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 our language?

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 `plai-typed`?

Yes

An expression can produce a function result

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

+

(if (positive? n) first second)
```

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 Grammar, Almost

```
<Expr>  ::=  <Num>
          |  <Sym>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  {let {[<Sym> <Expr>]} <Expr>}
          |  {<Sym> <Expr>}                    *
          |  {lambda {<Sym>} <Expr>}           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 ExprC ... -> number`~~

```
; interp ExprC ... -> 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 **<Expr>**

New Grammar

```
<Expr>  ::=  <Num>
          |  <Sym>
          |  {+ <Expr> <Expr>}
          |  {* <Expr> <Expr>}
          |  {let {[<Sym> <Expr>]} <Expr>}
          |  {<Sym> <Expr>}
          |  {lambda {<Sym>} <Expr>}
          |  {<Expr> <Expr>}
```

NEW

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 ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC)
         (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [letC (n : symbol)
        (rhs : ExprC)
        (body : ExprC)]
  [lamC (n : symbol)
        (body : ExprC)]
  [appC (fun : ExprC)
        (arg : ExprC)])
```

```
(test (parse '{lambda {x} {+ x 1}})
      (lamC 'x (plusC (idC 'x) (numC 1))))
```

Expression Datatype

```
(define-type ExprC
  [numC (n : number)]
  [idC (s : symbol)]
  [plusC (l : ExprC)
         (r : ExprC)]
  [multC (l : ExprC)
         (r : ExprC)]
  [letC (n : symbol)
        (rhs : ExprC)
        (body : ExprC)]
  [lamC (n : symbol)
        (body : ExprC)]
  [appC (fun : ExprC)
        (arg : ExprC)])
```

```
(test (parse '{{lambda {x} {+ x 1}} 10})
      (appC (lamC 'x (plusC (idC 'x) (numC 1)))
            (numC 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

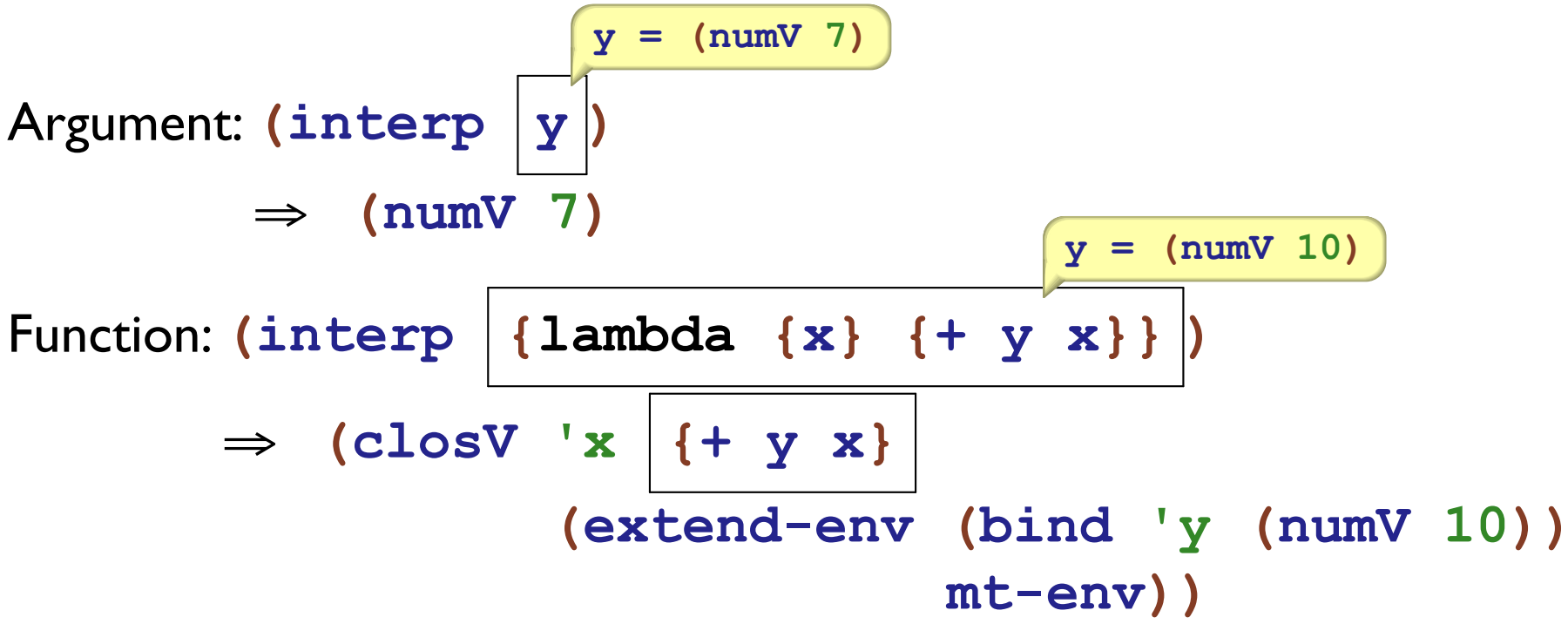
```
(interp {lambda {x} {+ y x}}) ⇒ ?
```

Representing Values

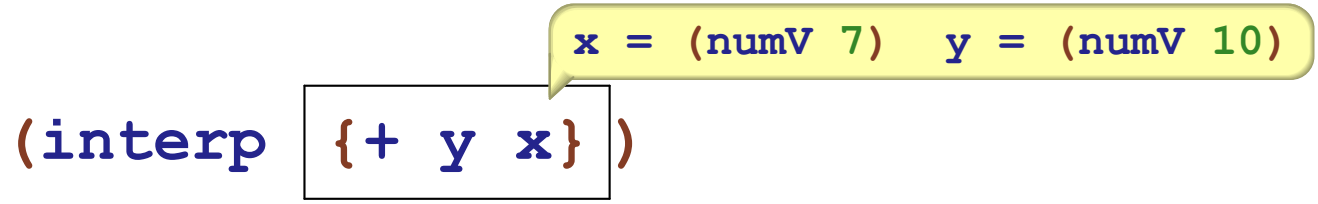
```
(define-type Value
  [numV (n : number)]
  [closV (arg : symbol)
         (body : ExprC)
         (env : Env)])

(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



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



Part 5

Interpreter

```
(define (interp [a : ExprC] [env : Env]) : Value
  (type-case ExprC a
    [numC (n) (numV n)]
    [idC (s) (lookup s env)]
    [plusC (l r) (num+ (interp l env) (interp r env))]
    [multC (l r) ...]
    [letC (n rhs body)
          ...]
    [lamC (n body) ...]
    [appC (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 : ExprC] [env : Env]) : Value
  (type-case ExprC a
    [numC (n) (numV n)]
    [idC (s) (lookup s env)]
    [plusC (l r) (num+ (interp l env) (interp r env))]
    [multC (l r) (num* (interp l env) (interp r env))]
    [letC (n rhs body)
          (interp body (extend-env
                        (bind n (interp rhs env))
                        env))]
    [lamC (n body) (closV n body env)]
    [appC (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")])]))
```