COMPUT325: Meta-interpretation

Dr. B. Price and Dr. R. Greiner

26th October 2004
Introduction

- \( \lambda \)-calculus fully expresses computations of any programming language

Is \( \lambda \)-calculus sufficient to express itself?
Introduction

- $\lambda$-calculus fully expresses computations of any programming language

- Is $\lambda$-calculus sufficiently expressive to express itself?
Implementing $\lambda$-calculus

- What would be required to automate $\lambda$-calculus representation and evaluation
Implementing $\lambda$-calculus

- What would be required to automate $\lambda$-calculus representation and evaluation
  - representation for constants, applications and function definitions
Implementing $\lambda$-calculus

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

- What would be required to automate $\lambda$-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 $\lambda$-calculus evaluation
    - checking for free variables
    - renaming variables
    - performing substitutions
  - garbage collection
Implementing $\lambda$-calculus

What would be required to automate $\lambda$-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 $\lambda$-calculus evaluation
  - checking for free variables
  - renaming variables
  - performing substitutions
Implementing $\lambda$-calculus

- What would be required to automate $\lambda$-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 $\lambda$-calculus evaluation
    - checking for free variables
    - renaming variables
    - performing substitutions
  - garbage collection
Why Garbage Collection

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

\[
\lambda x \left( \text{CONS} \left( \text{CONS} \right. \left. \left. 2 \right. \left. x \right. \right) \right) \text{CONS} \left( 3 \right. \left. x \right. \right) \text{CONS} \left( 1 \right. \left. \text{nil} \right. )
\]

- 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
Why Garbage Collection

- No imperative assignment → no side-effects
- Efficiency maintained by shared references
Why Garbage Collection

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

\[
(\lambda x \mid (\text{CONS} \ (\text{CONS} \ 2 \ x) \ (\text{CONS} \ 3 \ x)) \ (\text{CONS} \ 1 \ \text{nil}))
\]

- 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
Why Garbage Collection

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

$$\lambda x \mid (\textit{CONS} (\textit{CONS} 2 x) (\textit{CONS} 3 x)) \text{(CONS 1 nil)}$$

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

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

$$\left(\lambda x \mid (\text{CONS} \ (\text{CONS} \ 2 \ x) \ (\text{CONS} \ 3 \ x)) \ (\text{CONS} \ 1 \ \text{nil})\right)$$

- 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

\[
\lambda x \mid (\text{CONS} (\text{CONS} 2 x) (\text{CONS} 3 x)) (\text{CONS} 1 \text{ nil})
\]

- 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

\[ \lambda x . \ (\text{CONS} \ (\text{CONS} \ 2 \ x) \ (\text{CONS} \ 3 \ x)) \ \text{CONS} \ 1 \ \text{nil} \]

- 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
"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
What do we have to represent?
What do we have to represent?

\[ \langle \text{expression} \rangle := \langle \text{identifier} \rangle \mid \langle \text{application} \rangle \mid \langle \text{function} \rangle \]
What do we have to represent?

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

⟨identifier⟩ := a | b | c | ...
What do we have to represent?

\[
\langle \text{expression} \rangle := \langle \text{identifier} \rangle \mid \langle \text{application} \rangle \mid \langle \text{function} \rangle
\]

\[
\langle \text{identifier} \rangle := a \mid b \mid c \mid \ldots
\]

\[
\langle \text{application} \rangle := "(" \langle \text{expression} \rangle \langle \text{expression} \rangle ")"
\]
What do we have to represent?

- expression ::= identifier | application | function

- identifier ::= a | b | c | ...

- application ::= "(" expression expression ")"

- function ::= "(λ" identifier "|" expression ")"
In $\lambda$-calculus, all data types are represented as $\lambda$ expressions.
In λ-calculus, all data types are represented as λ expressions.

Need a way to distinguish: identifier, application, function.
In \( \lambda \)-calculus, all data types are represented as \( \lambda \) expressions.

Need a way to distinguish: identifier, application, function.

Use a cons cell where FIRST is type, and SECOND is data.
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
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 \(0, 1, 2, \ldots\)
- Where \(0\) is type-marked identifier with church number 0
  i.e. \(0 \equiv \text{cons}(0, 0), 1 \equiv \text{cons}(0, 1)\)
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 $0, 1, 2, \ldots$
- Where $0$ is type-marked identifier with church number 0
  
  i.e. $0 \equiv \text{cons}(0, 0), 1 \equiv \text{cons}(0, 1)$

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 $(a \ b)$

\[
\equiv (0 \ 1) \\
\equiv \text{cons}(1, \text{cons}(\text{cons}(0, 0), \text{cons}(0, 1)))
\]
Again, use CONS cell for function definition:
Again, use CONS cell for function definition:

$$(\lambda a \mid (a \ b))$$
Again, use CONS cell for function definition:

\[
(\lambda a \mid (a \, b))
\]

\[
\equiv\$(\lambda \$0 \mid \$( \$0 \, \$1) )
\]
Again, use CONS cell for function definition:

\[(\lambda a \mid (a \ b))\]
\[\equiv (\lambda 0 \mid (0 \ 1))\]
\[\equiv \text{cons}(2, \text{type} ;; \text{Type marker for function def})\]
Again, use CONS cell for function definition:

$$(\lambda a \mid (a \ b))$$

$$\equiv$$(\lambda 0 \mid $( 0 1))$$

$$\equiv$$cons($$_2$$, ; Type marker for function def
$$type$$
$$cons($$ ; Cons of parm and body

;; Parameter a
$$cons($$
$$cons($$
$$cons($$ ; Type for application
$$cons($$
$$cons($$ ; Parameter a
Again, use CONS cell for function definition:

$$(\lambda a \mid (a \ b))$$

$$\equiv (\lambda 0 \mid (0 \ 1))$$

$$\equiv \text{cons}(2, \text{type})$$ ; Type marker for function def

$$\text{cons}(\text{type})$$ ; Cons of parm and body

$$\text{cons}(0, 0)$$ ; Parameter a
Again, use CONS cell for function definition:

\[(\lambda a \ | \ (a \ b))\]

\[\equiv $(\lambda$0 \ | \ $(\ 0 \ 1))$\]

\[\equiv \text{cons(2, ; Type marker for function def}
\text{type}
\text{cons(}
\text{cons(0, 0 ), ; Cons of parm and body}
\text{type id}
\text{cons(1, ; Type for application}
\text{type}
\text{cons(0, 0 )})$)\]
Again, use CONS cell for function definition:

\[(\lambda a \mid (a \ b))\]

\[\equiv (\lambda$0 \mid $(0 \ 1))\]

\[\equiv \text{cons}(2, \text{;; Type marker for function def})\]

\[\text{cons(\text{type})}\]

\[\text{cons(\text{cons(0, 0)\;; Parameter a}}\]

\[\text{type id}\]

\[\text{cons(1, \text{;; Type for application}}\]

\[\text{type}\]

\[\text{cons(cons(0, 0), cons(0, 1)))\;; Body (}}\]

\[\text{type id type id}\]
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

\[ \text{new-id}(\text{last-id}) \]

\[ ;; \text{create a new identifier with type marker} \]

\[ \equiv \text{cons}(0, \text{successor}(\text{second}(\text{last-id}))) \]

\[ \text{new-app}(\text{function}, \text{argument}) \]

\[ ;; \text{create a new function application with type} \]

\[ \equiv \text{cons}(1, \text{cons}(\text{function}, \text{argument})) \]
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

