CS4212

Compiling Functional Languages

Matthew Flatt

Goal for Today

~~**Compile** a **functional language** to **assembly**~~

Interpret a **functional language** using **simple expressions**

- Interpreter itself starts out as a functional program
- We convert the interpreter to use only simple instructions:
  arithmetic, assignments, and jumps
- By-hand conversion of interpreter shows what a compiler does
  ... and what run-time data structures are needed:
  environments, *closures*, and *continuations*

Interpreters

A ***compiler*** translates a program

An ***interpreter*** returns a program's value

Plait = the **meta** language

```
(lambda (x) (+ x 1))
```

Curly = the **object** language

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

Racket and Plait

Historically: **Lisp** ⇒ **Scheme** ⇒ **Racket** ⇒ **Plait**

Racket and Plait

Historically: **Lisp** ⇒ **Scheme** ⇒ **Racket** ⇒ **Plait** ⇐ **ML**

Racket is

- a programming language
- a family of programming languages
- a language for creating programming languages

... including **Plait**

PLAI = *Programming Languages: Application and Interpretation*, a textbook

Plait's Parenthesized Prefix Notation

```
f(x)
```

```
(f x)
```

```
1+2
```

```
(+ 1 2)
```

```
1+2*3
```

```
(+ 1 (* 2 3))
```

```
s=6
```

```
(define s 6)
```

```
f(x)=x+1
```

```
(define (f x)
  (+ x 1))
```

$$\begin{cases} x<0 & -1 \\ x=0 & 0 \\ x>0 & 1 \end{cases}$$

```
(cond
  [(< x 0) -1]
  [(= x 0) 0]
  [(> x 0) 1])
```

Plait Data

- Numbers and strings — obvious

```
1   3.4   "Hello, World!"
```

- Booleans — straightforward

```
#t   #f
```

- Symbols and quoted lists — unusual

```
'apple   'define   '+

'(1 2 3)   '(f x)
```

Plait S-Expressions

- Backquote ` instead of regular quote '

convenient

```
`x
```

```
`{+ x 1}
```

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

Plait Datatypes

```
(define-type Shape
  (circle [radius : Number])
  (rectangle [width : Number]
             [height : Number]))

(define (area s)
  (type-case Shape s
    [(circle r) (* 3.14 (* r r))]
    [(rectangle w h) (* w h)]))

(test (area (circle 2))
      12.56)
(test (area (rectangle 3 4))
      12)
```

Meta and Object Languages

Example **Plait** program:

```
(define-type Exp
  (numE [n : Number])
  (plusE [l : Exp]
         [r : Exp])
  (multE [l : Exp]
         [r : Exp]))
```

Example **Curly** program:

```
{+ {* 3 4} 8}
```

Example **Curly** program as a **Plait** value:

```
`{+ {* 3 4} 8}
```

Meta and Object Languages

Example **Plait** program:

```
(define-type Exp
  (numE [n : Number])
  (plusE [l : Exp]
         [r : Exp])
  (multE [l : Exp]
         [r : Exp]))
```

Example **Curly** program:

```
{+ {* 3 4} 8}
```

Example **Curly** program as a **Plait** value:

```
`{+ {* 3 4} 8}
(plusE (multE (numE 3) (numE 4))
       (numE 8))
```

Part 1: Interpreter and Environments

Curly Arithmetic

```
{+ 2 1}
```

↠ `3`

Curly Arithmetic

```
{* 2 1}
```

↠ `2`

Curly Arithmetic

```
{+ 2 {* 4 3}}
```

↠ `14`

Curly Arithmetic

2

↠ 2

Representing Expressions

```
2

{+ 2 1}

{+ 2 {* 4 3}}
```

- numbers
- addition expressions
  - first and second arguments are expressions
- multiplication expressions
  - first and second arguments are expressions

Representing Expressions

```
2

{+ 2 1}

{+ 2 {* 4 3}}
```

```
(define-type Exp
  (numE [n : Number])
  (plusE [l : Exp]
         [r : Exp])
  (multE [l : Exp]
         [r : Exp]))
```

Curly Interpeter

```
(define (interp [a : Exp]) : Number
  (type-case Exp a
    [(numE n) n]
    [(plusE l r) (+ (interp l) (interp r))]
    [(multE l r) (* (interp l) (interp r))]))

(test (interp (numE 2))
      2)
(test (interp (plusE (numE 2) (numE 1)))
      3)
(test (interp (multE (numE 2) (numE 1)))
      2)
(test (interp (plusE (multE (numE 2) (numE 3))
                     (plusE (numE 5) (numE 8))))
      19)
```

Curly with Arithmetic

```
(define-type Exp
  (numE [n : Number])
  (plusE [l : Exp] [r : Exp])
  (multE [l : Exp] [r : Exp]))
```

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
```

Curly with Arithmetic

```
(define-type Exp
  (numE [n : Number])
  (plusE [l : Exp] [r : Exp])
  (multE [l : Exp] [r : Exp]))
```

```
<Exp>  ::=  <Number>
         |  {+ <Exp> <Exp>}
         |  {* <Exp> <Exp>}
```

Backus Naur Form
(***BNF***)

Curly with Local Definitions

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

