COMPUT325: Meta-interpretation

Dr. B. Price and Dr. R. Greiner

26th October 2004

Introduction

- λ-calculus fully expresses computations of any programming language

Introduction

- λ-calculus fully expresses computations of any programming language
- Is λ-calculus sufficiently expressive to express itself?

Implementing λ-calculus

- What would be required to automate λ-calculus representation and evaluation

Implementing λ-calculus

- What would be required to automate λ-calculus representation and evaluation
  - representation for constants, applications and function definitions

Implementing λ-calculus

- What would be required to automate λ-calculus representation and evaluation
  - representation for constants, applications and function definitions
  - Function for checking types of data

Implementing λ-calculus

- What would be required to automate λ-calculus representation and evaluation
  - representation for constants, applications and function definitions
  - Function for checking types of data
  - Functions for creating and accessing components of representations

Implementing λ-calculus

- What would be required to automate λ-calculus representation and evaluation
  - representation for constants, applications and function definitions
  - Function for checking types of data
  - Functions for creating and accessing components of representations
  - Functions for λ-calculus evaluation
    - checking for free variables
    - renaming variables
    - performing substitutions

Implementing λ-calculus

- What would be required to automate λ-calculus representation and evaluation
  - representation for constants, applications and function definitions
  - Function for checking types of data
  - Functions for creating and accessing components of representations
  - Functions for λ-calculus evaluation
    - checking for free variables
    - renaming variables
    - performing substitutions
  - garbage collection

Why Garbage Collection

- No imperative assignment $\longrightarrow$ no side-effects

Why Garbage Collection

- No imperative assignment $\longrightarrow$ no side-effects
- Efficiency maintained by shared references

Why Garbage Collection

- No imperative assignment $\longrightarrow$ no side-effects
- Efficiency maintained by shared references
- Sharing $\longrightarrow$ function arguments may be shared by others

$$(\lambda \texttt{x} \mid \texttt{(CONS } \underbrace{\texttt{(CONS 2 x)}}_{\texttt{x shared?}} \texttt{ (CONS 3 x))} \quad \underbrace{\texttt{(CONS 1 nil)}}_{\texttt{allocated CONS}}$$

Why Garbage Collection

- No imperative assignment $\longrightarrow$ no side-effects
- Efficiency maintained by shared references
- Sharing $\longrightarrow$ function arguments may be shared by others

$(\lambda$`x | (CONS (CONS 2 x) (CONS 3 x))`   `(CONS 1 nil)`

`(CONS 2 x)` — x shared?

`(CONS 1 nil)` — allocated CONS

- Function cannot tell if it is safe to modify arguments (i.e. cannot deallocate!)

Why Garbage Collection

- No imperative assignment $\longrightarrow$ no side-effects
- Efficiency maintained by shared references
- Sharing $\longrightarrow$ function arguments may be shared by others

$$(\lambda \texttt{x} \mid \texttt{(CONS } \underbrace{\texttt{(CONS 2 x)}}_{\texttt{x shared?}} \texttt{ (CONS 3 x))} \quad \underbrace{\texttt{(CONS 1 nil)}}_{\texttt{allocated CONS}}$$

- Function cannot tell if it is safe to modify arguments (i.e. cannot deallocate!)
- But functions must allocate memory for new values

Why Garbage Collection

- No imperative assignment → no side-effects
- Efficiency maintained by shared references
- Sharing → function arguments may be shared by others

$$(\lambda \texttt{x} \mid \texttt{(CONS } \underbrace{\texttt{(CONS 2 x)}}_{\texttt{x shared?}} \texttt{ (CONS 3 x))} \quad \underbrace{\texttt{(CONS 1 nil)}}_{\texttt{allocated CONS}}$$

- Function cannot tell if it is safe to modify arguments (i.e. cannot deallocate!)
- But functions must allocate memory for new values
- Recursive loops could quickly consume all memory

Why Garbage Collection

- No imperative assignment → no side-effects
- Efficiency maintained by shared references
- Sharing → function arguments may be shared by others

$(\lambda x \mid \texttt{(CONS } \underbrace{\texttt{(CONS 2 x)}}_{\texttt{x shared?}} \texttt{ (CONS 3 x))} \quad \underbrace{\texttt{(CONS 1 nil)}}_{\texttt{allocated CONS}}$

- Function cannot tell if it is safe to modify arguments (i.e. cannot deallocate!)
- But functions must allocate memory for new values
- Recursive loops could quickly consume all memory
- Garbage collectors analyze *global* pattern of dependencies to safely deallocate data

More on Memory Management

- "primitive values" with no shared sub-components can be passed by value - eliminating memory allocation

More on Memory Management

- "primitive values" with no shared sub-components can be passed by value - eliminating memory allocation
- static analysis of programs can detect arguments that are used only once (linearity)

More on Memory Management

- "primitive values" with no shared sub-components can be passed by value - eliminating memory allocation
- static analysis of programs can detect arguments that are used only once (linearity)
- programs can then be optimized to do
  - imperative in-place modification when it is safe
  - deterministic deallocation of memory to avoid garbage generation

More on Memory Management

- "primitive values" with no shared sub-components can be passed by value - eliminating memory allocation
- static analysis of programs can detect arguments that are used only once (linearity)
- programs can then be optimized to do
  - imperative in-place modification when it is safe
  - deterministic deallocation of memory to avoid garbage generation
- For small toy examples, we can ignore garbage collection issues

Representation: λ-Calculus BNF

- What do we have to represent?

Representation: λ-Calculus BNF

- What do we have to represent?

$$\langle\texttt{expression}\rangle\texttt{:=}\langle\texttt{identifier}\rangle\ \texttt{|}\ \langle\texttt{application}\rangle\ \texttt{|}\ \langle\texttt{function}\rangle$$

Representation: λ-Calculus BNF

- What do we have to represent?

$$\langle \texttt{expression} \rangle \texttt{:=} \langle \texttt{identifier} \rangle \;\texttt{|}\; \langle \texttt{application} \rangle \;\texttt{|}\; \langle \texttt{function} \rangle$$

$$\langle \texttt{identifier} \rangle \;\texttt{:=}\; \texttt{a} \;\texttt{|}\; \texttt{b} \;\texttt{|}\; \texttt{c} \;\texttt{|}\; \cdots$$

Representation: λ-Calculus BNF

- What do we have to represent?

⟨expression⟩:=⟨identifier⟩ | ⟨application⟩ | ⟨function⟩

⟨identifier⟩ := a | b | c | ···

⟨application⟩:= "(" ⟨expression⟩ ⟨expression⟩ ")"

Representation: λ-Calculus BNF

- What do we have to represent?

$\langle$expression$\rangle$:=$\langle$identifier$\rangle$ | $\langle$application$\rangle$ | $\langle$function$\rangle$

$\langle$identifier$\rangle$:= a | b | c | $\cdots$

$\langle$application$\rangle$:= "(" $\langle$expression$\rangle$ $\langle$expression$\rangle$ ")"

$\langle$function$\rangle$:= "(λ" $\langle$identifier$\rangle$ "|" $\langle$expression$\rangle$ ")"

Primitiveλ-Calculus Representation I

- In λ-calculus, all data types are represented as λ expressions

Primitiveλ-Calculus Representation I

- In λ-calculus, all data types are represented as λ expressions
- Need a way to distinguish: identifier, application, function

Primitiveλ-Calculus Representation I

- In λ-calculus, all data types are represented as λ expressions
- Need a way to distinguish: identifier, application, function
- Use a cons cell where `FIRST` is type, and `SECOND` is data

Primitive λ-Calculus Representation I

- In λ-calculus, all data types are represented as λ expressions
- Need a way to distinguish: identifier, application, function
- Use a cons cell where `FIRST` is type, and `SECOND` is data
- Let the integers 0, 1, 2 denote identifiers, applications and function defs respectively
  - Let Φ be the appropriate λ-calculus representation

```
[0   Φ]   ;; an identifier
[1   Φ]   ;; an application of functions
[2   Φ]   ;; a function definition
```

Primitiveλ-Calculus Representation II

- Use cons cell type marker with Church integers for identifiers
  - Instead of `x,y,z` we use integer identifiers
  - To discriminate from numeric integers, write `$0, $1, $2, ...`
  - Where `$0` is type-marked identifier with church number 0 i.e. $\texttt{\$0} \equiv \texttt{cons(}\underbrace{\texttt{0}}_{\texttt{type}}\texttt{,}\underbrace{\texttt{0}}_{\texttt{id}}\texttt{)}$, $\texttt{\$1} \equiv \texttt{cons(}\underbrace{\texttt{0}}_{\texttt{type}}\texttt{,}\underbrace{\texttt{1}}_{\texttt{id}}\texttt{)}$

Primitiveλ-Calculus Representation II

- Use cons cell type marker with Church integers for identifiers
  - Instead of x,y,z we use integer identifiers
  - To discriminate from numeric integers, write $0, $1, $2, ...
  - Where $0 is type-marked identifier with church number 0 i.e. $\$0 \equiv \texttt{cons}(\underbrace{0}_{\texttt{type}}, \underbrace{0}_{\texttt{id}})$, $\$1 \equiv \texttt{cons}(\underbrace{0}_{\texttt{type}}, \underbrace{1}_{\texttt{id}})$
- Use cons cell type marker with cons cell for applications
  - Consider application of a to b, (a b)
  - To discriminate from lists, write application (a b) as $(a b)

$$\equiv \$(\$0\ \$1)$$

$$\equiv \texttt{cons}(\underbrace{1}_{\texttt{type}}, \texttt{cons}(\texttt{cons}(\underbrace{0}_{\texttt{type}}, \underbrace{0}_{\texttt{id}}), \texttt{cons}(\underbrace{0}_{\texttt{type}}, \underbrace{1}_{\texttt{id}})))$$

Primitive λ-Calculus Representation III

- Again, use CONS cell for function definition:

Primitiveλ-Calculus Representation III

- Again, use CONS cell for function definition:

  (λa | (a b))

Primitive λ-Calculus Representation III

- Again, use CONS cell for function definition:

```
(λa | (a b))
≡$(λ$0 | $( $0 $1) )
```

Primitive λ-Calculus Representation III

- Again, use CONS cell for function definition:

```
(λa | (a b))
≡$(λ$0 | $( $0 $1) )
≡cons( 2 ,          ;; Type marker for function def
       type
```