\((\lambda a \mid (a \ b))\) \ c
Creating Representations II

\[ (\lambda a \mid (a \ b)) \ c \]
\[ \equiv \ \text{LET } a = 0 \ \text{IN} \]
\[ \quad \text{LET } b = \text{new-id}(a) \ \text{IN} \]
\[ \quad \text{LET } c = \text{new-id}(b) \ \text{IN} \]
Creating Representations II

\((\lambda a \mid (a \; b)) \; c\)

\[\equiv \text{LET } a = 0 \text{ IN}
\text{LET } b = \text{new-id}(a) \text{ IN}
\text{LET } c = \text{new-id}(b) \text{ IN}
\text{new-app}(
\text{new-def}(a, \text{new-app}(a, b)),
c)\]
Predicates using abstract programming idioms

- Recall: all datatypes are of the form: \((\text{type}, \text{value})\)
Predicates using abstract programming idioms

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

\[
\text{is-id}(\langle E \rangle) ;; \text{True if } \langle E \rangle \text{ is constant identifier}
\]
\[
\equiv \text{IF } \text{car}(\langle E \rangle)=0 \text{ THEN T ELSE F}
\]
Predicates using abstract programming idioms

- Recall: all datatypes are of the form: \((\text{type, value})\)

\[
is\text{-id}(\langle E \rangle) \\
\quad ;; \text{True if } \langle E \rangle \text{ is constant identifier} \\
\equiv \text{IF car}(\langle E \rangle) = 0 \text{ THEN } T \text{ ELSE } F
\]

\[
is\text{-app}(\langle E \rangle) \\
\quad ;; \text{True if } \langle E \rangle \text{ is constant identifier} \\
\equiv \text{IF car}(\langle E \rangle) = 1 \text{ THEN } T \text{ ELSE } F
\]
Representation of Type Predicates

Predicates using abstract programming idioms

- Recall: all datatypes are of the form: \((\text{type}, \text{value})\)

  \[
  \text{is-id}(\langle E \rangle) \\
  \quad ;; \text{True if } \langle E \rangle \text{ is constant identifier} \\
  \equiv \text{IF } \text{car}(\langle E \rangle) = 0 \text{ THEN } T \text{ ELSE } F \\
  \]

  \[
  \text{is-app}(\langle E \rangle) \\
  \quad ;; \text{True if } \langle E \rangle \text{ is constant identifier} \\
  \equiv \text{IF } \text{car}(\langle E \rangle) = 1 \text{ THEN } T \text{ ELSE } F \\
  \]

  \[
  \text{is-func}(\langle E \rangle) \\
  \quad ;; \text{True if } \langle E \rangle \text{ is constant identifier} \\
  \equiv \text{IF } \text{car}(\langle E \rangle) = 2 \text{ THEN } T \text{ ELSE } F \\
  \]
Accessing Representations

Abstract idioms for datatypes of the form \((\text{type}, \text{value})\)
Accessing Representations

Abstract idioms for datatypes of the form \((\text{type},\text{value})\)

- **Application Accessors for** \((\text{type} \ (\text{function} \ \text{argument}))\)

\[\text{get-func}(A) \equiv \text{car}(\text{cdr}(A)); \text{ie funct of application}\]
Accessing Representations

Abstract idioms for datatypes of the form \((\text{type}, \text{value})\)

- Application Accessors for \((\text{type} \ (\text{function argument}))\)

\[
\begin{align*}
\text{get-func}(A) & \equiv \text{car}(\text{cdr}(A)) \ ; \text{ie funct of application} \\
\text{get-arg}(A) & \equiv \text{cdr}(\text{cdr}(A)) \ ; \text{ie arg of application}
\end{align*}
\]
Accessing Representations

Abstract idioms for datatypes of the form \((\text{type, value})\)

- Application Accessors for \((\text{type (function argument)})\)

  \[
  \text{get-func}(A) \equiv \text{car}(\text{cdr}(A)) \; ; \text{ie funct of application}
  \]

  \[
  \text{get-arg}(A) \equiv \text{cdr}(\text{cdr}(A)) \; ; \text{ie arg of application}
  \]

- Function Definition Accessors for \((\text{type (parameter body)})\)
Accessing Representations

Abstract idioms for datatypes of the form \((\text{type}, \text{value})\)

- **Application Accessors for** \((\text{type (function argument)})\)
  
  \[
  \text{get-func}(A) \equiv \text{car(cdr}(A)) ; \text{ie funct of application}
  \]
  
  \[
  \text{get-arg}(A) \equiv \text{cdr(cdr}(A)) ; \text{ie arg of application}
  \]

- **Function Definition Accessors for** \((\text{type (parameter body)})\)
  
  \[
  \text{get-parm}(F) \equiv \text{car(cdr}(F)) ; \text{ie get } \lambda \text{ parameter}
  \]
Accessing Representations

Abstract idioms for datatypes of the form \((\text{type},\text{value})\)

- **Application Accessors for \((\text{type (function argument)})\)**

  \[
  \text{get-funct}(A) \equiv \text{car}(	ext{cdr}(A)) \; \text{;ie funct of application}
  \]
  \[
  \text{get-arg}(A) \equiv \text{cdr}(	ext{cdr}(A)) \; \text{;ie arg of application}
  \]

- **Function Definition Accessors for \((\text{type (parameter body)})\)**

  \[
  \text{get-parm}(F) \equiv \text{car}(	ext{cdr}(F)) \; \text{;ie get \(\lambda\) parameter}
  \]
  \[
  \text{get-body}(F) \equiv \text{cdr}(	ext{cdr}(F))
  \]
λ-calculus Evaluation Function

- Implement λ-evaluation as 3 functions:
Implement $\lambda$-evaluation as 3 functions:

- eval: takes a $\lambda$-calculus expression and returns its evaluation
Implement $\lambda$-evaluation as 3 functions:

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

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

- eval: takes a $\lambda$-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
\textbf{\textit{\lambda}-Calculus Eval Function}

\[
eval(\langle E \rangle) \equiv \\
\text{IF } \text{is-id}(e) \text{ THEN } \langle E \rangle \equiv f : a \text{ constant} \\
e\]