```
{let {[x {+ 1 2}]}
  {+ x x}}                ⇒   6
```

Curly with Local Definitions

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

    {+ {let {[x {+ 1 2}]}
         {+ x x}}
       1}                          ⇒   7
```

Curly with Local Definitions

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

```
{+ {let {[x {+ 1 2}]}
      {+ x x}}
   {let {[x {- 4 3}]}
      {+ x x}}}                    ⇒   8
```

Curly with Local Definitions

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

```
{+ {let {[x {+ 1 2}]}
     {+ x x}}
   {let {[y {- 4 3}]}
     {+ y y}}}                  ⇒   8
```

Curly with Local Definitions

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

```
{let {[x {+ 1 2}]}
  {let {[x {- 4 3}]}
    {+ x x}}}                    ⇒   2
```

Curly with Local Definitions

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

```
{let {[x {+ 1 2}]}
  {let {[y {- 4 3}]}
    {+ x x}}}                    ⇒   6
```

Curly with Local Definitions

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

```
{let {[x {+ 1 2}]}
  {let {[x {- 4 x}]}
    {+ x x}}}                    ⇒   2
```

Curly with Local Definitions

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

Curly with Local Definitions

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

```
(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]))
```

Substitution

```
(interp {let {[x 1]}
          {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}})
```

⇒

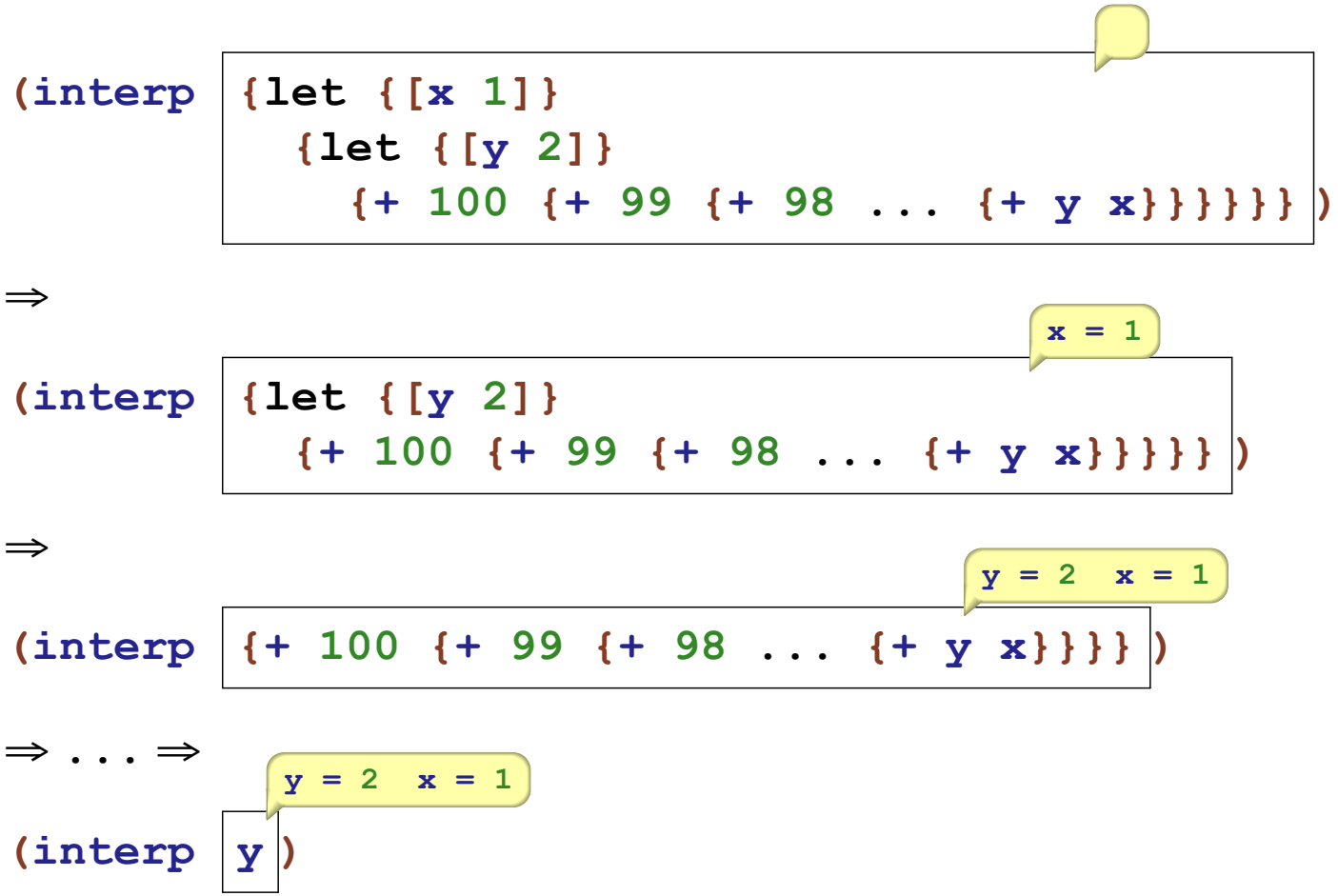
```
(interp {let {[y 2]}
          {+ 100 {+ 99 {+ 98 ... {+ y 1}}}}})
```