Primitive λ-Calculus Representation III

- Again, use CONS cell for function definition:

```
(λa | (a b))
≡$(λ$0 | $( $0 $1) )
≡cons( 2 ,          ;; Type marker for function def
       type
    cons(                ;; Cons of parm and body
```

Primitive λ-Calculus Representation III

- Again, use CONS cell for function definition:

```
(λa | (a b))
≡$(λ$0 | $( $0 $1) )
≡cons( 2 ,          ;; Type marker for function def
       type
    cons(                ;; Cons of parm and body
      cons( 0 , 0 )          ;; Parameter a
            type id
```

Primitive λ-Calculus Representation III

- Again, use CONS cell for function definition:

```
(λa | (a b))
≡$(λ$0 | $( $0 $1) )
≡cons( 2 ,              ;; Type marker for function def
      type
   cons(                       ;; Cons of parm and body
      cons( 0 , 0 )                ;; Parameter a
          type  id
      cons( 1 ,                    ;; Type for application
           type
```

Primitive λ-Calculus Representation III

- Again, use CONS cell for function definition:

```
(λa | (a b))
≡$(λ$0 | $( $0 $1) )
≡cons( 2 ,            ;; Type marker for function def
      type
    cons(                       ;; Cons of parm and body
      cons( 0 , 0 )               ;; Parameter a
           type id
      cons( 1 ,                  ;; Type for application
           type
        cons(cons( 0 , 0 ) ,cons( 0 , 1 )) ))) ;; Body (
                   type  id        type  id
```

Creating Representations I

Using abstract programming idioms

Creating Representations I

Using abstract programming idioms

```
new-id(last-id)
  ;; create a new identifier with type marker
  ≡ cons(0,successor(second(last-id)))
```

Creating Representations I

Using abstract programming idioms

```
new-id(last-id)
  ;; create a new identifier with type marker
  ≡ cons(0,successor(second(last-id)))

new-app(function,argument)
  ;; create a new function application with type
  ≡cons(1,cons(function,argument))
```

Creating Representations I

Using abstract programming idioms

```
new-id(last-id)
  ;; create a new identifier with type marker
  ≡ cons(0,successor(second(last-id)))

new-app(function,argument)
  ;; create a new function application with type
  ≡cons(1,cons(function,argument))

new-def(parameter, body)
  ;;create a new function definition with type
  ≡ cons(2,cons(parameter,body))
```

Creating Representations II

```
(λa | (a b) ) c
```

Creating Representations II

```
(λa | (a b) ) c
≡ LET a = 0 IN
    LET b = new-id(a) IN
      LET c = new-id(b) IN
```

Creating Representations II

```
(λa | (a b) ) c
≡ LET a = 0 IN
    LET b = new-id(a) IN
      LET c = new-id(b) IN
        new-app(
          new-def(a, new-app(a,b)),
          c)
```

Representation of Type Predicates

Predicates using abstract programming idioms

- Recall: all datatypes are of the form: `(type, value)`

Representation of Type Predicates

Predicates using abstract programming idioms

- Recall: all datatypes are of the form: (type, value)

```
is-id(⟨E⟩)
   ;; True if ⟨E⟩ is constant identifier
   ≡ IF car(⟨E⟩)=0 THEN T ELSE F
```

Representation of Type Predicates

Predicates using abstract programming idioms

- Recall: all datatypes are of the form: `(type, value)`

```
is-id(⟨E⟩)
  ;; True if ⟨E⟩ is constant identifier
  ≡ IF car(⟨E⟩)=0 THEN T ELSE F

is-app(⟨E⟩)
  ;; True if ⟨E⟩ is constant identifier
  ≡ IF car(⟨E⟩)=1 THEN T ELSE F
```

Representation of Type Predicates

Predicates using abstract programming idioms

- Recall: all datatypes are of the form: `(type, value)`

```
is-id(⟨E⟩)
  ;; True if ⟨E⟩ is constant identifier
  ≡ IF car(⟨E⟩)=0 THEN T ELSE F

is-app(⟨E⟩)
  ;; True if ⟨E⟩ is constant identifier
  ≡ IF car(⟨E⟩)=1 THEN T ELSE F

is-func(⟨E⟩)
  ;; True if ⟨E⟩ is constant identifier
  ≡ IF car(⟨E⟩)=2 THEN T ELSE F
```

Accessing Representations

Abstract idioms for datatypes of the form `(type,value)`

Accessing Representations

Abstract idioms for datatypes of the form `(type,value)`

- Application Accessors for `(type (function argument))`

```
get-func(A) ≡ car(cdr(A)) ;ie funct of application
```

Accessing Representations

Abstract idioms for datatypes of the form `(type,value)`

- Application Accessors for `(type (function argument))`

```
get-func(A) ≡ car(cdr(A)) ;ie funct of application

get-arg(A) ≡ cdr(cdr(A)) ;ie arg of application
```

Accessing Representations

Abstract idioms for datatypes of the form `(type,value)`

- Application Accessors for `(type (function argument))`

```
get-func(A) ≡ car(cdr(A)) ;ie funct of application

get-arg(A) ≡ cdr(cdr(A)) ;ie arg of application
```

- Function Definition Accessors for `(type (parameter body))`

Accessing Representations

Abstract idioms for datatypes of the form `(type,value)`

- Application Accessors for `(type (function argument))`

  `get-func(A)` $\equiv$ `car(cdr(A)) ;ie funct of application`

  `get-arg(A)` $\equiv$ `cdr(cdr(A)) ;ie arg of application`

- Function Definition Accessors for `(type (parameter body))`

  `get-parm(F)` $\equiv$ `car(cdr(F)) ;ie get` λ `parameter`

Accessing Representations

Abstract idioms for datatypes of the form `(type,value)`

- Application Accessors for `(type (function argument))`

```
get-func(A) ≡ car(cdr(A)) ;ie funct of application

get-arg(A) ≡ cdr(cdr(A)) ;ie arg of application
```

- Function Definition Accessors for `(type (parameter body))`

```
get-parm(F) ≡ car(cdr(F)) ;ie get λ parameter


get-body(F) ≡ cdr(cdr(F))
```

λ-calculus Evaluation Function

- Implement λ-evaluation as 3 functions:

λ-calculus Evaluation Function

- Implement λ-evaluation as 3 functions:
  - eval: takes a λ-calculus expression and returns its evaluation

λ-calculus Evaluation Function

- Implement λ-evaluation as 3 functions:
  - eval: takes a λ-calculus expression and returns its evaluation
  - apply: applies a function to an argument

λ-calculus Evaluation Function

- Implement λ-evaluation as 3 functions:
  - eval: takes a λ-calculus expression and returns its evaluation
  - apply: applies a function to an argument
  - subs: substitutes an expression for a constant in an expression

λ-calculus Evaluation Function

- Implement λ-evaluation as 3 functions:
  - eval: takes a λ-calculus expression and returns its evaluation
  - apply: applies a function to an argument
  - subs: substitutes an expression for a constant in an expression
- Implementations are given in abstract programming notation

λ-Calculus Eval Function

```
eval(⟨E⟩) ≡

  IF is-id(e)
  THEN ;;⟨E⟩≡f  : a constant
    e
```

λ-Calculus Eval Function

```
eval(⟨E⟩) ≡

  IF is-id(e)
  THEN ;;⟨E⟩≡f   : a constant
    e

  ELSE IF is-app(e)
    THEN  ;;⟨E⟩≡(⟨F⟩ ⟨A⟩) : application
      apply(get-func(e), get-arg(e))
```

λ-Calculus Eval Function

```
eval(⟨E⟩) ≡

  IF is-id(e)
  THEN ;;⟨E⟩≡f   : a constant
    e

  ELSE IF is-app(e)
    THEN  ;;⟨E⟩≡(⟨F⟩ ⟨A⟩) : application
      apply(get-func(e), get-arg(e))

    ELSE ;;⟨E⟩≡(λx | ⟨BODY⟩ ) : definition
      new-func(get-parm(e), eval(get-body(e)))
```

λ-Calculus Eval Function

```
eval(⟨E⟩) ≡

  IF is-id(e)
  THEN ;;⟨E⟩≡f   : a constant
    e

  ELSE IF is-app(e)
    THEN  ;;⟨E⟩≡(⟨F⟩ ⟨A⟩) : application
      apply(get-func(e), get-arg(e))

    ELSE ;;⟨E⟩≡(λx / ⟨BODY⟩ ) : definition
      new-func(get-parm(e), eval(get-body(e)))
```

- Note: body of definitions are evaluated before use

Applicative-Order Apply Function

```
apply(⟨F⟩,⟨A⟩) ≡  ;; apply function ⟨F⟩ to argument ⟨A⟩
  LET b=eval(⟨A⟩) IN
```

Applicative-Order Apply Function

```
apply(⟨F⟩,⟨A⟩) ≡  ;; apply function ⟨F⟩ to argument ⟨A⟩
  LET b=eval(⟨A⟩) IN
    IF is-id(⟨F⟩)
    THEN ;;(⟨F⟩⟨A⟩)≡(f⟨A⟩)
  d    new-app(⟨F⟩, b)
```

Applicative-Order Apply Function

```
apply(⟨F⟩,⟨A⟩) ≡  ;; apply function ⟨F⟩ to argument ⟨A⟩
  LET b=eval(⟨A⟩) IN
    IF is-id(⟨F⟩)
    THEN ;;(⟨F⟩⟨A⟩)≡(f⟨A⟩)
  d     new-app(⟨F⟩, b)
    ELSE IF is-app(⟨F⟩)
      THEN ;;(⟨F⟩⟨A⟩)≡((⟨G⟩⟨C⟩)⟨A⟩)
        IF is-id(get-func(⟨F⟩))
        THEN new-app(
              new-app(get-func(⟨F⟩,eval(⟨C⟩)),
              b)
        ELSE apply(eval(⟨F⟩), b)
```

Applicative-Order Apply Function

```
apply(⟨F⟩,⟨A⟩) ≡  ;; apply function ⟨F⟩ to argument ⟨A⟩
  LET b=eval(⟨A⟩) IN
    IF is-id(⟨F⟩)
    THEN ;;(⟨F⟩⟨A⟩)≡(f⟨A⟩)
  d    new-app(⟨F⟩, b)
    ELSE IF is-app(⟨F⟩)
      THEN ;;(⟨F⟩⟨A⟩)≡((⟨G⟩⟨C⟩)⟨A⟩)
        IF is-id(get-func(⟨F⟩))
        THEN new-app(
              new-app(get-func(⟨F⟩,eval(⟨C⟩)),
              b)
        ELSE apply(eval(⟨F⟩), b)

      ELSE ;;((λx|⟨G⟩)⟨A⟩)
        eval(subs(b, get-parm(⟨F⟩), get-body(⟨F⟩)))
```