\text{\begin{itemize}
\item Note: bodies of definitions are evaluated before use
\end{itemize}}
\[ \text{eval}(\langle E \rangle) \equiv \]

\[ \text{IF is-id(e)} \]
\[ \text{THEN } ;;\langle E \rangle \equiv f : \text{a constant} \]
\[ e \]

\[ \text{ELSE IF is-app(e)} \]
\[ \text{THEN } ;;\langle E \rangle \equiv (\langle F \rangle \langle A \rangle) : \text{application} \]
\[ \text{apply(get-func(e), get-arg(e))} \]
\textbf{\textit{$\lambda$}-Calculus Eval Function}

\[
\text{eval}(\langle E \rangle) \equiv \\
\text{IF is-id(e)} \quad \text{THEN } ;;\langle E \rangle \equiv f \ : \ a \ \textit{constant} \\
e \\
\text{ELSE IF is-app(e)} \quad \text{THEN } ;;\langle E \rangle \equiv (\langle F \rangle \langle A \rangle) \ : \ \textit{application} \\
\quad \text{apply(get-func(e), get-arg(e))} \\
\text{ELSE } ;;\langle E \rangle \equiv (\lambda x \ / \ \langle BODY \rangle \ ) \ : \ \textit{definition} \\
\quad \text{new-func(get-parm(e), eval(get-body(e)))}
\]

\textit{Note: bodies of definitions are evaluated before use}
\( \lambda \)-Calculus Eval Function

\[
eval(\langle E \rangle) \equiv
\]

IF is-id(e)
THEN \( ; ; \langle E \rangle \equiv f : a \text{ constant} \)
e

ELSE IF is-app(e)
THEN \( ; ; \langle E \rangle \equiv (\langle F \rangle \langle A \rangle) : \text{application} \)
apply(get-func(e), get-arg(e))

ELSE \( ; ; \langle E \rangle \equiv (\lambda x \mid \langle \text{BODY} \rangle ) : \text{definition} \)
new-func(get-parm(e), eval(get-body(e)))

- Note: body of definitions are evaluated before use
Applicative-Order Apply Function

\[
\text{apply}(\langle F \rangle, \langle A \rangle) \equiv \text{apply function } \langle F \rangle \text{ to argument } \langle A \rangle \\
\text{LET } b=\text{eval}(\langle A \rangle) \text{ IN }
\]
Applicative-Order Apply Function

\[
\text{apply}(\langle F \rangle, \langle A \rangle) \equiv \;; \text{ apply function } \langle F \rangle \text{ to argument } \langle A \rangle
\]

LET \( b = \text{eval}(\langle A \rangle) \) IN

IF \( \text{is-id}(\langle F \rangle) \)
THEN \( \;; (\langle F \rangle \langle A \rangle) \equiv (f \langle A \rangle) \)
ELSE \( \text{new-app}(\langle F \rangle, b) \)
Applicative-Order Apply Function

\[
\text{apply}(\langle F \rangle, \langle A \rangle) \equiv \text{\text{\text{apply function}} } \langle F \rangle \text{ to argument } \langle A \rangle
\]

\[
\text{LET } b=\text{eval}(\langle A \rangle) \text{ IN}
\]

\[
\text{IF is-id}(\langle F \rangle) \text{ THEN } (\langle F \rangle(\langle A \rangle) \equiv (f(\langle A \rangle))
\]

\[
d \quad \text{new-app}(\langle F \rangle, b)
\]

\[
\text{ELSE IF is-app}(\langle F \rangle) \text{ THEN } (\langle F \rangle(\langle A \rangle) \equiv ((\langle G \rangle(\langle C \rangle))(\langle A \rangle))
\]

\[
\text{IF is-id}(\text{get-func}(\langle F \rangle)) \text{ THEN new-app(}
\]

\[
\text{new-app}(\text{get-func}(\langle F \rangle), \text{eval}(\langle C \rangle)),
\]

\[
b\)
\]

\[
\text{ELSE apply}(\text{eval}(\langle F \rangle), b)
\]
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⟩)
  ELSE 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⟩)))
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 $\lambda y$ must be renamed

Simplification: Do not check for free parameters always rename formal parameters
\(\lambda\)-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 \(\lambda y\) must be renamed

- In an application like \((\lambda y \mid y) x\)
  - formal parameter \(\lambda y\) does not have to be renamed
  - But, renaming \(\lambda y\) does not alter meaning
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 $\lambda y$ must be renamed

In an application like $(\lambda y \mid y) \ x$
  - formal parameter $\lambda y$ does not have to be renamed
  - But, renaming $\lambda y$ does not alter meaning

Simplification: Do not check for free parameters — always rename formal parameters
\( \text{subs}(s,v,\langle E \rangle) \); substitute \( s \) for \( v \) in expression \( \langle E \rangle \)

\[
\text{IF is-id(}\langle E \rangle\text{)
THEN } ;; \text{ base case, either constant matches or not}
\text{IF } \langle E \rangle = v \text{ THEN } s \text{ ELSE } \langle E \rangle
\]
\lambda\text{-Calculus Substitution II}

\texttt{subs(s,v,}\langle E\rangle\texttt{) ;; substitute s for var v in expression }\langle E\rangle\texttt{)}

\texttt{IF is-id(}\langle E\rangle\texttt{) THEN };; \texttt{base case, either constant matches or not}
\texttt{IF }\langle E\rangle=\texttt{v THEN s ELSE }\langle E\rangle\texttt{)}

\texttt{ELSE IF is-app(}\langle E\rangle\texttt{) THEN };; \texttt{application, substitute within (}\langle F\rangle \langle A\rangle\texttt{)}
\texttt{new-app( subs(s,v,get-func(}\langle E\rangle\texttt{)),}
\texttt{ subs(s,v,get-arg(}\langle E\rangle\texttt{)))))}
\[ \text{subs}(s,v,\langle E \rangle) \quad ;; \text{substitute } s \text{ for var } v \text{ in expression } \langle E \rangle \]

IF is-id(\langle E \rangle)
THEN \quad ;; \text{base case, either constant matches or not}
    IF \langle E \rangle = v \text{ THEN } s \text{ ELSE } \langle E \rangle
ELSE IF is-app(\langle E \rangle)
THEN \quad ;; \text{application, substitute within } (\langle F \rangle \langle A \rangle)
    new-app( \text{subs}(s,v,\text{get-func}(\langle E \rangle)),
                \text{subs}(s,v,\text{get-arg}(\langle E \rangle)))

(\text{continued on next slide} \ldots)
ELSE ;; Definition ($\lambda f/B$) - check variable issues!

LET f = get-parm($E$) IN
ELSE ;; Definition ($\lambda f/B$) - 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,subs(z,f,b)))
ELSE ;; Definition ($\lambda f/\langle B \rangle$) - check variable issues!