⇒

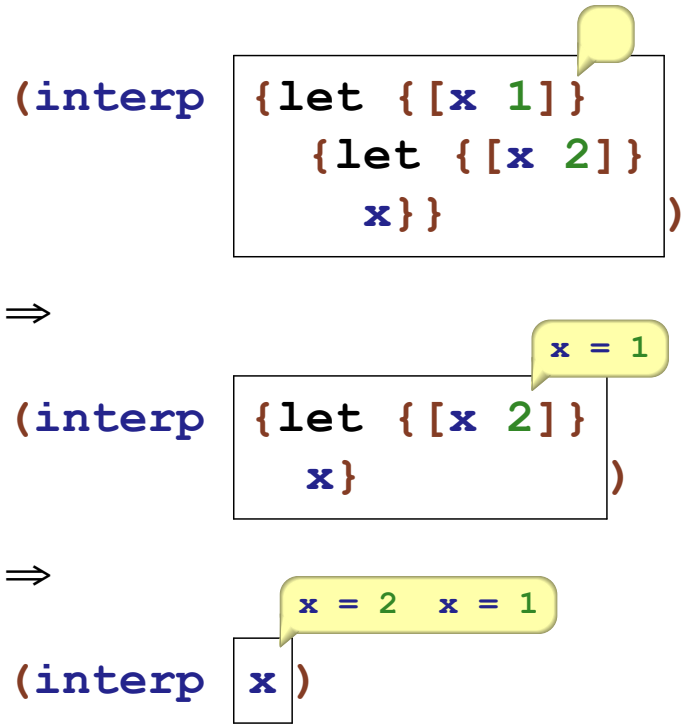
```
(interp {+ 100 {+ 99 {+ 98 ... {+ 2 1}}}})
```

With **n** variables, evaluation will take $O(\mathbf{n}^2)$ time!

Environments as Deferred Substitution



Deferring Substitution with the Same Identifier



Always add to start, then always check from start

Representing Environments

Change

```
interp : (Exp -> Number)
```

to

```
interp : (Exp Env -> Number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (Symbol Number -> Binding)
lookup : (Symbol Env -> Number)
```

```
mt-env
```

Representing Environments

Change

```
interp : (Exp -> Number)
```

to

```
interp : (Exp Env -> Number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (Symbol Number -> Binding)
lookup : (Symbol Env -> Number)
```

x = 1

```
(extend-env (bind 'x 1)
            mt-env)
```

Representing Environments

Change

```
interp : (Exp -> Number)
```

to

```
interp : (Exp Env -> Number)
```

```
mt-env : Env
extend-env : (Binding Env -> Env)
bind : (Symbol Number -> Binding)
lookup : (Symbol Env -> Number)
```

y = 2 x = 1

```
(extend-env (bind 'y 2)
            (extend-env (bind 'x 1)
                        mt-env))
```

Environments

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

(define-type-alias Env (Listof Binding))

(define mt-env empty)
(define extend-env cons)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  ....)
```

```
(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty ....]
    [(cons b rst-env) ....]))
```

```
(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty (error 'lookup "free variable")]
    [(cons b rst-env) ....]))
```

```
(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty (error 'lookup "free variable")]
    [(cons b rst-env) ....
                                                b

                                  (lookup n rst-env)  ]))


(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty (error 'lookup "free variable")]
    [(cons b rst-env) ....
                          (symbol=? n (bind-name b))

                               (lookup n rst-env)  ]))


(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Environment Lookup

```
(define (lookup [n : Symbol] [env : Env]) : Number
  (type-case (Listof Binding) env
    [empty (error 'lookup "free variable")]
    [(cons b rst-env) (cond
                        [(symbol=? n (bind-name b))
                         (bind-val b)]
                        [else (lookup n rst-env)])]))


(test/exn (lookup 'x mt-env)
          "free variable")
(test (lookup 'x (extend-env (bind 'x 1) empty-env))
      1)
(test (lookup 'x (extend-env (bind 'y 1)
                             (extend-env (bind 'x 2) empty-env)))
      2)
```