λ-Calculus Substitution I

- In an application like $(\lambda x \mid (\lambda y \mid x))$ y
  - argument x is a free variable that would get bound on substitution
  - so, formal parameter λy must be renamed

λ-Calculus Substitution I

- In an application like (λx | (λy | x)) y
  - argument x is a free variable that would get bound on substitution
  - so, formal parameter λy must be renamed
- In an application like (λy | y) x
  - formal parameter λy does not have to be renamed
  - But, renaming λy does not alter meaning

λ-Calculus Substitution I

- In an application like (λx | (λy | x)) y
  - argument x is a free variable that would get bound on substitution
  - so, formal parameter λy must be renamed
- In an application like (λy | y) x
  - formal parameter λy does not have to be renamed
  - But, renaming λy does not alter meaning
- Simplification: Do not check for free parameters — always rename formal parameters

λ-Calculus Substitution II

```
subs(s,v,⟨E⟩)  ;; substitute s for var v in expression ⟨E⟩

  IF is-id(⟨E⟩)
  THEN ;; base case, either constant matches or not
    IF ⟨E⟩=v THEN s ELSE ⟨E⟩
```

λ-Calculus Substitution II

```
subs(s,v,⟨E⟩)  ;; substitute s for var v in expression ⟨E⟩

  IF is-id(⟨E⟩)
  THEN ;; base case, either constant matches or not
    IF ⟨E⟩=v THEN s ELSE ⟨E⟩

  ELSE IF is-app(⟨E⟩)
  THEN ;; application, substitute within (⟨F⟩ ⟨A⟩)
    new-app( subs(s,v,get-func(⟨E⟩)),
             subs(s,v,get-arg(⟨E⟩)))
```

λ-Calculus Substitution II

```
subs(s,v,⟨E⟩)  ;; substitute s for var v in expression ⟨E⟩

  IF is-id(⟨E⟩)
  THEN ;; base case, either constant matches or not
    IF ⟨E⟩=v THEN s ELSE ⟨E⟩

  ELSE IF is-app(⟨E⟩)
  THEN ;; application, substitute within (⟨F⟩ ⟨A⟩)
    new-app( subs(s,v,get-func(⟨E⟩)),
             subs(s,v,get-arg(⟨E⟩)))
```

(continued on next slide ...)

λ-Calculus Substitution III

```
ELSE ;; Definition (λf/⟨B⟩) - check variable issues!
    LET f = get-parm(⟨E⟩) IN
```

λ-Calculus Substitution III

```
ELSE ;; Definition (λf|⟨B⟩) - check variable issues!

    LET f = get-parm(⟨E⟩) IN

      IF f=v
      THEN ;; var shadowed by formal parameter -> done!
        ⟨E⟩
```

λ-Calculus Substitution III

```
ELSE ;; Definition (λf|⟨B⟩) - check variable issues!

    LET f = get-parm(⟨E⟩) IN

       IF f=v
       THEN ;; var shadowed by formal parameter -> done!
         ⟨E⟩

       ELSE ;; always rename binding variable
          LET z=new-id() AND b = get-body(⟨E⟩) IN
             new-func(
               z, subs(s,v,        ;; beta substitution
                       subs(z,f,b))) ;; alpha renaming
```

Appyling λ-Calculus Evaluation

- To evaluate: (λ x | x) a

Appyling λ-Calculus Evaluation

- To evaluate: (λ x | x) a

```
LETREC zero = (λ sz| z)
AND    successor = (λx (λsz|s(xsz))
AND    add =
```

Appyling λ-Calculus Evaluation

- To evaluate: $(\lambda\ x \mid x)\ a$

```
LETREC zero = (λ sz| z)
AND    successor = (λx (λsz|s(xsz))
AND    add =
⋮
AND    zerop =
```

Appyling λ-Calculus Evaluation

- To evaluate: $(\lambda\ x \mid x)\ a$

```
LETREC zero = (λ sz| z)
AND    successor = (λx (λsz|s(xsz))
AND    add =
⋮
AND    zerop =
⋮
AND    eval = ⟨BODY⟩
AND    apply = ⟨BODY⟩ AND subs = ⟨BODY⟩  IN
```

Appyling λ-Calculus Evaluation

- To evaluate: $(\lambda\ x \mid x)\ a$

```
LETREC zero = (λ sz| z)
AND    successor = (λx (λsz|s(xsz))
AND    add =
⋮
AND    zerop =
⋮
AND    eval = ⟨BODY⟩
AND    apply = ⟨BODY⟩ AND subs = ⟨BODY⟩  IN

LET x = 0 IN
LET a = new-id(x) IN
```

Appyling λ-Calculus Evaluation

- To evaluate: (λ x | x) a

```
LETREC zero = (λ sz| z)
AND    successor = (λx (λsz|s(xsz))
AND    add =
⋮
AND    zerop =
⋮
AND    eval = ⟨BODY⟩
AND    apply = ⟨BODY⟩ AND subs = ⟨BODY⟩  IN

LET x = 0 IN
LET a = new-id(x) IN
   eval( new-app(new-func(x,x),a) )
```

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

```
eval[ (λy | s) ]    ;; Case: function def
```

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

```
eval[ (λy | s) ]    ;; Case: function def
  new-func[  get-id[ (λy | s) ]
```

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

```
eval[ (λy | s) ]    ;; Case: function def
  new-func[  get-id[ (λy | s) ]
             eval[s] ]
```

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

```
eval[ (λy | s) ]    ;; Case: function def
  new-func[  get-id[ (λy | s) ]
             eval[s] ]
    get-id[ (λy | s) ] → y
```

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

```
eval[ (λy | s) ]    ;; Case: function def
  new-func[  get-id[ (λy | s) ]
             eval[s] ]
    get-id[ (λy | s) ] → y
    get-body[ (λy | s) ] → s
```

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

```
eval[ (λy | s) ]    ;; Case: function def
  new-func[  get-id[ (λy | s) ]
             eval[s] ]
    get-id[ (λy | s) ] → y
    get-body[ (λy | s) ] → s
  new-func[ y, s ]
```

λ-Calculus Evaluation Example I

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with λ-C arguments

```
eval[ (λy | s) ]    ;; Case: function def
  new-func[  get-id[ (λy | s) ]
             eval[s] ]
    get-id[ (λy | s) ] → y
    get-body[ (λy | s) ] → s
  new-func[ y, s ]
→(λy | s)
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]   ;; Case: application
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)

      eval[
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)

      eval[
         subs[  eval[x],
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)

      eval[
         subs[  eval[x],
                get-id[ (λy | s) ]
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)

      eval[
         subs[  eval[x],
                get-id[ (λy | s) ]
                get-body[ (λy| s) ]  ]
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)

      eval[
         subs[  eval[x],
                get-id[ (λy | s) ]
                get-body[ (λy| s) ]  ]
      eval[ subs[ x, y, s ] ]
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)

      eval[
         subs[  eval[x],
                get-id[ (λy | s) ]
                get-body[ (λy| s) ]  ]
      eval[ subs[ x, y, s ] ]
      eval[ s]
```

λ-Calculus Evaluation Example II

```
eval[ ((λy | s) x) ]  ;; Case: application

   apply[ get-fun[ ((λy|s) x) ],
          get-arg[ ((λy|s) x) ]  ]

 ≡apply[ (λy | s), x ] ;; (definition, arg)

      eval[
         subs[  eval[x],
                get-id[ (λy | s) ]
                get-body[ (λy| s) ]  ]
      eval[ subs[ x, y, s ] ]
      eval[ s]
→  s
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
           eval[subs[eval[x], y ,s)]]
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
           eval[subs[eval[x], y ,s)]]
             eval[x] → x  ;; constant identifier
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
           eval[subs[eval[x], y ,s)]]
             eval[x] → x  ;; constant identifier
           eval[subs[ x, y, s ] ]
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
           eval[subs[eval[x], y ,s)]]
             eval[x] → x  ;; constant identifier
           eval[subs[ x, y, s ] ]
      → s
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
           eval[subs[eval[x], y ,s)]]
             eval[x] → x  ;; constant identifier
           eval[subs[ x, y, s ] ]
      → s
    eval[ subs[s,y,s] ]
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
           eval[subs[eval[x], y ,s)]]
             eval[x] → x  ;; constant identifier
           eval[subs[ x, y, s ] ]
      → s
    eval[ subs[s,y,s] ]
      subs[s,y,s] → s
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
           eval[subs[eval[x], y ,s)]]
             eval[x] → x  ;; constant identifier
           eval[subs[ x, y, s ] ]
      → s
    eval[ subs[s,y,s] ]
      subs[s,y,s] → s
    eval[s]
```

λ-Calculus Evaluation Example III

```
eval [ ( (λy|s) ((λy|s) x) ) ] ;; Case: application
  apply[ (λy|s), ((λy|s) x)  ] ;; Case: (Def, Arg)
    eval[ subs[ eval[((λy|s) x)], y, s ]
      eval[((λy|s) x)] ; case: application
        apply[ (λy | s), x ] ;; (definition, arg)
           eval[subs[eval[x], y ,s)]]
             eval[x] → x  ;; constant identifier
           eval[subs[ x, y, s ] ]
      → s
    eval[ subs[s,y,s] ]
      subs[s,y,s] → s
    eval[s]
→ s
```

λ-Calculus Evaluation as Function

- A *normal order version of apply* is required for recursive functions

λ-Calculus Evaluation as Function

- A *normal order version of apply* is required for recursive functions
- Need to add accumulator variables to pass forward next identifier number
  - Not conceptually difficult, but messes up code

λ-Calculus Evaluation as Function

- A *normal order version of apply* is required for recursive functions
- Need to add accumulator variables to pass forward next identifier number
  - Not conceptually difficult, but messes up code
- And that's it:

λ-Calculus Evaluation as Function

- A *normal order version of apply* is required for recursive functions
- Need to add accumulator variables to pass forward next identifier number
  - Not conceptually difficult, but messes up code
- And that's it:
  - λ-calculus evalulation can be written as a λ-calculus expression

λ-Calculus Evaluation as Function

- A *normal order version of apply* is required for recursive functions
- Need to add accumulator variables to pass forward next identifier number
  - Not conceptually difficult, but messes up code
- And that's it:
  - λ-calculus evaluation can be written as a λ-calculus expression
  - Therefore, λ-calculus evaluation is just another function