LET $f = \text{get-parm}(\langle E \rangle)$ IN

IF $f = v$
THEN ;; var shadowed by formal parameter -> done!
\langle E \rangle

ELSE ;; always rename binding variable
LET $z = \text{new-id}()$ AND $b = \text{get-body}(\langle E \rangle)$ IN
new-func(
  $z$, subs(s,v, ;; beta substitution
  subs(z,f,b))) ;; alpha renaming
Applying $\lambda$-Calculus Evaluation

- To evaluate: $(\lambda x \mid x) \ a$
Applying λ-Calculus Evaluation

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

LETREC zero = $(\lambda sz \mid z)$

AND successor = $(\lambda x (\lambda sz \mid s(xsz)))$

AND add =
Applying $\lambda$-Calculus Evaluation

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

  \begin{verbatim}
  LETREC zero = (\sz \mid z)
  AND successor = (\x (\sz \mid s(xs_sz))
  AND add =
  :
  AND zerop =
  \end{verbatim}
Applying \(\lambda\)-Calculus Evaluation

- To evaluate: \((\lambda\ x\ |\ x)\ a\)

\[
\text{LETREC}\ \text{zero} = (\lambda\ \text{sz}|\ z) \\
\text{AND} \ \text{successor} = (\lambda\ x\ (\lambda\text{sz}|\text{s(xsz)})) \\
\text{AND} \ \text{add} = \\
\vdots \\
\text{AND} \ \text{zerop} = \\
\vdots \\
\text{AND} \ \text{eval} = \langle\text{BODY}\rangle \\
\text{AND} \ \text{apply} = \langle\text{BODY}\rangle \text{ AND } \text{subs} = \langle\text{BODY}\rangle \text{ IN}
\]
Applying $\lambda$-Calculus Evaluation

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

\[
\begin{align*}
\text{LETREC} & \quad \text{zero} = (\lambda \text{sz} \mid z) \\
\text{AND} & \quad \text{successor} = (\lambda x (\lambda \text{sz} \mid s(xsz))) \\
\text{AND} & \quad \text{add} = \\
\text{AND} & \quad \text{zerop} = \\
\text{AND} & \quad \text{eval} = \langle \text{BODY} \rangle \\
\text{AND} & \quad \text{apply} = \langle \text{BODY} \rangle \text{ AND } \text{subs} = \langle \text{BODY} \rangle \quad \text{IN} \\
\text{LET} & \quad x = 0 \quad \text{IN} \\
\text{LET} & \quad a = \text{new-id}(x) \quad \text{IN}
\end{align*}
\]
Applying λ-Calculus Evaluation

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

\[
\text{LETREC } \text{zero} = (\lambda sz \mid z) \\
\text{AND successor} = (\lambda x (\lambda sz \mid s(xsz))) \\
\text{AND add} = \\
:\text{AND zerop} = \\
:\text{AND eval} = \langle \text{BODY} \rangle \\
\text{AND apply} = \langle \text{BODY} \rangle \text{ AND subs} = \langle \text{BODY} \rangle \text{ IN}
\]

\[
\text{LET } x = 0 \text{ IN} \\
\text{LET } a = \text{new-id}(x) \text{ IN} \\
\quad \text{eval} ( \text{new-app} (\text{new-func}(x,x),a) )
\]
Here, we ignore underlying representation
Here, we ignore underlying representation

Just examine how Eval, Apply and Subs work together
\(\text{\textbf{\textlambda-}Calculus Evaluation Example I}\)

- Here, we ignore underlying representation
- Just examine how Eval, Apply and Subs work together
- Square brackets avoid confusion with \(\text{\textlambda-}\text{C}\) arguments
Here, we ignore underlying representation

Just examine how Eval, Apply and Subs work together

Square brackets avoid confusion with \(\lambda\)-C arguments

\[
eval[(\lambda y \mid s)] ;; \text{Case: function def}
\]
Here, we ignore underlying representation

Just examine how Eval, Apply and Subs work together

Square brackets avoid confusion with $\lambda$-C arguments

\[
eval[(\lambda y \mid s)] \quad ;; \text{Case: function def}
\]
\[
\text{new-func[ get-id[(\lambda y \mid s)]]}
\]
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

\[ \text{eval}[(\lambda y \mid s)] \quad ;; \text{Case: function def} \]
\[ \text{new-func}[ \text{get-id}[(\lambda y \mid s)] \]
\[ \text{eval}[s] \]
Here, we ignore underlying representation

Just examine how Eval, Apply and Subs work together

Square brackets avoid confusion with $\lambda$-C arguments

```plaintext
eval[ (\lambda y | s) ] ;; Case: function def
new-func[ get-id[ (\lambda y | s) ]
    eval[s] ]
get-id[ (\lambda y | s) ] \rightarrow y
```
Here, we ignore underlying representation

Just examine how Eval, Apply and Subs work together

Square brackets avoid confusion with \( \lambda \)-C arguments

\[
\begin{align*}
eval[(\lambda y \mid s)] & \quad ;; \text{Case: function def} \\
\text{new-func} & [ \text{get-id}[(\lambda y \mid s)] ] \\
& \text{eval}[s] \\
\text{get-id}[(\lambda y \mid s)] & \rightarrow y \\
\text{get-body}[(\lambda y \mid s)] & \rightarrow s
\end{align*}
\]
Here, we ignore underlying representation

Just examine how Eval, Apply and Subs work together

Square brackets avoid confusion with \( \lambda \)-C arguments

\[
\text{eval[ (\lambda y \mid s) ] ;; Case: function def}
\]
\[
\text{new-func[ get-id[ (\lambda y \mid s) ]}
\]
\[
\text{eval[s] ]}
\]
\[
\text{get-id[ (\lambda y \mid s) ] } \rightarrow y
\]
\[
\text{get-body[ (\lambda y \mid s) ] } \rightarrow s
\]
\[
\text{new-func[ y, s ]}
\]
Here, we ignore underlying representation

Just examine how Eval, Apply and Subs work together

Square brackets avoid confusion with λ-C arguments

\[
\begin{align*}
\text{eval}[(\lambda y \mid s)] & \quad ;; \text{Case: function def} \\
\text{new-func}[ & \quad \text{get-id}[(\lambda y \mid s)]] \\
& \quad \text{eval}[s] ] \\
\text{get-id}[(\lambda y \mid s)] & \rightarrow y \\
\text{get-body}[(\lambda y \mid s)] & \rightarrow s \\
\text{new-func}[y, s] & \rightarrow (\lambda y \mid s)
\end{align*}
\]
\(\lambda\)-Calculus Evaluation Example II

\[
\text{eval[ ((}\lambda y \mid s) \ x) ]] ;; \text{Case: application}
\]
\textbf{\(\lambda\)-Calculus Evaluation Example II}

\begin{align*}
\text{eval}[\ (\lambda y \ | \ s \ x)] & \quad ;; \text{Case: application} \\
\text{apply}[\ \text{get-fun}[\ (\lambda y | s) \ x],\ ]
\end{align*}
\textbf{\(\lambda\)-Calculus Evaluation Example II}

\[
\text{eval} [ ( (\lambda y \mid s) \ x) ] ;; \text{Case: application}
\]

\[
\text{apply} [ \text{get-fun} [ ( (\lambda y \mid s) \ x) ] , \\
\text{get-arg} [ ( (\lambda y \mid s) \ x) ] ]
\]
\[ \text{eval}\left[\left(\lambda y \mid s\right)x\right] \quad ;; \text{Case: application} \]