Interp with Environments

```
(interp {let {[x 1]}
          {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}}
        mt-env)

⇒ (interp {let {[y 2]}
            {+ 100 {+ 99 {+ 98 ... {+ y x}}}}}
          (extend-env (bind 'x 1) mt-env))

⇒ (interp {+ 100 {+ 99 {+ 98 ... {+ y x}}}}
          (extend-env (bind 'y 2)
                      (extend-env (bind 'x 1)
                                  mt-env)))

⇒ ...

⇒ (interp y (extend-env (bind 'y 2)
                        (extend-env (bind 'x 1)
                                    mt-env)))
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env])
  (type-case Exp a
    [(numE n) n]
    [(idE s) ...]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(multE l r) (* (interp l env) (interp r env))]
    [(letE n rhs body) ...]))
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(multE l r) (* (interp l env) (interp r env))]
    [(letE n rhs body) ...]))
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(multE l r) (* (interp l env) (interp r env))]
    [(letE n rhs body) ...]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(multE l r) (* (interp l env) (interp r env))]
    [(letE n rhs body) ...
                                ...
                                 ...      (interp rhs env defs)
                                ...
                          ...           ]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(multE l r) (* (interp l env) (interp r env))]
    [(letE n rhs body) ...
                                  ...
                                   (bind n (interp rhs env defs))
                                  ...
                          ...          ]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(multE l r) (* (interp l env) (interp r env))]
    [(letE n rhs body) ...
                                (extend-env
                                 (bind n (interp rhs env defs))
                                 env)
                       ...           ]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(multE l r) (* (interp l env) (interp r env))]
    [(letE n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env defs))
                                env)
                               defs)]))
```

```
{let {[x {+ 1 2}]}
  {* x x}}
```

Interp with Environments

```
(define (interp [a : Exp] [env : Env])
  (type-case Exp a
    [(numE n) n]
    [(idE s) (lookup s env)]
    [(plusE l r) (+ (interp l env) (interp r env))]
    [(multE l r) (* (interp l env) (interp r env))]
    [(letE n rhs body) (interp body
                               (extend-env
                                (bind n (interp rhs env defs))
                                env)
                               defs)]))
```

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`**,
...

Next step: add functions as values

We'll use the keyword **`lambda`** for historical reasons

Why Functions as Values

Abstraction is easier with functions as values

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

Separate function **`define`** form becomes unnecessary

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

⇒

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

Curly with Functions

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

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)))
```

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}}
```

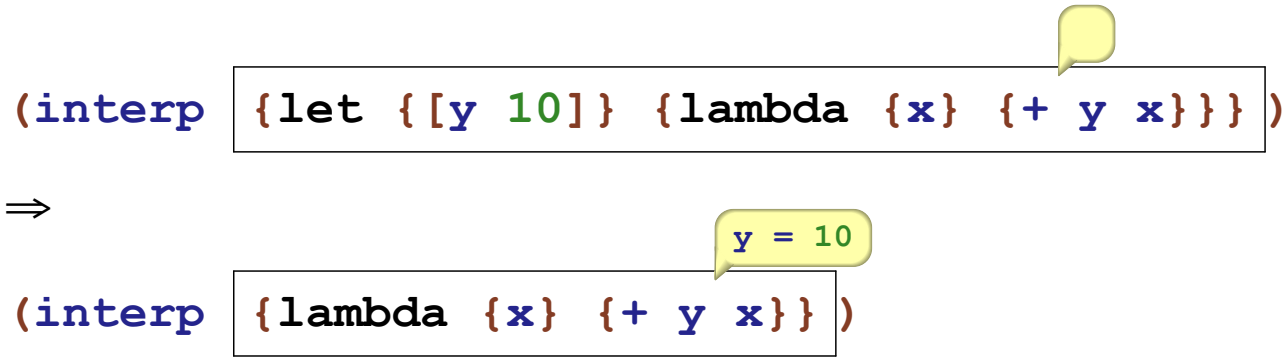
Functions with Substitutions

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

⇒

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

Functions with Deferred Substitution



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}}) ⇒ ?
```

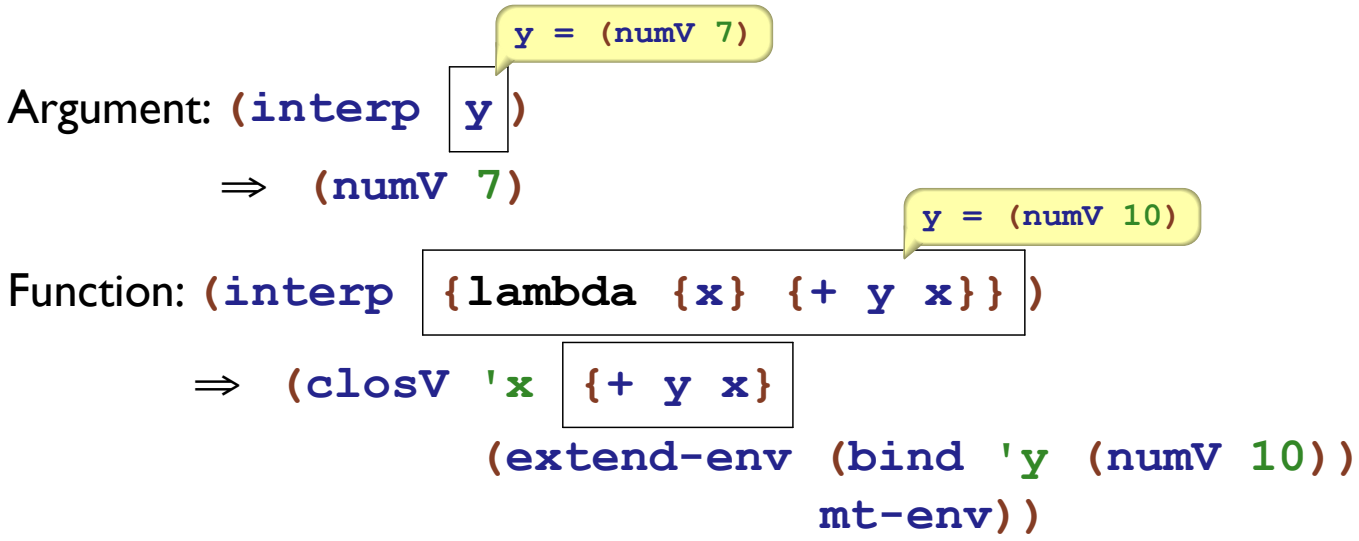
A ***closure*** combines an expression with an environment

Representing Values

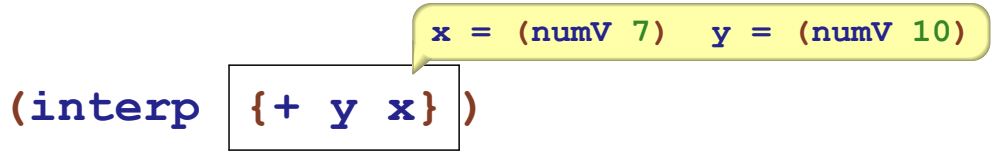
```
(define-type Value
  (numV [n : Number])
  (closV [arg : Symbol]
         [body : Exp]
         [env : Env]))