λ-Calculus Evaluation as Function

- A *normal order version of apply* is required for recursive functions
- Need to add accumulator variables to pass forward next identifier number
  - Not conceptually difficult, but messes up code
- And that's it:
  - λ-calculus evalulation can be written as a λ-calculus expression
  - Therefore, λ-calculus evaluation is just another function
  - λ-calculus can be used to implement λ-calculus

Bootstrapping

- Functional languages can be written in abstract programming language

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
    - Simple to implement

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
  - Simple to implement
  - Easy to show correctness

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
  - Simple to implement
  - Easy to show correctness
- Easy to prototype a new language

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
  - Simple to implement
  - Easy to show correctness
- Easy to prototype a new language
  - Define translation from new language to abstract programming language

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
  - Simple to implement
  - Easy to show correctness
- Easy to prototype a new language
  - Define translation from new language to abstract programming language
  - Run new language on top of abstract programming layer

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
  - Simple to implement
  - Easy to show correctness
- Easy to prototype a new language
  - Define translation from new language to abstract programming language
  - Run new language on top of abstract programming layer
  - Write native code compiler in new language

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
  - Simple to implement
  - Easy to show correctness
- Easy to prototype a new language
  - Define translation from new language to abstract programming language
  - Run new language on top of abstract programming layer
  - Write native code compiler in new language
  - Now can compile new language directly to platform

Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to λ-calculus
- λ-calculus has simple syntax, evaluation rules and semantics
  - Simple to implement
  - Easy to show correctness
- Easy to prototype a new language
  - Define translation from new language to abstract programming language
  - Run new language on top of abstract programming layer
  - Write native code compiler in new language
  - Now can compile new language directly to platform
- Called bootstrapping

λ-Calculus Evaluation in Lisp

- Possible to implement λ-Calculus Evaluator in Lisp

λ-Calculus Evaluation in Lisp

- Possible to implement λ-Calculus Evaluator in Lisp
- But: Lisp has:
  - basic datatypes: numbers, lists, constants
  - a type system with predicates: 'atom', 'consp'
  - primitive functions: `+`, `-`, `cons`, `car`, `cdr`

λ-Calculus Evaluation in Lisp

- Possible to implement λ-Calculus Evaluator in Lisp
- But: Lisp has:
  - basic datatypes: numbers, lists, constants
  - a type system with predicates: 'atom', 'consp'
  - primitive functions: +, -, cons, car, cdr
- Can replace low-level λ-calculus idioms for numbers and lists with high-level Lisp implementations

λ-Calculus Evaluation in Lisp

- Possible to implement λ-Calculus Evaluator in Lisp
- But: Lisp has:
  - basic datatypes: numbers, lists, constants
  - a type system with predicates: 'atom', 'consp'
  - primitive functions: `+`, `-`, `cons`, `car`, `cdr`
- Can replace low-level λ-calculus idioms for numbers and lists with high-level Lisp implementations
- Do not need separate structure to represent type of data

λ-Calculus Evaluation in Lisp

- Possible to implement λ-Calculus Evaluator in Lisp
- But: Lisp has:
  - basic datatypes: numbers, lists, constants
  - a type system with predicates: 'atom', 'consp'
  - primitive functions: +, -, cons, car, cdr
- Can replace low-level λ-calculus idioms for numbers and lists with high-level Lisp implementations
- Do not need separate structure to represent type of data
- Requires
  - rewrite of creators, accessors and predicates
  - extra case in interpreter to intercept and call built-in functions directly
  - minor changes to other components

Efficiency Issues

- Consider the following example

```
(λx | IF T  THEN ((λy| (λz| y z) x) x)
            ELSE ((λy| y) x))              ) z
```

$\xrightarrow{\beta}$ `[ z/x ] IF T THEN ((λy| (λz| y z) x) x) ELSE ((λy| y)`

Efficiency Issues

- Consider the following example

```
(λx | IF T  THEN ((λy| (λz| y z) x) x)
            ELSE ((λy| y) x))               ) z
```

$\xrightarrow{\beta}$ `[ z/x ] IF T THEN ((λy| (λz| y z) x) x) ELSE ((λy| y)`

- Followed generic β-reduction. Notice anything odd?

Efficiency Issues

- Consider the following example

```
(λx | IF T  THEN ((λy| (λz| y z) x) x)
            ELSE ((λy| y) x))           ) z
```

$\xrightarrow{\beta}$ `[ z/x ] IF T THEN ((λy| (λz| y z) x) x) ELSE ((λy| y)`

- Followed generic β-reduction. Notice anything odd?
  - Substituted for both halves of IF statement even though ELSE is *never* used

Efficiency Issues

- Consider the following example

```
(λx | IF T  THEN ((λy| (λz| y z) x) x)
            ELSE ((λy| y) x))          ) z
```

$\xrightarrow{\beta}$ `[ z/x ] IF T THEN ((λy| (λz| y z) x) x) ELSE ((λy| y)`

- Followed generic β-reduction. Notice anything odd?
  - Substituted for both halves of IF statement even though ELSE is *never* used
  - Substitution involves rebuilding a copy of the expression

Efficiency Issues

- Consider the following example

```
(λx | IF T  THEN ((λy| (λz| y z) x) x)
            ELSE ((λy| y) x))             ) z
```

$\xrightarrow{\beta}$ `[ z/x ] IF T THEN ((λy| (λz| y z) x) x) ELSE ((λy| y)`

- Followed generic β-reduction. Notice anything odd?
  - Substituted for both halves of IF statement even though `ELSE` is *never* used
  - Substitution involves rebuilding a copy of the expression
    - `(λz| y z)` rebuilt even though no x

Efficiency Issues

- Consider the following example

```
(λx | IF T  THEN ((λy| (λz| y z) x) x)
            ELSE ((λy| y) x))              ) z
```

$\xrightarrow{\beta}$ `[ z/x ] IF T THEN ((λy| (λz| y z) x) x) ELSE ((λy| y)`

- Followed generic β-reduction. Notice anything odd?
  - Substituted for both halves of IF statement even though `ELSE` is *never* used
  - Substitution involves rebuilding a copy of the expression
    - `(λz| y z)` rebuilt even though no x
- In `(λx| ( λ y| ( λz | ⟨E⟩)))`, expression ⟨E⟩ is rebuilt 3 times!

Lazy Substitution

- How do we avoid redundant substitutions?

Lazy Substitution

- How do we avoid redundant substitutions?
  1. Note any parameter substitutions introduced by applications
     (keep in ordered list)

Lazy Substitution

- How do we avoid redundant substitutions?
  1. Note any parameter substitutions introduced by applications
     (keep in ordered list)
  2. Start processing the expression

Lazy Substitution

- How do we avoid redundant substitutions?
  1. Note any parameter substitutions introduced by applications
     (keep in ordered list)
  2. Start processing the expression
  3. Perform substitution only if parameter encountered

Binding Lists

- Bindings list are a simple approach to efficient substitution

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
```

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
```

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
→ IF T THEN (λz|y) ELSE (λz | z y)
```

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
→ IF T THEN (λz|y) ELSE (λz | z y)
eval[ IF T THEN (λz|y) ELSE (λz | z y) ] →(λz|y)
```

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
→ IF T THEN (λz|y) ELSE (λz | z y)
eval[ IF T THEN (λz|y) ELSE (λz | z y) ] →(λz|y)
```

- Smart substitution: eval until substitution is needed, then substitute

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
```

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
→ IF T THEN (λz|y) ELSE (λz | z y)
eval[ IF T THEN (λz|y) ELSE (λz | z y) ] →(λz|y)
```

- Smart substitution: eval until substitution is needed, then substitute

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
eval[ IF T THEN (λz|x) ELSE (λz | z x ), {x←y} ]
```

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
→ IF T THEN (λz|y) ELSE (λz | z y)
eval[ IF T THEN (λz|y) ELSE (λz | z y) ] →(λz|y)
```

- Smart substitution: eval until substitution is needed, then substitute

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
eval[ IF T THEN (λz|x) ELSE (λz | z x ), {x←y} ]
   eval[ T, {x←y}]
```

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
→ IF T THEN (λz|y) ELSE (λz | z y)
eval[ IF T THEN (λz|y) ELSE (λz | z y) ] →(λz|y)
```

- Smart substitution: eval until substitution is needed, then substitute

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
eval[ IF T THEN (λz|x) ELSE (λz | z x ), {x←y} ]
   eval[ T, {x←y}]
   eval[ (λz|x), {x←y}]
```

Binding Lists

- Bindings list are a simple approach to efficient substitution
- Naive eval: substitute everything first, then eval

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
[y/x] ( IF T THEN (λz|x) ELSE (λz| z x) )
→ IF T THEN (λz|y) ELSE (λz | z y)
eval[ IF T THEN (λz|y) ELSE (λz | z y) ] →(λz|y)
```

- Smart substitution: eval until substitution is needed, then substitute

```
eval[ ( (λx| IF T THEN (λz|x) ELSE (λz | z x ))  y)]
eval[ IF T THEN (λz|x) ELSE (λz | z x ), {x←y} ]
    eval[ T, {x←y}]
    eval[ (λz|x), {x←y}]
          eval[ x, {x←y}]→(λz|y)
```

Binding Parameters to Expressions

- Parameter value may in turn be an expression

```
eval[ (λx | (* 2 x)) (+ 3 2), {} ]
```

Binding Parameters to Expressions

- Parameter value may in turn be an expression

```
eval[ (λx | (* 2 x)) (+ 3 2), {} ]
  eval[ (* 2 x) , {x←(+ 3 2)} ]
```

Binding Parameters to Expressions

- Parameter value may in turn be an expression

```
eval[ (λx | (* 2 x)) (+ 3 2), {} ]
  eval[ (* 2 x) , {x←(+ 3 2)} ]
    eval[2,{x←(+ 3 2)}] → 2
```

Binding Parameters to Expressions

- Parameter value may in turn be an expression

```
eval[ (λx | (* 2 x)) (+ 3 2), {} ]
  eval[ (* 2 x) , {x←(+ 3 2)} ]
    eval[2,{x←(+ 3 2)}] → 2
    eval[x,{x←(+ 3 2)}]
```

Binding Parameters to Expressions

- Parameter value may in turn be an expression