\[ \text{apply}\left[\text{get-fun}\left[\left(\lambda y \mid s\right)x\right],\right. \]
\[ \quad \left.\text{get-arg}\left[\left(\lambda y \mid s\right)x\right]\right] \]

\[ \equiv \text{apply}\left[\left(\lambda y \mid s\right),x\right] \quad ;; \text{(definition, arg)} \]
\( \lambda \)-Calculus Evaluation Example II

\[
\text{eval}\[ (\lambda y \mid s) x) \]\;; \text{Case: application}
\]

\[
\text{apply}\[ \text{get-fun}\[ (\lambda y \mid s) x) \],
\text{get-arg}\[ (\lambda y \mid s) x) \] \]
\]

\[
\equiv \text{apply}\[ (\lambda y \mid s), x \] ;; \text{(definition, arg)}
\]

\[
\text{eval}\]
\]
\textbf{λ-Calculus Evaluation Example II}

\begin{verbatim}
  eval[ ((\lambda y \mid s) x) ] ;; Case: application

  apply[ get-fun[ ((\lambda y \mid s) x) ],
         get-arg[ ((\lambda y \mid s) x) ] ]

  \equiv apply[ (\lambda y \mid s), x ] ;; (definition, arg)

  eval[
    subs[ eval[x],
      
    ]
\end{verbatim}
λ-Calculus Evaluation Example II

\[ \text{eval}[ (\lambda y \mid s) x) ] \ ;; \text{Case: application} \]

\[ \text{apply}[ \text{get-fun}[ (\lambda y \mid s) x) ], \]
\[ \text{get-arg}[ (\lambda y \mid s) x) ] \]

\[ \equiv \text{apply}[ (\lambda y \mid s), x ] \ ;; \text{(definition, arg)} \]

\[ \text{eval}[ \]
\[ \text{subs}[ \text{eval}[x], \]
\[ \text{get-id}[ (\lambda y \mid s) ] \]
eval[(\(\lambda y \mid s\) \(x\))] ;; Case: application

\[
\text{apply[ get-fun[(\(\lambda y \mid s\) \(x\))],} \\
\text{get-arg[(\(\lambda y \mid s\) \(x\))] ]}
\]

\(\equiv\text{apply[(\(\lambda y \mid s\), \(x\))]};; (\text{definition, arg})

\[
\text{eval[} \\
\text{subs[ eval[}x\text{],} \\
\text{get-id[(\(\lambda y \mid s\)]} \\
\text{get-body[(\(\lambda y \mid s\)]]} \text{] }}
\]
\( \lambda \)-Calculus Evaluation Example II

\[
eval[ ((\lambda y \mid s) \, x)] \quad \text{;; Case: application}
\]

\[
apply[ \, \text{get-fun}[ ((\lambda y | s) \, x) ] ,
      \text{get-arg}[ ((\lambda y | s) \, x) ] ]
\]

\[
\equiv apply[ (\lambda y \mid s), x ] \quad \text{;; (definition, arg)}
\]

\[
eval[
    \text{subs}[ \, \eval[x],
               \text{get-id}[ (\lambda y \mid s) ]
               \text{get-body}[ (\lambda y | s) ] ]
    \eval[ \, \text{subs}[ x, y, s ] ]
\]

Dr. B. Price and Dr. R. Greiner

COMPUT325: Meta-interpretation
\( \text{eval[ } ((\lambda y \mid s) \, x) \text{ ] } ;; \text{ Case: application} \)

\[ \text{apply[ get-fun[ ((\lambda y|s) \, x) ],} \]
\[ \text{get-arg[ ((\lambda y|s) \, x) ] } \]

\[ \equiv \text{apply[ (\lambda y \mid s), x ] } ;; \text{(definition, arg)} \]

\[ \text{eval[} \]
\[ \text{subs[ eval[x],} \]
\[ \text{get-id[ (\lambda y \mid s) ]} \]
\[ \text{get-body[ (\lambda y\mid s) ] } \]
\[ \text{eval[ subs[ x, y, s ] ]} \]
\[ \text{eval[ s]} \]
\lambda-Calculus Evaluation Example II

eval[ ((\lambda y \mid s) x) ] ;; Case: application

apply[ get-fun[ ((\lambda y \mid s) x) ],
       get-arg[ ((\lambda y \mid s) x) ] ]

≡ apply[ (\lambda y \mid s), x ] ;; (definition, arg)

eval[
subs[ eval[x],
       get-id[ (\lambda y \mid s) ]
       get-body[ (\lambda y \mid s) ] ]

eval[ subs[ x, y, s ] ]
eval[ s ]

→ s
\[ \text{eval} \left[ (\lambda y | s) \ ((\lambda y | s) \ x) \right] ;; \text{Case: application} \]
\( \lambda \)-Calculus Evaluation Example III

\[
\text{eval} \ [ \ ( (\lambda y | s ) ((\lambda y | s ) x) ) \ ] \ ;; \ \text{Case: application} \\
\text{apply} [ \ (\lambda y | s ), ((\lambda y | s ) x) ] \ ;; \ \text{Case: (Def, Arg)}
\]
\[ \text{eval } [ ( (\lambda y|s) \ (\lambda y|s) \ x ) ] \ ;; \text{ Case: application} \]
\[ \text{apply}[ (\lambda y|s), \ (\lambda y|s) \ x ] \ ;; \text{ Case: (Def, Arg)} \]
\[ \text{eval}[ \text{subs}[ \text{eval}[((\lambda y|s) \ x)], \ y, \ s ] ] \]
\( \lambda \text{-Calculus Evaluation Example III} \)

\[
\begin{align*}
\text{eval} & \left[ ( (\lambda y | s) \ ((\lambda y | s) \ x) ) \right] ;; \text{Case: application} \\
\text{apply} & \left[ (\lambda y | s), ((\lambda y | s) \ x) \right] ;; \text{Case: (Def, Arg)} \\
\text{eval} & \left[ \text{subs} \left[ \text{eval}\left[ ((\lambda y | s) \ x) \right], y, s \right] \right] \\
\text{eval} & \left[ ((\lambda y | s) \ x) \right] ; \text{case: application}
\end{align*}
\]
\( \text{eval} \left[ ( (\lambda y|s) ( (\lambda y|s) x ) ) \right] \); Case: application
apply[ (\lambda y|s), ( (\lambda y|s) x ) ] ;; Case: (Def, Arg)
\text{eval}[ \text{subs}[ \text{eval}[((\lambda y|s) x)], \, y, \, s ]
\text{eval}[( (\lambda y|s) x )] ; \text{case: application}
apply[ (\lambda y \mid s), \, x ] ;; (definition, arg)
\( \text{eval} \left[ ( \lambda y | s ) \ ((\lambda y | s ) \ x) \right] \); Case: application
apply[ (\lambda y | s ), ((\lambda y | s ) \ x) ] ;; Case: (Def, Arg)
\text{eval} [ \text{subs} [ \text{eval} [ ((\lambda y | s) \ x) ], y, s ]
\text{eval} [ ((\lambda y | s) \ x) ] \); case: application
apply[ (\lambda y \mid s ), x ] ;; (definition, arg)
\text{eval} [ \text{subs} [ \text{eval} [ x ], y, s ] ]\)
\( \text{eval} \left[ ( (\lambda y | s) \ ( (\lambda y | s) \ x) ) \right] \); Case: application
\( \text{apply}[ (\lambda y | s), \ ( (\lambda y | s) \ x) ] \); Case: (Def, Arg)
\( \text{eval}[ \text{subs}[ \text{eval}[ ( (\lambda y | s) \ x) ], \ y, \ s ] \)
\( \text{eval}[ ( (\lambda y | s) \ x) ] \); case: application
\( \text{apply}[ (\lambda y | s), \ x ] \); (definition, arg)
\( \text{eval}[\text{subs}[\text{eval}[x], \ y, \ s]] \)
\( \text{eval}[x] \rightarrow x \); constant identifier
\(\lambda\)-Calculus Evaluation Example III

