambda {x} {+ y x}})
              ⇒ (closV 'x {+ y x}
                         (extend-env (bind 'y (numV 10))
                                     mt-env))
```

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

```
                   x = (numV 7)   y = (numV 10)
       (interp {+ y x})
```

Part 5

Interpreter

```
(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    [(numE n) (numV n)]
    [(idE s) (lookup s env)]
    [(plusE l r) (num+ (interp l env) (interp r env))]
    [(multE l r) ...]
    [(letE n rhs body)
     ...]
    [(lamE n body) ...]
    [(appE fun arg)
     ...]))
```

Add and Multiply

```
(define (num+ [l : Value] [r : Value]) : Value
  (cond
   [(and (numV? l) (numV? r))
    (numV (+ (numV-n l) (numV-n r)))]
   [else
    (error 'interp "not a number")]))

(define (num* [l : Value] [r : Value]) : Value
  (cond
   [(and (numV? l) (numV? r))
    (numV (* (numV-n l) (numV-n r)))]
   [else
    (error 'interp "not a number")]))
```

Add and Multiply

```
(define (num-op op l r)
  (cond
   [(and (numV? l) (numV? r))
    (numV (op (numV-n l) (numV-n r)))]
   [else
    (error 'interp "not a number")]))

(define (num+ [l : Value] [r : Value]) : Value
  (num-op + l r))

(define (num* [l : Value] [r : Value]) : Value
  (num-op * l r))
```

Interpreter

```
(define (interp [a : Exp] [env : Env]) : Value
  (type-case Exp a
    [(numE n) (numV n)]
    [(idE s) (lookup s env)]
    [(plusE l r) (num+ (interp l env) (interp r env))]
    [(multE l r) (num* (interp l env) (interp r env))]
    [(letE n rhs body)
     (interp body (extend-env
                   (bind n (interp rhs env))
                   env))]
    [(lamE n body) (closV n body env)]
    [(appE fun arg)
     (type-case Value (interp fun env)
       [(closV n body c-env)
        (interp body
               (extend-env
                (bind n (interp arg env))
                c-env))]
       [else (error 'interp "not a function")])]))
```