```
eval[ (λx | (* 2 x)) (+ 3 2), {} ]
  eval[ (* 2 x) , {x←(+ 3 2)} ]
    eval[2,{x←(+ 3 2)}] → 2
    eval[x,{x←(+ 3 2)}]
       eval[ (+ 3 2) ] → 5
```

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

  `eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]`

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

```
eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]
  eval[ (λy |(+ x y)) 5, {x←3} ]
```

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

```
eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]
  eval[ (λy |(+ x y)) 5, {x←3} ]
    eval[ (+ x y), {y←5, x←3} ]
```

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

```
eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]
  eval[ (λy |(+ x y)) 5, {x←3} ]
    eval[ (+ x y), {y←5, x←3} ]
      eval[x, {y←5, x←3}]
```

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

```
eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]
  eval[ (λy |(+ x y)) 5, {x←3} ]
    eval[ (+ x y), {y←5, x←3} ]
      eval[x, {y←5, x←3}]
         eval[3]→ 3
```

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

```
eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]
  eval[ (λy |(+ x y)) 5, {x←3} ]
    eval[ (+ x y), {y←5, x←3} ]
      eval[x, {y←5, x←3}]
         eval[3]→ 3
      eval[y, {y←5, x←3}]
```

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

```
eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]
  eval[ (λy |(+ x y)) 5, {x←3} ]
    eval[ (+ x y), {y←5, x←3} ]
      eval[x, {y←5, x←3}]
         eval[3]→ 3
      eval[y, {y←5, x←3}]
         eval[5]→ 5
```

Bindings and Multiple Arguments

- Multiple bindings are added to bindings list in order of occurence

```
eval[ (λx | (λy |(+ x y)) ) 3 5, {} ]
  eval[ (λy |(+ x y)) 5, {x←3} ]
    eval[ (+ x y), {y←5, x←3} ]
      eval[x, {y←5, x←3}]
         eval[3]→ 3
      eval[y, {y←5, x←3}]
         eval[5]→ 5
    eval[ (+ 3 5) ]→5
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
       eval[x, {x←5, x←3} ] →5
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
       eval[x, {x←5, x←3} ] →5
    →10
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
       eval[x, {x←5, x←3} ] →5
    →10
  →10
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
       eval[x, {x←5, x←3} ] →5
    →10
  →10
  eval[ (+ 10 x), {x←3} ]
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
       eval[x, {x←5, x←3} ] →5
    →10
  →10
  eval[ (+ 10 x), {x←3} ]
    eval[10, {x←3}]→10
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
       eval[x, {x←5, x←3} ] →5
    →10
  →10
  eval[ (+ 10 x), {x←3} ]
    eval[10, {x←3}]→10
    eval[x, {x←3}] → 3
```

Bindings and Shadowed Arguments

- Bindings looked up from left to right. First value found is used

```
eval[ (λx | (+ ((λx |(+ x x)) 5) x) 3, {} ]
eval[ (+ ((λx |(+ x x)) 5) x), {x←3} ]
  eval[ ((λx |(+ x x)) 5), {x←3} ]
    eval[(+ x x), {x←5, x←3} ]
       eval[x, {x←5, x←3} ] →5
       eval[x, {x←5, x←3} ] →5
    →10
  →10
  eval[ (+ 10 x), {x←3} ]
    eval[10, {x←3}]→10
    eval[x, {x←3}] → 3
  →13
```

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
```

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate (λx| + x y) further

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate (λx| + x y) further
- But, should have y bound to 4

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate $(\lambda x| + x\ y)$ further
- But, should have y bound to 4
- Our simple interpreter actually handles this (but poorly):

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate $(\lambda x| + x\ y)$ further
- But, should have y bound to 4
- Our simple interpreter actually handles this (but poorly):
  - evaluate λ-body: $+\ x\ y$ in environment $(y \leftarrow 4)$

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate $(\lambda x| + x\ y)$ further
- But, should have y bound to 4
- Our simple interpreter actually handles this (but poorly):
  - evaluate λ-body: $+ x\ y$ in environment $(y \leftarrow 4)$
  - create new function with evaluated body $(\lambda x|\ \langle \text{BODY} \rangle)$

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate $(\lambda x| + x\ y)$ further
- But, should have y bound to 4
- Our simple interpreter actually handles this (but poorly):
  - evaluate λ-body: + x y in environment (y←4)
  - create new function with evaluated body $(\lambda x|\ \langle \text{BODY} \rangle)$

```
eval[+ x y, {y←4}] → (+ x 4)
```

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate $(\lambda x | + x\ y)$ further
- But, should have y bound to 4
- Our simple interpreter actually handles this (but poorly):
  - evaluate λ-body: $+$ x y in environment $(y \leftarrow 4)$
  - create new function with evaluated body $(\lambda x| \langle \text{BODY} \rangle)$

```
eval[+ x y, {y←4}] → (+ x 4)
→ (λx| + x 4)
```

Problems with Bindings and Free Variables I

```
eval[ (λy | (λx | + x y)) 4, {}]
eval[ (λx | + x y), {y←4}]
```

- No application here — cannot evaluate $(\lambda x| + x\ y)$ further
- But, should have y bound to 4
- Our simple interpreter actually handles this (but poorly):
  - evaluate λ-body: + x y in environment (y←4)
  - create new function with evaluated body $(\lambda x|\ \langle \text{BODY} \rangle)$

  ```
  eval[+ x y, {y←4}] → (+ x 4)
  → (λx| + x 4)
  ```
- Above solution breaks: See next slide!

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
```

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
```

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
  DO NOT DO THIS!
  → (λy | eval[ ( y y), {y←4} ]  )
```

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
  DO NOT DO THIS!
  → (λy | eval[ ( y y), {y←4} ]  )
   ≡ (λy | 4 4)
```

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
  DO NOT DO THIS!
  → (λy | eval[ ( y y), {y←4} ]  )
   ≡ (λy | 4 4)
```

- Dynamic binding results in wrong answer! The "funarg" problem

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
  DO NOT DO THIS!
  → (λy | eval[ ( y y), {y←4} ]  )
   ≡ (λy | 4 4)
```

- Dynamic binding results in wrong answer! The "funarg" problem
- Could try to represent fact that y is bound in inner λ

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
```

DO NOT DO THIS!

```
→ (λy | eval[ ( y y), {y←4} ]  )
 ≡ (λy | 4 4)
```

- Dynamic binding results in wrong answer! The "funarg" problem
- Could try to represent fact that y is bound in inner λ

```
eval[ (λy| (y y)), {y←4} ]
```

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
  DO NOT DO THIS!
  → (λy | eval[ ( y y), {y←4} ]  )
   ≡ (λy | 4 4)
```

- Dynamic binding results in wrong answer! The "funarg" problem
- Could try to represent fact that y is bound in inner λ

```
eval[ (λy| (y y)), {y←4} ]
    DO NOT DO THIS!
   → (λy | eval[ ( y y), {y←y, y←4} ]  )
```

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
  DO NOT DO THIS!
  → (λy | eval[ ( y y), {y←4} ]  )
   ≡ (λy | 4 4)
```

- Dynamic binding results in wrong answer! The "funarg" problem
- Could try to represent fact that y is bound in inner λ

```
eval[ (λy| (y y)), {y←4} ]
    DO NOT DO THIS!
   → (λy | eval[ ( y y), {y←y, y←4} ]  )
   → (λy | ( y y))
```

Problems with Bindings and Free Variables II

```
eval[ (λy | (λy| (y y))) 4, {} ]
eval[ (λy| (y y)), {y←4} ]
  DO NOT DO THIS!
  → (λy | eval[ ( y y), {y←4} ]  )
   ≡ (λy | 4 4)
```

- Dynamic binding results in wrong answer! The "funarg" problem
- Could try to represent fact that y is bound in inner λ

```
eval[ (λy| (y y)), {y←4} ]
    DO NOT DO THIS!
   → (λy | eval[ ( y y), {y←y, y←4} ]  )
   → (λy | ( y y))
```

- *Solution might break in more complex case - not sure at this point*

Closures

- The set of bindings that are active for a definition is called its *environment* or *context*

Closures

- The set of bindings that are active for a definition is called its *environment* or *context*
- An expression is "executed in" an environment

Closures

- The set of bindings that are active for a definition is called its *environment* or *context*
- An expression is "executed in" an environment
- An expression together with its environment is called a *closure*

Closures

- The set of bindings that are active for a definition is called its *environment* or *context*
- An expression is "executed in" an environment
- An expression together with its environment is called a *closure*
- <closure>={expression, environment}

Closures

- The set of bindings that are active for a definition is called its *environment* or *context*
- An expression is "executed in" an environment
- An expression together with its environment is called a *closure*
- <closure>={expression, environment}
- By saving a closure with a λ we can ensure it evaluates to the same thing whenever and wherever it is executed

Closures

- The set of bindings that are active for a definition is called its *environment* or *context*
- An expression is "executed in" an environment
- An expression together with its environment is called a *closure*
- <closure>={expression, environment}
- By saving a closure with a λ we can ensure it evaluates to the same thing whenever and wherever it is executed
- Should be no free variables in a closure

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```

Regular apply: eval f1, eval a1, apply f1 to a1

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```

Regular apply: eval f1, eval a1, apply f1 to a1

```
  f1 = eval[ (λx | x) ,{}]
```

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```
Regular apply: eval f1, eval a1, apply f1 to a1

```
f1 = eval[ (λx | x) ,{}]
```
Definition:make closure

Simple Application with Closures

eval[(λx | x) 2 ,{}]

Regular apply: eval f1, eval a1, apply f1 to a1

f1 = eval[(λx | x) ,{}]

Definition:make closure

f1 = <(λx | x) ,{}>

Simple Application with Closures

eval[(λx | x) 2 ,{}]

Regular apply: eval f1, eval a1, apply f1 to a1

f1 = eval[(λx | x) ,{}]

Definition:make closure

f1 = <(λx | x) ,{}>

a1 = eval[2] = 2

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```

Regular apply: eval f1, eval a1, apply f1 to a1

```
f1 = eval[ (λx | x) ,{}]
```

Definition:make closure

```
f1 = <(λx | x) ,{}>

a1 = eval[ 2 ] = 2

apply[ f1, a1 ]
```

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```

Regular apply: eval f1, eval a1, apply f1 to a1