eval \[ ( (\lambda y|s) ((\lambda y|s) x) ) \] ;; Case: application
apply\[ (\lambda y|s), ((\lambda y|s) x) \] ;; Case: (Def, Arg)
eval\[ subs[ eval[((\lambda y|s) x)], y, s ] \]
eval[((\lambda y|s) x)] ;; case: application
apply\[ (\lambda y | s), x \] ;; (definition, arg)
eval[subs[eval[x], y ,s]]
eval[x] \rightarrow x ;; constant identifier
eval[subs[ x, y, s ] ]
\(\lambda\)-Calculus Evaluation Example III

eval \[ ( (\lambda y|s) ((\lambda y|s) x) ) ] ;; Case: application
apply[ (\lambda y|s), ((\lambda y|s) x) ] ;; Case: (Def, Arg)
eval[ subs[ eval[(\lambda y|s) x]], y, s ]
eval[(\lambda y|s) x] ; case: application
apply[ (\lambda y | s), x ] ;; (definition, arg)
eval[subs[eval[x], y ,s]]
eval[x] \rightarrow x ;; constant identifier
eval[subs[ x, y, s ] ]

\rightarrow s
\[ \text{eval} \left[ (\lambda y|s) ((\lambda y|s) x) \right] \] ;; Case: application
\[ \text{apply}\left[ (\lambda y|s), ((\lambda y|s) x) \right] \] ;; Case: (Def, Arg)
\[ \text{eval}\left[ \text{subs}\left[ \text{eval}\left[ ((\lambda y|s) x) \right], \ y, \ s \right] \right] \]
\[ \text{eval}\left[ ((\lambda y|s) x) \right] \] ; case: application
\[ \text{apply}\left[ (\lambda y| s), x \right] \] ;; (definition, arg)
\[ \text{eval}\left[ \text{subs}\left[ \text{eval}\left[ x \right], \ y, s \right] \right] \]
\[ \text{eval}\left[ x \right] \rightarrow x \] ;; constant identifier
\[ \text{eval}\left[ \text{subs}\left[ x, y, s \right] \right] \]
\[ \rightarrow s \]
\[ \text{eval}\left[ \text{subs}\left[ s, y, s \right] \right] \]
\[ \text{eval} \left[ ( (\lambda y|s) ( (\lambda y|s) x ) ) \right] \]; Case: application
\[ \text{apply}[ (\lambda y|s), ( (\lambda y|s) x ) ] \]; Case: (Def, Arg)
\[ \text{eval}[ \text{subs}[ \text{eval}[ ( (\lambda y|s) x )], y, s ] ] \]
\[ \text{eval}[ ( (\lambda y|s) x )] \]; case: application
\[ \text{apply}[ (\lambda y|s), x ] \]; (definition, arg)
\[ \text{eval}[\text{subs}[\text{eval}[x], y, s]] \]
\[ \text{eval}[x] \rightarrow x \]; constant identifier
\[ \text{eval}[\text{subs}[x, y, s]] \]
\[ \rightarrow s \]
\[ \text{eval}[\text{subs}[s, y, s]] \]
\[ \text{subs}[s, y, s] \rightarrow s \]
\[ \text{eval} \left[ \left( (\lambda y | s) \left( (\lambda y | s) \, x \right) \right) \right] \]; Case: application

apply\left[ (\lambda y | s), \left( (\lambda y | s) \, x \right) \right] \]; Case: (Def, Arg)

eval\left[ \text{subs}[ \text{eval}\left[ \left( (\lambda y | s) \, x \right) \right], \, y, \, s \right]\right]

\text{eval}\left[ \left( (\lambda y | s) \, x \right) \right]; \text{case: application}

apply\left[ (\lambda y | s), \, x \right]; \text{(definition, arg)}

\text{eval}\left[ \text{subs}[\text{eval}\left[ x \right], \, y, \, s] \right]\right]

\text{eval}[x] \rightarrow x ;; \text{constant identifier}

\text{eval}\left[ \text{subs}[\, x, \, y, \, s \, ] \right]\right]

\rightarrow s
\text{eval}\left[ \text{subs}[s, y, s] \right]\right]

\text{subs}[s, y, s] \rightarrow s
\text{eval}[s]
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
A normal order version of apply is required for recursive functions.

Therefore, $\lambda$-calculus evaluation is just another function.
\(\lambda\)-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: \(\lambda\)-calculus evaluation can be written as a \(\lambda\)-calculus expression.

Therefore, \(\lambda\)-calculus evaluation is just another function.

\(\lambda\)-calculus can be used to implement \(\lambda\)-calculus.
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:
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:
  - \( \lambda \)-calculus evaluation can be written as a \( \lambda \)-calculus expression
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:
  \( \lambda \)-calculus evaluation can be written as a \( \lambda \)-calculus expression
  Therefore, \( \lambda \)-calculus evaluation is just another 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:
  - $\lambda$-calculus evaluation can be written as a $\lambda$-calculus expression
  - Therefore, $\lambda$-calculus evaluation is just another function
  - $\lambda$-calculus can be used to implement $\lambda$-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 $\lambda$-calculus
Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to \(\lambda\)-calculus
- \(\lambda\)-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 $\lambda$-calculus
- $\lambda$-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 $\lambda$-calculus
- $\lambda$-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 $\lambda$-calculus
- $\lambda$-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 $\lambda$-calculus
- $\lambda$-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 $\lambda$-calculus
- $\lambda$-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 $\lambda$-calculus
- $\lambda$-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

No now can compile new language directly to platform
Called bootstrapping
Bootstrapping

- Functional languages can be written in abstract programming language
- Abstract programming has a simple translation to \( \lambda \)-calculus
- \( \lambda \)-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 $\lambda$-calculus
- $\lambda$-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
Possible to implement \(\lambda\)-Calculus Evaluator in Lisp

But: Lisp has:

- basic data types: numbers, lists, constants
- a type system with predicates: 'atom', 'consp'
- primitive functions: +, -, cons, car, cdr

Can replace low-level \(\lambda\)-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
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