(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



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



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")])]))
```

Part 2: Continuations and Compilation

Evaluation and "To do" Lists

```
(interp (plusE (numE 1) (numE 2)) mt-env)

⇒ (num+ (interp (numE 1) mt-env)
        (interp (numE 2) mt-env))

⇒ (interp (numE 1) mt-env)
```

To do:
```
(num+ ●
      (interp (numE 2) mt-env))
```

```
⇒ (numV 1)
```

To do:
```
(num+ ●
      (interp (numE 2) mt-env))
```

```
⇒ (interp (numE 2) mt-env)
```

To do:
```
(num+ (numV 1)
      ●)
```

```
⇒ (numV 2)
```

To do:
```
(num+ (numV 1)
      ●)
```

```
⇒ (num+ (numV 1) (numV 2))
```

Continuations

A "to do" list is a ***continuation***

To do:

```
(num+ ●
      (interp (numE 2) mt-env))
```

Continuations

A "to do" list is a ***continuation***

To do:

```
(+ 3
   (* ●
      (f (rest ls))))
```

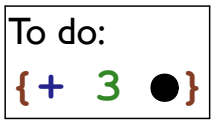
A ***stack*** is one way to implement continuations

To do:

```
(* ● (f (rest ls)))
(+ 3 ●)
```

The terms ***stack*** and ***continuation*** are sometimes used interchangably

Representing Continuations




```
(define-type Cont
  ....)
```

Representing Continuations

To do:
`{+ 3 ●}`

```
(define-type Cont
  (doPlusK [v : Value])
  ....)

 (doPlusK (numV 3))
```

Representing Continuations

To do:
`{+ ● {f 0}}`

```
(define-type Cont
  (doPlusK [v : Value])
  ....)
```

Representing Continuations

To do:
`{+ ● {f 0}}`

```
(define-type Cont
  (plusSecondK [r : Exp]
               [e : Env])
  (doPlusK [v : Value])
  ....)

(plusSecondK (appE (idE 'f) (numE 0))
             mt-env)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  (plusSecondK [r : Exp]
               [e : Env])
  (doPlusK [v : Value])
  ....)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  (plusSecondK [r : Exp]
               [e : Env]
               [k : Cont])
  (doPlusK [v : Value]
           [k : Cont])
  ....)
```

Representing Continuations

To do:

```
{+ ● {f 0}}
{+ 3 ●}
```

```
(define-type Cont
  (doneK)
  (plusSecondK [r : Exp]
               [e : Env]
               [k : Cont])
  (doPlusK [v : Value]
           [k : Cont])
  ....)
```

```
(plusSecondK (appE (idE 'f) (numE 0))
             mt-env
             (doPlusK (numV 3)
                      (doneK)))
```

`interp` with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (num+ (interp l env)
                         (interp r env))]
      ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (num+ ●
                         (interp r env))]
      ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (num+ ●
                         (interp r env))]
      ...)))
```

To do:

`{+ ● <Exp>}`

`interp` with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (num+ ●
                         (interp r env))]
      ...)))
```

To do:

```
(num+ ●
      (interp r env))
```

`interp` with Continuations

```
(define interp : (Exp Env -> Value)
  (lambda (a env)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (plusSecondK r env)]
      ...)))
```

To do:

```
(num+ ●
      (interp r env))
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (plusSecondK r env k)]
      ...)))
```

To do:

```
(num+ ●
      (interp r env))
....
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env)
                   (plusSecondK r env k)]
      ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(plusE l r) (interp l env
                           (plusSecondK r env k))]
      ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (numV n)]
     ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...

      ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(doneK) v]
      ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(plusSecondK r env next-k)
       (interp r env
               ...)]
      ...)))
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(plusSecondK r env next-k)
       (interp r env
               ...)]
      ...)))
```

To do:

```
(num+ ●
      (interp r env))
....
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(plusSecondK r env next-k)
       (interp r env
               ...)]
      ...)))
```

To do:

```
(num+ v
      ●)
....
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(plusSecondK r env next-k)
       (interp r env
               (doPlusK v next-k))]
      ...)))
```

To do:

```
(num+ v
      ●)
....
```

`interp` with Continuations

```
(define interp : (Exp Env Cont -> Value)
  (lambda (a env k)
    (type-case Exp a
      ...
      [(numE n) (continue k (numV n))]
     ...)))

(define continue : (Cont Value -> Value)
  (lambda (k v)
    (type-case Cont k
      ...
      [(doPlusK v-l next-k)
       (continue next-k (num+ v-l v))]
      ...)))
```

To do:

```
(num+ v-l
      ●)
....
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(numE n) (continue k (numV n))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(doneK) v]
    ...))
```

```
(interp (numE 5) mt-env (doneK))

⇒ (continue (doneK) (numV 5))

⇒ (numV 5)
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(plusE l r) (interp l env (plusSecondK r env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(plusSecondK r env next-k)
     (interp r env (doPlusK v next-k))]
    [(doPlusK v-l next-k)
     (continue next-k (num+ v-l v))]
    ...))
```

```
(interp (plusE (numE 5) (numE 2)) mt-env (doneK))

⇒ (interp (numE 5)
          (plusSecondK (numE 2) mt-env (doneK)))

⇒ (continue (plusSecondK (numE 2) mt-env (doneK))
            (numV 5))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(plusE l r) (interp l env (plusSecondK r env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(plusSecondK r env next-k)
     (interp r env (doPlusK v next-k))]
    [(doPlusK v-l next-k)
     (continue next-k (num+ v-l v))]
    ...))
```

```
⇒ (continue (plusSecondK (numE 2) mt-env (doneK))
            (numV 5))

⇒ (interp (numE 2) mt-env
          (doPlusK (numV 5) (doneK)))

⇒ (continue (doPlusK (numV 5) (doneK))
            (numV 2))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(plusE l r) (interp l env (plusSecondK r env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(plusSecondK r env next-k)
     (interp r env (doPlusK v next-k))]
    [(doPlusK v-l next-k)
     (continue next-k (num+ v-l v))]
    ...))
```

```
⇒ (continue (doPlusK (numV 5) (doneK))
            (numV 2))

⇒ (continue (doneK)
            (numV 7))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(lamE n body)
     (continue k (closV n body env))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...

    ...))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                     next-k)]
       [else (error ...)])]
    ...))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                 next-k)]
       [else (error ...)])]
    ...))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                     next-k)]
       [else (error ...)])]
    ...))
```

E_1 =
```
(extend-env
 (bind 'f (closV 'x
                 (idE 'x)
                 mt-env))
 mt-env)
```

```
(interp (appE (idE 'f) (numE 1))
        E1
        (doneK))

⇒ (interp (idE 'f)
          E1
          (appArgK (numE 1) E1 (doneK)))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                     next-k)]
       [else (error ...)])]
    ...))
```

E_1 = `(extend-env (bind 'f (closV 'x (idE 'x) mt-env)) mt-env)`

⇒ `(interp (idE 'f)` E_1 `(appArgK (numE 1)` E_1 `(doneK)))`

⇒ `(continue (appArgK (numE 1)` E_1 `(doneK)) (closV 'x (idE 'x) mt-env))`

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                     next-k)]
       [else (error ...)])]
    ...))
```

E_1 =
```
(extend-env
 (bind 'f (closV 'x
                 (idE 'x)
                 mt-env))
 mt-env)
```

⇒
```
(continue (appArgK (numE 1) E1 (doneK))
          (closV 'x (idE 'x) mt-env))
```

⇒
```
(interp (numE 1)
        E1
        (doAppK (closV 'x (idE 'x) mt-env) (doneK)))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                     next-k)]
       [else (error ...)])]
    ...))
```

E_1 =
```
(extend-env
 (bind 'f (closV 'x
                 (idE 'x)
                 mt-env))
 mt-env)
```

⇒ `(interp (numE 1)` E_1 `(doAppK (closV 'x (idE 'x) mt-env) (doneK)))`

⇒ `(continue (doAppK (closV 'x (idE 'x) mt-env) (doneK))`
`(numV 1))`

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                     next-k)]
       [else (error ...)])]
    ...))
```

E_1 =
```
(extend-env
 (bind 'f (closV 'x
                 (idE 'x)
                 mt-env))
 mt-env)
```

⇒
```
(continue (doAppK (closV 'x (idE 'x) mt-env) (doneK))
          (numV 1))
```

⇒
```
(interp (idE 'x)
        (extend-env (bind 'x (numV 1)) mt-env)
        (doneK))
```

`interp` with Continuations

```
(define (interp [a : Exp] [env : Env] [k : Cont]) : Value
  (type-case Exp a ...
    [(appE fun arg) (interp fun env (appArgK arg env k))]
    ...))