```
f1 = eval[ (λx | x) ,{}]
```

Definition:make closure

```
f1 = <(λx | x) ,{}>
```

```
a1 = eval[ 2 ] = 2
```

```
apply[ f1, a1 ]
```

Eval f1 body in environment

with x=a1 and context of f1={}

Simple Application with Closures

eval[(λx | x) 2 ,{}]

Regular apply: eval f1, eval a1, apply f1 to a1

f1 = eval[(λx | x) ,{}]

Definition:make closure

f1 = <(λx | x) ,{}>

a1 = eval[2] = 2

apply[f1, a1]

Eval f1 body in environment

with x=a1 and context of f1={}

eval[x, {x←2}+{}]

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```
Regular apply: eval f1, eval a1, apply f1 to a1

```
f1 = eval[ (λx | x) ,{}]
```
Definition:make closure
```
f1 = <(λx | x) ,{}>
```

```
a1 = eval[ 2 ] = 2
```

```
apply[ f1, a1 ]
```
Eval f1 body in environment
with x=a1 and context of f1={}
```
eval[ x, {x←2}+{} ]
```

→2

Simple Application with Closures

```
eval[ (λx | x) 2 ,{}]
```
Regular apply: eval f1, eval a1, apply f1 to a1

```
  f1 = eval[ (λx | x) ,{}]
```
Definition:make closure
```
  f1 = <(λx | x) ,{}>

  a1 = eval[ 2 ] = 2

  apply[ f1, a1 ]
```
Eval f1 body in environment

with x=a1 and context of f1={}
```
  eval[ x, {x←2}+{} ]
```

$\rightarrow$2

- Seems like extra machinery, but useful in complex cases

Forming and Applying Closures

- Forming closures

Forming and Applying Closures

- Forming closures
  - Given definition (λp $|\langle$BODY$\rangle$) defined in environment E

Forming and Applying Closures

- Forming closures
  - Given definition (λp |⟨BODY⟩) defined in environment E
  - We form the closure <(λp |⟨BODY⟩),E>

Forming and Applying Closures

- Forming closures
  - Given definition (λp |$\langle$BODY$\rangle$) defined in environment E
  - We form the closure <(λp |$\langle$BODY$\rangle$),E>
- To apply closure <(λp |$\langle$BODY$\rangle$),E> to argument A in context G

Forming and Applying Closures

- Forming closures
  - Given definition (λp |⟨BODY⟩) defined in environment E
  - We form the closure <(λp |⟨BODY⟩),E>
- To apply closure <(λp |⟨BODY⟩),E> to argument A in context G
  - evaluate ⟨BODY⟩

Forming and Applying Closures

- Forming closures
  - Given definition $(\lambda p \mid \langle BODY \rangle)$ defined in environment E
  - We form the closure $<(\lambda p \mid \langle BODY \rangle),E>$
- To apply closure $<(\lambda p \mid \langle BODY \rangle),E>$ to argument A in context G
  - evaluate $\langle BODY \rangle$
  - in an environment $= \{ p \leftarrow A + E + G \}$

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
```

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Regular apply, eval f1, eval a1, apply f1 to a1

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Regular apply, eval f1, eval a1, apply f1 to a1

```
  f1=eval[(λx|(λy|(y 3)) (λz|z+x)), {}]
```

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Regular apply, eval f1, eval a1, apply f1 to a1

```
f1=eval[(λx|(λy|(y 3)) (λz|z+x)), {}]
```

Definition:make closure

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Regular apply, eval f1, eval a1, apply f1 to a1

```
  f1=eval[(λx|(λy|(y 3)) (λz|z+x)), {}]
```

Definition:make closure

```
  f1=<(λx|(λy|(y 3)) (λz|z+x)),{}>
```

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Regular apply, eval f1, eval a1, apply f1 to a1

```
  f1=eval[(λx|(λy|(y 3)) (λz|z+x)), {}]
```

Definition:make closure

```
  f1=<(λx|(λy|(y 3)) (λz|z+x)),{}>
  a1=eval[ 1, {}] = 1
```

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Regular apply, eval f1, eval a1, apply f1 to a1

```
  f1=eval[(λx|(λy|(y 3)) (λz|z+x)), {}]
```

Definition:make closure

```
  f1=<(λx|(λy|(y 3)) (λz|z+x)),{}>
  a1=eval[ 1, {}] = 1
  apply(f1,a1)
```

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Regular apply, eval f1, eval a1, apply f1 to a1

```
  f1=eval[(λx|(λy|(y 3)) (λz|z+x)), {}]
```

Definition:make closure

```
  f1=<(λx|(λy|(y 3)) (λz|z+x)),{}>
  a1=eval[ 1, {}] = 1
  apply(f1,a1)
```

Eval f1 body with a1 and context of f1

Trickier Application with Closures I

```
LET x=1 IN LET y=(λz|z+x) IN y(3)
eval[(λx|(λy|(y 3)) (λz|z+x)) 1, {}]
```

Regular apply, eval f1, eval a1, apply f1 to a1

```
  f1=eval[(λx|(λy|(y 3)) (λz|z+x)), {}]
```

Definition:make closure

```
  f1=<(λx|(λy|(y 3)) (λz|z+x)),{}>
  a1=eval[ 1, {}] = 1
  apply(f1,a1)
```

Eval f1 body with a1 and context of f1

```
  eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Regular apply, eval f2, eval a2, apply f2 to a2

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Regular apply, eval f2, eval a2, apply f2 to a2

```
f2=eval[(λy|(y 3)),{x=1}]
```

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```
Regular apply, eval f2, eval a2, apply f2 to a2
```
f2=eval[(λy|(y 3)),{x=1}]
```
Definition:make closure

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Regular apply, eval f2, eval a2, apply f2 to a2

```
f2=eval[(λy|(y 3)),{x=1}]
```

Definition:make closure

```
f2=<(λy|(y 3)),{x=1}>
```

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Regular apply, eval f2, eval a2, apply f2 to a2

```
f2=eval[(λy|(y 3)),{x=1}]
```

Definition:make closure

```
f2=<(λy|(y 3)),{x=1}>
a2=eval[(λz|z+x) ,{x=1}]
```

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
   Regular apply,  eval f2,  eval a2,  apply  f2 to  a2
   f2=eval[(λy|(y 3)),{x=1}]
   Definition:make closure
   f2=<(λy|(y 3)),{x=1}>
   a2=eval[(λz|z+x) ,{x=1}]
   Definition:make closure
```

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Regular apply, eval f2, eval a2, apply f2 to a2

```
f2=eval[(λy|(y 3)),{x=1}]
```

Definition:make closure

```
f2=<(λy|(y 3)),{x=1}>
a2=eval[(λz|z+x) ,{x=1}]
```

Definition:make closure

```
a2=<(λz|z+x) ,{x=1}>
```

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Regular apply, eval f2, eval a2, apply f2 to a2

```
f2=eval[(λy|(y 3)),{x=1}]
```

Definition:make closure

```
f2=<(λy|(y 3)),{x=1}>
a2=eval[(λz|z+x) ,{x=1}]
```

Definition:make closure

```
a2=<(λz|z+x) ,{x=1}>
apply(f2,a2)
```

Trickier Application with Closures II

`eval[(λy|(y 3)) (λz|z+x) ,{x=1}]`

Regular apply, eval f2, eval a2, apply f2 to a2

`f2=eval[(λy|(y 3)),{x=1}]`

Definition:make closure

`f2=<(λy|(y 3)),{x=1}>`

`a2=eval[(λz|z+x) ,{x=1}]`

Definition:make closure

`a2=<(λz|z+x) ,{x=1}>`

`apply(f2,a2)`

Eval f2 body with a2 and context of f2

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Regular apply, eval f2, eval a2, apply f2 to a2

```
f2=eval[(λy|(y 3)),{x=1}]
```

Definition:make closure

```
f2=<(λy|(y 3)),{x=1}>
a2=eval[(λz|z+x) ,{x=1}]
```

Definition:make closure

```
a2=<(λz|z+x) ,{x=1}>
apply(f2,a2)
```

Eval f2 body with a2 and context of f2

a2 is a closure— parm y is bound to a closure

Trickier Application with Closures II

```
eval[(λy|(y 3)) (λz|z+x) ,{x=1}]
```

Regular apply, eval f2, eval a2, apply f2 to a2

```
f2=eval[(λy|(y 3)),{x=1}]
```

Definition:make closure

```
f2=<(λy|(y 3)),{x=1}>
a2=eval[(λz|z+x) ,{x=1}]
```

Definition:make closure

```
a2=<(λz|z+x) ,{x=1}>
apply(f2,a2)
```

Eval f2 body with a2 and context of f2

a2 is a closure— parm y is bound to a closure

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```
Regular apply, eval f3, eval a3, apply f3 to a3
```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
  apply[f3,a3]
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
  apply[f3,a3]
```

Eval f2 body with a2 and context of f2

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

```
f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
f3=<(λz|z+x),{x=1}>
a3=eval[3]=3
apply[f3,a3]
```

Eval f2 body with a2 and context of f2

```
Eval[z+x,{z=3,x=1}]
```

Trickier Application with Closures III

`eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]`

Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
  apply[f3,a3]
```

  Eval f2 body with a2 and context of f2

```
    Eval[z+x,{z=3,x=1}]
```

    Regular apply...

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
  apply[f3,a3]
```

Eval f2 body with a2 and context of f2

```
    Eval[z+x,{z=3,x=1}]
```

Regular apply...

```
       f4=eval[z,{z=3,x=1}]=3
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```
Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
  apply[f3,a3]
```
Eval f2 body with a2 and context of f2

```
    Eval[z+x,{z=3,x=1}]
```
Regular apply…

```
        f4=eval[z,{z=3,x=1}]=3
        a4=eval[x,{z=3,x=1}]=1
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
  apply[f3,a3]
```

Eval f2 body with a2 and context of f2

```
    Eval[z+x,{z=3,x=1}]
```

Regular apply...

```
      f4=eval[z,{z=3,x=1}]=3
      a4=eval[x,{z=3,x=1}]=1
      apply[f4,a4]
```

Trickier Application with Closures III

```
eval[(y 3),{y=<(λz|z+x),{x=1}>,x=1}]
```

Regular apply, eval f3, eval a3, apply f3 to a3

```
  f3=eval[y,{y=<(λz|z+x),{x=1}>,x=1}]
  f3=<(λz|z+x),{x=1}>
  a3=eval[3]=3
  apply[f3,a3]
```

Eval f2 body with a2 and context of f2

```
    Eval[z+x,{z=3,x=1}]
```

Regular apply...

```
       f4=eval[z,{z=3,x=1}]=3
       a4=eval[x,{z=3,x=1}]=1
       apply[f4,a4]
       eval[+ 3 1] → 4
```