\(\lambda\)-Calculus Evaluation in Lisp

- Possible to implement \(\lambda\)-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 \(\lambda\)-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
Consider the following example

\[(\lambda x \mid \text{IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y \ z) \ x) \ x) \ \text{ELSE } ((\lambda y \mid y) \ x)) z \] 

\[\beta \rightarrow [z/x] \text{ IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y \ z) \ x) \ x) \ \text{ELSE } ((\lambda y \mid y) \ x) \]
Consider the following example

\[
(\lambda x \mid \text{IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y z) x) x) \\
\text{ELSE } ((\lambda y \mid y) x)) \quad z
\]

\[\beta \rightarrow [ z/x ] \text{ IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y z) x) x) \text{ ELSE } ((\lambda y \mid y) x)\]

Followed generic \(\beta\)-reduction. Notice anything odd?
Consider the following example

\[
(\lambda x \mid \text{IF T THEN } ((\lambda y \mid (\lambda z \mid y z) x) x) \text{ ELSE } ((\lambda y \mid y) x)) z
\]

\[\beta \rightarrow [ z/x ] \text{ IF T THEN } ((\lambda y \mid (\lambda z \mid y z) x) x) \text{ ELSE } ((\lambda y \mid y) x)\]

Followed generic \(\beta\)-reduction. Notice anything odd?

- Substituted for both halves of IF statement even though \text{ELSE} is never used
Consider the following example

$$(\lambda x \mid \text{IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y z) x) x) \text{ ELSE } ((\lambda y \mid y) x)) z$$

$\beta\rightarrow [z/x] \text{ IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y z) x) x) \text{ ELSE } ((\lambda y \mid y) x)$$

Followed generic $\beta$-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
Consider the following example

\[(\lambda x \mid \text{IF T THEN } ((\lambda y \mid (\lambda z \mid y \ z) \ x) \ x) \ \text{ELSE } ((\lambda y \mid y) \ x)) \ z\]

\[\beta \rightarrow \left[ z/x \right] \text{IF T THEN } ((\lambda y \mid (\lambda z \mid y \ z) \ x) \ x) \ \text{ELSE } ((\lambda y \mid y) \ x)\]

Followed generic \(\beta\)-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
  - \((\lambda z \mid y \ z)\) rebuilt even though no \(x\)
Efficiency Issues

Consider the following example

$$(\lambda x \mid \text{IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y \ z) \ x) \ x) \ \text{ELSE } ((\lambda y \mid y) \ x)) \ z$$

$\xrightarrow{\beta} [ z/x ] \text{ IF } T \text{ THEN } ((\lambda y \mid (\lambda z \mid y \ z) \ x) \ x) \ \text{ELSE } ((\lambda y \mid y) \ x)$$

Followed generic $\beta$-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
  - $(\lambda z \mid y \ z)$ rebuilt even though no $x$
- In $(\lambda x \mid (\lambda y \mid (\lambda z \mid \langle E \rangle)))$, expression $\langle E \rangle$ 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

\[
\text{eval}[ ( (\lambda x | \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z \ x) ) \ y ) ]
\]
Binding Lists

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

\[
eval[(\lambda x | \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z x)) \ y)] \ [y/x] (\text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z x))
\]
Binding Lists

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

\[
eval[(\lambda x | \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z \ x)) \ y)] [y/x] \ (\text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z \ x)) \rightarrow \text{IF } T \text{ THEN } (\lambda z | y) \text{ ELSE } (\lambda z | z \ y)
\]
Binding Lists

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

\[
\text{eval}\left[\left( (\lambda x | \text{IF T THEN (}\lambda z|x) \text{ELSE (}\lambda z | z \ x )) \ y \right)\right] \ [y/x] \ (\text{IF T THEN (}\lambda z|x) \text{ELSE (}\lambda z | z \ x )) \\
\rightarrow \text{IF T THEN (}\lambda z|y) \text{ELSE (}\lambda z | z \ y) \\
\text{eval}\left[\text{IF T THEN (}\lambda z|y) \text{ELSE (}\lambda z | z \ y) \right] \rightarrow (\lambda z|y)
\]
Binding Lists

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

\[
\text{eval}\left[\ (\lambda x| \ IF \ T \ THEN \ (\lambda z| x) \ ELSE \ (\lambda z | z \ x )) \ y)\right] \\
[y/x] \ ( IF \ T \ THEN \ (\lambda z| x) \ ELSE \ (\lambda z | z \ x )) \\
\rightarrow \ IF \ T \ THEN \ (\lambda z| y) \ ELSE \ (\lambda z | z \ y)
\]

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

\[
\text{eval}\left[\ (\lambda x| \ IF \ T \ THEN \ (\lambda z| x) \ ELSE \ (\lambda z | z \ x )) \ y)\right]
\]
Binding Lists

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

\[
eval[ ( \lambda x | \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z x) ) \ y ] \\
[y/x] ( \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z x) ) \\
\rightarrow \text{IF } T \text{ THEN } (\lambda z | y) \text{ ELSE } (\lambda z | z y) \\
\]

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

\[
eval[ ( \lambda x | \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z x) ) \ y ] \\
eval[ \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z x) , \{x\leftarrow y\} ]
\]
Binding Lists

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

\[
\text{eval}\left[ \left( \lambda x | \text{IF T THEN} \ (\lambda z | x) \ ELSE \ (\lambda z | z \ x) \right) \ y \right] \\
\left[ y/x \right] \left( \text{IF T THEN} \ (\lambda z | x) \ ELSE \ (\lambda z | z \ x) \right) \\
\rightarrow \ \text{IF T THEN} \ (\lambda z | y) \ ELSE \ (\lambda z | z \ y) \\
\text{eval}\left[ \ \text{IF T THEN} \ (\lambda z | y) \ ELSE \ (\lambda z | z \ y) \right] \rightarrow (\lambda z | y)
\]

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

\[
\text{eval}\left[ \left( \lambda x | \text{IF T THEN} \ (\lambda z | x) \ ELSE \ (\lambda z | z \ x) \right) \ y \right] \\
\text{eval}\left[ \text{IF T THEN} \ (\lambda z | x) \ ELSE \ (\lambda z | z \ x) \right], \{x\leftarrow y\} \\
\text{eval}[ \ T, \{x\leftarrow y\}]
\]
Binding Lists

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

\[
\text{eval}\left[ (\lambda x | \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z \ x)) \ y)\right] \\
[y/x] (\text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z \ x)) \rightarrow \text{IF } T \text{ THEN } (\lambda z | y) \text{ ELSE } (\lambda z | z \ y) \\
\text{eval}\left[ \text{IF } T \text{ THEN } (\lambda z | y) \text{ ELSE } (\lambda z | z \ y) \right] \rightarrow (\lambda z | y)
\]

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

\[
\text{eval}\left[ (\lambda x | \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z \ x)) \ y)\right] \\
\text{eval}\left[ \text{IF } T \text{ THEN } (\lambda z | x) \text{ ELSE } (\lambda z | z \ x), \ {x\leftarrow y}\right] \\
\text{eval}[ T, \ {x\leftarrow y}] \\
\text{eval}[ (\lambda z | x), \ {x\leftarrow y}] \\
\]
Binding Lists