(define (continue [k : Cont] [v : Value]) : Value
  (type-case Cont k ...
    [(appArgK a env next-k)
     (interp a env (doAppK v next-k))]
    [(doAppK v-f next-k)
     (type-case Value v-f
       [(closV n body c-env)
        (interp body (extend-env
                      (bind n v)
                      c-env)
                     next-k)]
       [else (error ...)])]
    ...))
```

E_1 =
```
(extend-env
 (bind 'f (closV 'x
                 (idE 'x)
                 mt-env))
 mt-env)
```

⇒
```
(interp (idE 'x)
        (extend-env (bind 'x (numV 1)) mt-env)
        (doneK))
```

⇒
```
(continue (doneK)
          (numV 1))
```

`interp` and `continue`

In **`lambda-k.rkt`**:

- **`interp`** calls **`continue`** only as a tail call
- **`continue`** calls **`interp`** only as a tail call
- **`lookup`** calls **`lookup`** only as a tail call
- nothing else is recursive

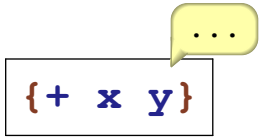
∴ the Plait continuation is always bounded and small

Identifier Address

Suppose that

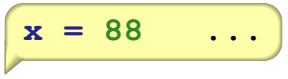
```
{let {[x 88]}
  {+ x y}}
```

appears in a program; the body is eventually evaluated:



where will `x` be in the environment?

Answer: always at the beginning:

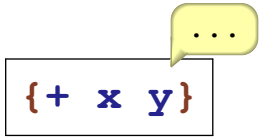


Identifier Address

Suppose that

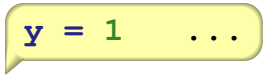
```
{let {[y 1]}
  {+ x y}}
```

appears in a program; the body is eventually evaluated:



where will `y` be in the environment?

Answer: always at the beginning:

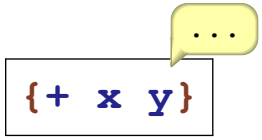


Identifier Address

Suppose that

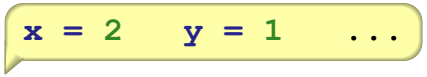
```
{let {[y 1]}
  {let {[x 2]}
    {+ x y}}}
```

appears in a program; the body is eventually evaluated:



where will `y` be in the environment?

Answer: always second:

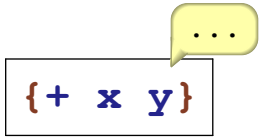


Identifier Address

Suppose that

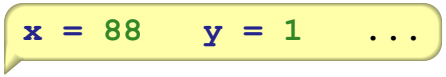
```
{let {[y 1]}
  {let {[x 88]}
    {* {+ x y} 17}}}
```

appears in a program; the body is eventually evaluated:



where will `x` and `y` be in the environment?

Answer: always first and second:

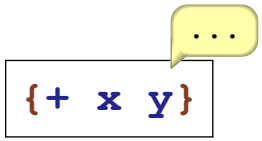


Identifier Address

Suppose that

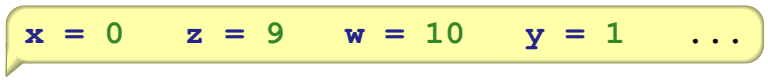
```
{let {[y 1]}
  {let {[w 10]}
    {let {[z 9]}
      {let {[x 0]}
        {+ x y}}}}}
```

appears in a program; the body is eventually evaluated:



where will `x` and `y` be in the environment?

Answer: always first and fourth:

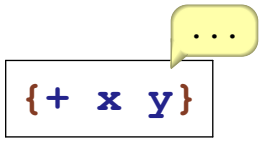


Identifier Address

Suppose that

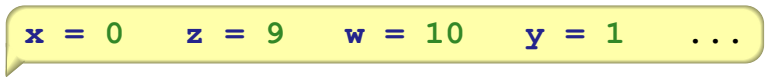
```
{let {[y {let {[r 9]} {* r 8}}]}
  {let {[w 10]}
    {let {[z {let {[q 9]} q}]}
      {let {[x 0]}
        {+ x y}}}}}
```

appears in a program; the body is eventually evaluated:



where will `x` and `y` be in the environment?

Answer: always first and fourth:



Lexical Scope

- For any expression, we can tell which identifiers will be in the environment at run time
- The order of the environment is predictable

Compilation of Variables

A compiler can transform an **Exp** expression to an expression without identifiers — only lexical addresses

```
; compile : Exp ... -> ExpD
```

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

```
(define-type ExpD
  (numD [n Number])
  (addD [l : ExpD]
        [r : ExpD])
  (multD [l : ExpD]
         [r : ExpD])
  (atD [pos : Number])
  (lamD [body : ExpD])
  (appD [fun : ExpD]
        [arg : ExpD]))
```

Compile Examples

```
(compile 1 ...) ⇒ 1

(compile {+ 1 2} ...) ⇒ {+ 1 2}

(compile x ...) ⇒ compile: free identifier

(compile {lambda {x} {+ 1 x}} ...)
⇒ {lambda {+ 1 {at 0}}}

(compile {lambda {y} {lambda {x} {+ x y}}} ...)
  ⇒ {lambda {lambda {+ {at 0} {at 1}}}}
```

Implementing the Compiler