Other Uses for Closures

- Closures can be used for creating delayed computations
  - Delay and force predicates covered earlier

Other Uses for Closures

- Closures can be used for creating delayed computations
  - Delay and force predicates covered earlier
- Making recursion more efficient

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y
- Normal order is inefficient in general - but suppose we use it

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y
- Normal order is inefficient in general - but suppose we use it
- Bindings evaluate Fixed-Point Combinator correcty

  F$\equiv$(λf | (λn | zerop(n) 0 f(n-1)))

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y

- Normal order is inefficient in general - but suppose we use it

- Bindings evaluate Fixed-Point Combinator correcty

  F≡(λf | (λn | zerop(n) 0 f(n-1)))
  Y≡(λf | (λx| f (x x)) (λx| f (x x)))

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y
- Normal order is inefficient in general - but suppose we use it
- Bindings evaluate Fixed-Point Combinator correcty

```
F≡(λf | (λn | zerop(n) 0 f(n-1)))
Y≡(λf | (λx| f (x x))  (λx| f (x x)) )
eval[YF,{}]
```

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y

- Normal order is inefficient in general - but suppose we use it
- Bindings evaluate Fixed-Point Combinator correcty

```
F≡(λf | (λn | zerop(n) 0 f(n-1)))
Y≡(λf | (λx| f (x x))  (λx| f (x x)) )
eval[YF,{}]
eval[(λf | (λx| f (x x))  (λx| f (x x)) ) F,{}]
```

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y
- Normal order is inefficient in general - but suppose we use it
- Bindings evaluate Fixed-Point Combinator correcty

```
F≡(λf | (λn | zerop(n) 0 f(n-1)))
Y≡(λf | (λx| f (x x))  (λx| f (x x)) )
eval[YF,{}]
eval[(λf | (λx| f (x x))  (λx| f (x x)) ) F,{}]
eval[(λx| f (x x))  (λx| f (x x)), {f←F}]
```

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y
- Normal order is inefficient in general - but suppose we use it
- Bindings evaluate Fixed-Point Combinator correcty

```
F≡(λf | (λn | zerop(n) 0 f(n-1)))
Y≡(λf | (λx| f (x x))  (λx| f (x x)) )
eval[YF,{}]
eval[(λf | (λx| f (x x))  (λx| f (x x)) ) F,{}]
eval[(λx| f (x x))  (λx| f (x x)), {f←F}]
⋮
→(λx| F (x x))  (λx| F (x x))
```

Bindings and Recursion I

- Applicative order reduction blows up with Combinator Y
- Normal order is inefficient in general - but suppose we use it
- Bindings evaluate Fixed-Point Combinator correcty

$$
\begin{array}{l}
\texttt{F}\equiv(\lambda\texttt{f | (}\lambda\texttt{n | zerop(n) 0 f(n-1)))} \\
\texttt{Y}\equiv(\lambda\texttt{f | (}\lambda\texttt{x| f (x x)) (}\lambda\texttt{x| f (x x)))} \\
\texttt{eval[YF,\{\}]} \\
\texttt{eval[(}\lambda\texttt{f | (}\lambda\texttt{x| f (x x)) (}\lambda\texttt{x| f (x x))) F,\{\}]} \\
\texttt{eval[(}\lambda\texttt{x| f (x x)) (}\lambda\texttt{x| f (x x)), \{f}\leftarrow\texttt{F\}]} \\
\vdots \\
\rightarrow\texttt{(}\lambda\texttt{x| F (x x)) (}\lambda\texttt{x| F (x x))} \\
\equiv\langle\texttt{YF}\rangle
\end{array}
$$

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
```

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
eval[ (λn | zerop(n) 0 f(n-1)) 1, {f←⟨YF⟩}]
```

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
eval[ (λn | zerop(n) 0 f(n-1)) 1, {f←⟨YF⟩}]
eval[ zerop(n) 0 f(n-1), {n←1,f←⟨YF⟩}]
```

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
eval[ (λn | zerop(n) 0 f(n-1)) 1, {f←⟨YF⟩}]
eval[ zerop(n) 0 f(n-1), {n←1,f←⟨YF⟩}]
   eval[ zerop(n), {n←1,f←⟨YF⟩}] → F
```

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
eval[ (λn | zerop(n) 0 f(n-1)) 1, {f←⟨YF⟩}]
eval[ zerop(n) 0 f(n-1), {n←1,f←⟨YF⟩}]
   eval[ zerop(n), {n←1,f←⟨YF⟩}] → F
   eval[ f(n-1), {n←1,f←⟨YF⟩}]
```

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
eval[ (λn | zerop(n) 0 f(n-1)) 1, {f←⟨YF⟩}]
eval[ zerop(n) 0 f(n-1), {n←1,f←⟨YF⟩}]
   eval[ zerop(n), {n←1,f←⟨YF⟩}] → F
   eval[ f(n-1), {n←1,f←⟨YF⟩}]
     eval[ ⟨YF⟩, {n←1,f←⟨YF⟩}] → F ⟨YF⟩
```

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
eval[ (λn | zerop(n) 0 f(n-1)) 1, {f←⟨YF⟩}]
eval[ zerop(n) 0 f(n-1), {n←1,f←⟨YF⟩}]
   eval[ zerop(n), {n←1,f←⟨YF⟩}] → F
   eval[ f(n-1), {n←1,f←⟨YF⟩}]
     eval[ ⟨YF⟩, {n←1,f←⟨YF⟩}] → F ⟨YF⟩
     eval[ n-1, {n←1,f←⟨YF⟩} ] → 0
```

Bindings and Recursion II

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 1, {}]
eval[ (λn | zerop(n) 0 f(n-1)) 1, {f←⟨YF⟩}]
eval[ zerop(n) 0 f(n-1), {n←1,f←⟨YF⟩}]
   eval[ zerop(n), {n←1,f←⟨YF⟩}] → F
   eval[ f(n-1), {n←1,f←⟨YF⟩}]
     eval[ ⟨YF⟩, {n←1,f←⟨YF⟩}] → F ⟨YF⟩
     eval[ n-1, {n←1,f←⟨YF⟩} ] → 0
   eval[ F <YF> 0, {n←1,f←⟨YF⟩} ]
```

Bindings and Recursion III

- Process repeats

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 0,
         {n←1,f←⟨YF⟩} ]
```

Bindings and Recursion III

- Process repeats

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 0,
        {n←1,f←⟨YF⟩} ]

   eval[ (λn | zerop(n) 0 f(n-1)) 0,
        {f←⟨YF⟩,n←1,f←⟨YF⟩}]
```

Bindings and Recursion III

- Process repeats

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 0,
       {n←1,f←⟨YF⟩} ]

   eval[ (λn | zerop(n) 0 f(n-1)) 0,
       {f←⟨YF⟩,n←1,f←⟨YF⟩}]

   eval[ zerop(n) 0 f(n-1) 0,
       {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
```

Bindings and Recursion III

- Process repeats

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 0,
      {n←1,f←⟨YF⟩} ]

  eval[ (λn | zerop(n) 0 f(n-1)) 0,
      {f←⟨YF⟩,n←1,f←⟨YF⟩}]

  eval[ zerop(n) 0 f(n-1) 0,
      {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
    eval[ zerop(n), {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
```

Bindings and Recursion III

- Process repeats

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 0,
      {n←1,f←⟨YF⟩} ]

   eval[ (λn | zerop(n) 0 f(n-1)) 0,
      {f←⟨YF⟩,n←1,f←⟨YF⟩}]

   eval[ zerop(n) 0 f(n-1) 0,
      {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
      eval[ zerop(n), {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
      eval[ zerop(0),
         {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}] →0
```

Bindings and Recursion III

- Process repeats

```
eval[ (λf | (λn | zerop(n) 0 f(n-1))) ⟨YF⟩ 0,
      {n←1,f←⟨YF⟩} ]

    eval[ (λn | zerop(n) 0 f(n-1)) 0,
          {f←⟨YF⟩,n←1,f←⟨YF⟩}]

    eval[ zerop(n) 0 f(n-1) 0,
          {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
        eval[ zerop(n), {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}]
        eval[ zerop(0),
              {n←0,f←⟨YF⟩,n←1,f←⟨YF⟩}] →0
    eval[0] →0
```

Closures and Recursion I

- With normal order, we may have to eval args many times

Closures and Recursion I

- With normal order, we may have to eval args many times
- Difficult to make use of specialized primitives

Closures and Recursion I

- With normal order, we may have to eval args many times
- Difficult to make use of specialized primitives
  - Lambda-calculus expressions can be reduced in normal or applicative order. The order of reductions does not matter

$$
\begin{aligned}
& (\lambda x \mid x)\ (\lambda y \mid 3)\ 1 \\
\equiv\ & [(\lambda y \mid 3)/x]\ x\ 1 \\
\equiv\ & (\lambda y \mid 3)\ 1 \\
\equiv\ & 3
\end{aligned}
$$

Closures and Recursion I

- With normal order, we may have to eval args many times
- Difficult to make use of specialized primitives
  - Lambda-calculus expressions can be reduced in normal or applicative order. The order of reductions does not matter

    $(\lambda x \mid x)\ (\lambda y \mid 3)\ 1$
    $\equiv [(\lambda y \mid 3)/x]\ x\ 1$
    $\equiv (\lambda y \mid 3)\ 1$
    $\equiv 3$
  - Normal order passes unreduced arguments to functions

    ```
    '+ f(y)=3 1
    ```

Closures and Recursion I

- With normal order, we may have to eval args many times
- Difficult to make use of specialized primitives
  - Lambda-calculus expressions can be reduced in normal or applicative order. The order of reductions does not matter

    $(\lambda$x | x) $(\lambda$y | 3) 1
    $\equiv$ [$(\lambda$y | 3)/x] x 1
    $\equiv$ $(\lambda$y | 3) 1
    $\equiv$ 3

  - Normal order passes unreduced arguments to functions

    ```
    '+ f(y)=3 1
    ```

  - Efficient specialized functions cannot accept arbitrary expressions as arguments

Closures and Recursion I

- With normal order, we may have to eval args many times
- Difficult to make use of specialized primitives
  - Lambda-calculus expressions can be reduced in normal or applicative order. The order of reductions does not matter

    $(\lambda x \mid x)\ (\lambda y \mid 3)\ 1$
    $\equiv [(\lambda y \mid 3)/x]\ x\ 1$
    $\equiv (\lambda y \mid 3)\ 1$
    $\equiv 3$

  - Normal order passes unreduced arguments to functions

    ```
    '+ f(y)=3 1
    ```

  - Efficient specialized functions cannot accept arbitrary expressions as arguments
  - The '+ function cannot accept f(y)=3 as an argument, it only works on numbers

Closures and Recursion I

- With normal order, we may have to eval args many times
- Difficult to make use of specialized primitives
  - Lambda-calculus expressions can be reduced in normal or applicative order. The order of reductions does not matter

```
(λx | x) (λy | 3) 1
≡ [(λy | 3)/x] x 1
≡ (λy | 3) 1
≡ 3
```

  - Normal order passes unreduced arguments to functions

```
'+ f(y)=3 1
```

  - Efficient specialized functions cannot accept arbitrary expressions as arguments
  - The '+ function cannot accept f(y)=3 as an argument, it only works on numbers
- We end up with many copies of the function in the environment

Closures and Recursion II

- Closures can be used to
  - implement recursion with applicative order reduction
  - eliminate duplicate copies of functions

Closures and Recursion II

- Closures can be used to
  - implement recursion with applicative order reduction
  - eliminate duplicate copies of functions
- Closures permit delayed execution
  - Recursive function calls create a closure
  - Execute closure only if result is actually required

Closures and Recursion II

- Closures can be used to
  - implement recursion with applicative order reduction
  - eliminate duplicate copies of functions
- Closures permit delayed execution
  - Recursive function calls create a closure
  - Execute closure only if result is actually required
- Every instance of a recuring evaluation uses the same closure
  - The body is the same
  - The lexical definition is the same, so environments are the same

Closures and Recursion II

- Closures can be used to
  - implement recursion with applicative order reduction
  - eliminate duplicate copies of functions
- Closures permit delayed execution
  - Recursive function calls create a closure
  - Execute closure only if result is actually required
- Every instance of a recuring evaluation uses the same closure
  - The body is the same
  - The lexical definition is the same, so environments are the same
- Imperatively modify closure so that it points to itself

Closures and Recursion II

- Closures can be used to
  - implement recursion with applicative order reduction
  - eliminate duplicate copies of functions
- Closures permit delayed execution
  - Recursive function calls create a closure
  - Execute closure only if result is actually required
- Every instance of a recuring evaluation uses the same closure
  - The body is the same
  - The lexical definition is the same, so environments are the same
- Imperatively modify closure so that it points to itself
- Imperative operation is internal so it does not affect referential transparency

Closures and Recursion III

- Assume we change each application to a closure before application

Closures and Recursion III

- Assume we change each application to a closure before application

  E≡LETREC f=⟨BODY⟩ IN ⟨EXPR⟩

Closures and Recursion III

- Assume we change each application to a closure before application

```
E≡LETREC f=⟨BODY⟩ IN ⟨EXPR⟩

C ≡ <⟨BODY⟩,{f←C}>
```

Closures and Recursion III

- Assume we change each application to a closure before application

$\texttt{E}\equiv\texttt{LETREC f=}\langle\texttt{BODY}\rangle\texttt{ IN }\langle\texttt{EXPR}\rangle$

$\texttt{C} \equiv \texttt{<}\langle\texttt{BODY}\rangle\texttt{,\{f}\leftarrow\texttt{C\}>}$

$\texttt{E} \equiv\texttt{<}\langle\texttt{EXPR}\rangle\texttt{,\{f}\leftarrow\texttt{C\}>}$

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)
```

```
C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
eval[ <(z 1), {z←C}>, {}]
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
eval[ <(z 1), {z←C}>, {}]
eval[ (z 1), {z←C}]   ;; application (f1 a1)
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
eval[ <(z 1), {z←C}>, {}]
eval[ (z 1), {z←C}]   ;; application (f1 a1)
  f1=eval[ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>, {z←C}]
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
eval[ <(z 1), {z←C}>, {}]
eval[ (z 1), {z←C}]   ;; application (f1 a1)
  f1=eval[ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>, {z←C}]
  f1=<(λn|IF zerop(n) 0 z(n-1)), {z←C}>
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
eval[ <(z 1), {z←C}>, {}]
eval[ (z 1), {z←C}]   ;; application (f1 a1)
  f1=eval[ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>, {z←C}]
  f1=<(λn|IF zerop(n) 0 z(n-1)), {z←C}>
  a1=eval[1]=1
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
eval[ <(z 1), {z←C}>, {}]
eval[ (z 1), {z←C}]   ;; application (f1 a1)
  f1=eval[ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>, {z←C}]
  f1=<(λn|IF zerop(n) 0 z(n-1)), {z←C}>
  a1=eval[1]=1
  apply[f1,a1]
  applying a closure, get body, add parm to env
```

Closures and Recursion IV

```
LETREC z(n)=IF zerop(n) 0 z(n-1) IN z(1)

C ≡ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>

E ≡ <(z 1), {z←C}>

eval[E,{}]
eval[ <(z 1), {z←C}>, {}]
eval[ (z 1), {z←C}]   ;; application (f1 a1)
  f1=eval[ <(λn|IF zerop(n) 0 z(n-1)), {z←C}>, {z←C}]
  f1=<(λn|IF zerop(n) 0 z(n-1)), {z←C}>
  a1=eval[1]=1
  apply[f1,a1]
  applying a closure, get body, add parm to env
  eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Application:evaluate the arguments

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Application:evaluate the arguments

```
  eval[ zerop(n) , {n←1,z←C} ]→F
```

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Application:evaluate the arguments

```
eval[ zerop(n) , {n←1,z←C} ]→F
eval[ 0,{n←1,z←C} ] → 0
```

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Application:evaluate the arguments

```
  eval[ zerop(n) , {n←1,z←C} ]→F
  eval[ 0,{n←1,z←C} ] → 0
  eval[ z(n-1) ,{n←1,z←C}]
```

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Application:evaluate the arguments

```
eval[ zerop(n) , {n←1,z←C} ]→F
eval[ 0,{n←1,z←C} ] → 0
eval[ z(n-1) ,{n←1,z←C}]
```

Application:evaluate the arguments

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```
Application:evaluate the arguments
```
  eval[ zerop(n) , {n←1,z←C} ]→F
  eval[ 0,{n←1,z←C} ] → 0
  eval[ z(n-1) ,{n←1,z←C}]
```
Application:evaluate the arguments
```
    eval[ n-1,{n←1,z←C} ] → 0
```

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
  Application:evaluate the arguments
  eval[ zerop(n) , {n←1,z←C} ]→F
  eval[ 0,{n←1,z←C} ] → 0
  eval[ z(n-1) ,{n←1,z←C}]
  Application:evaluate the arguments
    eval[ n-1,{n←1,z←C} ] → 0
    → <z, {n←0,n←1,z←C}>
```

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Application:evaluate the arguments

```
eval[ zerop(n) , {n←1,z←C} ]→F
eval[ 0,{n←1,z←C} ] → 0
eval[ z(n-1) ,{n←1,z←C}]
```

Application:evaluate the arguments

```
  eval[ n-1,{n←1,z←C} ] → 0
  → <z, {n←0,n←1,z←C}>
```

Since zerop evaluates to false, evaluate recursive term

```
eval[ <z, {n←0,n←1,z←C}> ]
```

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Application:evaluate the arguments

```
eval[ zerop(n) , {n←1,z←C} ]→F
eval[ 0,{n←1,z←C} ] → 0
eval[ z(n-1) ,{n←1,z←C}]
```

Application:evaluate the arguments

```
  eval[ n-1,{n←1,z←C} ] → 0
  → <z, {n←0,n←1,z←C}>
```

Since zerop evaluates to false, evaluate recursive term

```
eval[ <z, {n←0,n←1,z←C}> ]
```

z will get value from context again.

Recursion emerges from self-reference

Closures and Recursion V

```
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]
```

Application:evaluate the arguments

```
eval[ zerop(n) , {n←1,z←C} ]→F
eval[ 0,{n←1,z←C} ] → 0
eval[ z(n-1) ,{n←1,z←C}]
```

Application:evaluate the arguments

```
  eval[ n-1,{n←1,z←C} ] → 0
  → <z, {n←0,n←1,z←C}>
```

Since zerop evaluates to false, evaluate recursive term

```
eval[ <z, {n←0,n←1,z←C}> ]
```

z will get value from context again.

Recursion emerges from self-reference

```
eval[IF zerop(n) 0 z(n-1), {n←0,n←1,z←C}> ]
```

Meta-interpretation of Lisp

- Abstract Programming gives structure to a λ-calculus formula

```
(IF zerop(n) THEN 1 ELSE (2*n))
```

Meta-interpretation of Lisp

- Abstract Programming gives structure to a λ-calculus formula

  `(IF zerop(n) THEN 1 ELSE (2*n))`

- Structure is illusory: translated into a large tangled λ-calculus expression

Meta-interpretation of Lisp

- Abstract Programming gives structure to a λ-calculus formula

  `(IF zerop(n) THEN 1 ELSE (2*n))`
- Structure is illusory: translated into a large tangled λ-calculus expression
- So we need general mechanisms like closures to help with recursion

Meta-interpretation of Lisp

- Abstract Programming gives structure to a λ-calculus formula

  `(IF zerop(n) THEN 1 ELSE (2*n))`
- Structure is illusory: translated into a large tangled λ-calculus expression
- So we need general mechanisms like closures to help with recursion
- In Lisp, the structure is explicit as an IF statement is represented explicitly as an s-expression

Meta-interpretation of Lisp

- Abstract Programming gives structure to a λ-calculus formula

```
(IF zerop(n) THEN 1 ELSE (2*n))
```

- Structure is illusory: translated into a large tangled λ-calculus expression
- So we need general mechanisms like closures to help with recursion
- In Lisp, the structure is explicit as an IF statement is represented explicitly as an s-expression
- In Lisp, we can treat IF as a special case, executing only the predicate and the appropriate clause

Meta-interpretation of Lisp

- Abstract Programming gives structure to a λ-calculus formula

  `(IF zerop(n) THEN 1 ELSE (2*n))`
- Structure is illusory: translated into a large tangled λ-calculus expression
- So we need general mechanisms like closures to help with recursion
- In Lisp, the structure is explicit as an IF statement is represented explicitly as an s-expression
- In Lisp, we can treat IF as a special case, executing only the predicate and the appropriate clause
- Can reserce closures in LISP for
  - function definitions to preserve lexical scope
  - implementation of lazy evaluation