```
(define (compile [a : Exp] [env : EnvC])
  (type-case Exp a
    [(numE n) (numD n)]
    [(plusE l r) (plusD (compile l env)
                        (compile r env))]
    [(multE l r) (multD (compile l env)
                        (compile r env))]
    [(idE n) (atD (locate n env))]
    [(lamE n body-expr)
     (lamD
      (compile body-expr
               (extend-env (bindE n)
                           env)))]
    [(appE fun-expr arg-expr)
     (appD (compile fun-expr env)
           (compile arg-expr env))]))
```

Compile-Time Environment

Mimics the run-time environment, but without values:

```
(define-type BindingC
  (bindE [name : Symbol]))

(define-type-alias EnvC (Listof BindingC))

(define (locate name env)
  (cond
   [(empty? env) (error 'locate "free variable")]
   [else (if (symbol=? name (bindC-name (first env)))
             0
             (+ 1 (locate name (rest env))))]))
```

`interp` for Compiled

Almost the same as `interp` for `Exp`:

```
(define (interp a env)
  (type-case ExpD a
    [(numD n) (numV n)]
    [(plusD l r) (num+ (interp l env)
                       (interp r env))]
    [(multD l r) (num* (interp l env)
                       (interp r env))]
    [(atD pos) (list-ref env pos)]
    [(lamD body-expr)
     (closV body-expr env)]
    [(appD fun-expr arg-expr)
     (let ([fun-val (interp fun-expr env)]
           [arg-val (interp arg-expr env)])
       (interp (closV-body fun-val)
               (cons arg-val
                     (closV-env fun-val))))]))
```

Timing Effect of Compilation

Given

```
(define c {{{{lambda {x}
                {lambda {y}
                  {lambda {z} {+ {+ x x} {+ x x}}}}}
              1}
             2}
            3})
(define d (compile c mt-env))
```

then

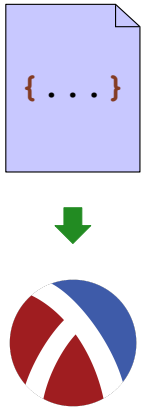
```
(interp d empty)
```

is significantly faster than

```
(interp c mt-env)
```

Using the built-in `list-ref` simulates machine array indexing, but don't take timings too seriously

From Plait to Machine-Like Code



From Plait to Machine-Like Code

From Plait to Machine-Like Code

- Everything must be a number

From Plait to Machine-Like Code

- Everything must be a number
- No **`define-type`** or **`type-case`**

From Plait to Machine-Like Code

- Everything must be a number
- No **`define-type`** or **`type-case`**
- No implicit continuations

From Plait to Machine-Like Code

- Everything must be a number
- No **`define-type`** or **`type-case`**
- No implicit continuations
- No implicit allocation

From Plait to Machine-Like Code

`compile-1.rkt`

Step 1:

`Exp` → `ExpD`

```
{lambda {x}          {lambda
  {+ 1 x}}             {+ 1 {at 0}}}
```

Eliminates all run-time names

From Plait to Machine-Like Code

`compile-2.rkt`

Step 2:

`interp` → `interp` + `continue`

Eliminates implicit continuations

From Plait to Machine-Like Code

`compile-3.rkt`

Step 3:

function calls → regsiters and `goto`

From Plait to Machine-Like Code

`compile-3.rkt`

Step 3:

function calls → regsiters and `goto`

```
(interp l
        env
        (plusSecondK r
                     env
                     k))
```

```
(begin
  (set! expr-reg l)
  (set! k-reg (plusSecondK r
                           env-reg
                           k-reg))
  (interp))
```

Makes argument passing explicit

From Plait to Machine-Like Code

`compile-4.rkt`

Step 4:

```
(multSecondK r          →  (malloc3 3
             env-reg                (ref expr-reg 2)
             k-reg)                 env-reg
                                    k-reg)
```

From Plait to Machine-Like Code

`compile-4.rkt`

Step 4:

```
doneK       → 1
plusSecondK → 2
…
numD        → 8
plusD       → 9
…
numV        → 15
closV       → 16
```

From Plait to Machine-Like Code

`compile-4.rkt`

Step 4:

```
(type-case Cont k-reg          →   (case (ref k-reg 0)
  ...                                ...
  [(multSecondK r env k)             [(3)
   ... r                              ... (ref k-reg 1)
   ... env                            ... (ref k-reg 2)
   ... k ..]                          ... (ref k-reg 3) ...]
  ...)                               ...)
```

From Plait to Machine-Like Code

`compile-4.rkt`

Step 4:

```
(define memory (make-vector 1500 0))
(define ptr-reg 0)

(define (malloc3 tag a b c)
  (begin
    (vector-set! memory ptr-reg tag)
    (vector-set! memory (+ ptr-reg 1) a)
    (vector-set! memory (+ ptr-reg 2) b)
    (vector-set! memory (+ ptr-reg 3) c)
    (set! ptr-reg (+ ptr-reg 4))
    (- ptr-reg 4)))
```

Makes all allocation explicit

Makes everything a number