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

\[
\text{eval}[(\lambda x. \text{IF } T \text{ THEN } (\lambda z|x) \text{ ELSE } (\lambda z \mid z \ x)) \ y)]
\]
\[
[y/x] (\text{IF } T \text{ THEN } (\lambda z|x) \text{ ELSE } (\lambda z \mid z \ x))
\]
\[
\rightarrow \text{IF } T \text{ THEN } (\lambda z|y) \text{ ELSE } (\lambda z \mid z \ y)
\]
\[
\text{eval} [ \text{IF } T \text{ THEN } (\lambda z|y) \text{ ELSE } (\lambda z \mid z \ y) ] \rightarrow (\lambda z|y)
\]

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

\[
\text{eval} [(\lambda x. \text{IF } T \text{ THEN } (\lambda z|x) \text{ ELSE } (\lambda z \mid z \ x)) \ y)]
\]
\[
\text{eval} [ \text{IF } T \text{ THEN } (\lambda z|x) \text{ ELSE } (\lambda z \mid z \ x), \{x ← y\} ]
\]
\[
\text{eval} [ T, \{x ← y\}]
\]
\[
\text{eval} [ (\lambda z|x), \{x ← y\}]
\]
\[
\text{eval} [ x, \{x ← y\}] \rightarrow (\lambda z|y)
\]
Binding Parameters to Expressions

- Parameter value may in turn be an expression

\[
\text{eval}\left[ (λx \ | \ (\ast \ 2 \ x)) \ (+ \ 3 \ 2), \ {} \right]
\]
Binding Parameters to Expressions

- Parameter value may in turn be an expression

\[
\text{eval}[(\lambda x \mid (* 2 x)) (+ 3 2), \{\}] \\
\text{eval}[(* 2 x), \{x\leftarrow(+ 3 2)\}] 
\]
Parameter value may in turn be an expression

\[
\begin{align*}
\text{eval}[ (\lambda x \mid (\times 2 x)) (+ 3 2), \{\} ] \\
\text{eval}[ (* 2 x), \{x\leftarrow(+ 3 2}\} ] \\
\text{eval}[2,\{x\leftarrow(+ 3 2)\}] \rightarrow 2
\end{align*}
\]
Binding Parameters to Expressions

- Parameter value may in turn be an expression

\[ \text{eval} \left[ (\lambda x \mid (* 2 x)) (+ 3 2), \{} \right] \]
\[ \text{eval} \left[ (* 2 x), \{x \leftarrow (+ 3 2)\} \right] \]
\[ \text{eval}[2, \{x \leftarrow (+ 3 2)\}] \rightarrow 2 \]
\[ \text{eval}[x, \{x \leftarrow (+ 3 2)\}] \]
Parameter value may in turn be an expression

\[
\text{eval}\left[ (\lambda x \mid (* 2 \, x)) \, (+ \, 3 \, 2), \{\} \right] \\
\text{eval}\left[ (* 2 \, x), \{x \leftarrow (+ \, 3 \, 2)\} \right] \\
\text{eval}\left[ 2, \{x \leftarrow (+ \, 3 \, 2)\} \right] \rightarrow 2 \\
\text{eval}\left[ x, \{x \leftarrow (+ \, 3 \, 2)\} \right] \\
\text{eval}\left[ (+ \, 3 \, 2) \right] \rightarrow 5
\]
Multiple bindings are added to bindings list in order of occurrence

\[
eval[(\lambda x \mid (\lambda y \mid (+ x y))) \ 3 \ 5, \ {} ]
\]
Bindings and Multiple Arguments

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

\[
\text{eval}[(\lambda x \mid (\lambda y \mid (+ x y))) \ 3 \ 5, \ \{\} \ ]
\]
\[
\text{eval}[(\lambda y \mid (+ x y)) \ 5, \ \{x\leftarrow3\} \ ]
\]
Bindings and Multiple Arguments

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

\[
\text{eval}\left[ (\lambda x \ | \ (\lambda y \ | (+ \ x \ y)) \right) \ 3 \ 5, \ {} \ ]
\]
\[
\text{eval}\left[ (\lambda y \ | (+ \ x \ y)) \ 5, \ {x\leftarrow3} \ ]
\]
\[
\text{eval}\left[ (+ \ x \ y), \ {y\leftarrow5, \ x\leftarrow3} \ ]
\]
Bindings and Multiple Arguments

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

\[
\text{eval}[(\lambda x \mid (\lambda y \mid (+ x y))) \, 3 \, 5, \{\} \, ]
\]
\[
\text{eval}[(\lambda y \mid (+ x y)) \, 5, \{x \leftarrow 3\} \, ]
\]
\[
\text{eval}[ (+ x y), \{y \leftarrow 5, x \leftarrow 3\} \, ]
\]
\[
\text{eval}[x, \{y \leftarrow 5, x \leftarrow 3\}]
\]
Multiple bindings are added to bindings list in order of occurrence

\[
\text{eval}[ (\lambda x \mid (\lambda y \mid (+ x y))) \ 3 \ 5, \ {} ]
\]
\[
\text{eval}[ (\lambda y \mid (+ x y)) \ 5, \ \{x \leftarrow 3\} ]
\]
\[
\text{eval}[ (+ x y), \ \{y \leftarrow 5, \ x \leftarrow 3\} ]
\]
\[
\text{eval}[x, \ \{y \leftarrow 5, \ x \leftarrow 3\}]
\]
\[
\text{eval}[3] \rightarrow 3
\]
Multiple bindings are added to bindings list in order of occurrence.

\[
\text{eval}\left[ (\lambda x \mid (\lambda y \mid (+ x y)) \right) 3 \ 5, \ {} \right]
\text{eval}\left[ (\lambda y \mid (+ x y)) \ 5, \ {} \{x \leftarrow 3\} \right]
\text{eval}\left[ (+ x y), \ {} \{y \leftarrow 5, \ x \leftarrow 3\} \right]
\text{eval}[x, \ {} \{y \leftarrow 5, \ x \leftarrow 3\}]
\text{eval}[3] \rightarrow 3
\text{eval}[y, \ {} \{y \leftarrow 5, \ x \leftarrow 3\}]
\]
Multiple bindings are added to bindings list in order of occurrence

\[
\begin{align*}
\text{eval} \left[ (\lambda x \mid (\lambda y \mid (+ x y)) \right) 3 &\ 5, \ \{\} \ ] \\
\text{eval} \left[ (\lambda y \mid (+ x y)) \right] 5, \ \{x \leftarrow 3\} \ ] \\
\text{eval} \left[ (+ x y), \ \{y \leftarrow 5, \ x \leftarrow 3\} \right] \\
\text{eval}[x, \ \{y \leftarrow 5, \ x \leftarrow 3\}] \\
\text{eval}[3] &\rightarrow 3 \\
\text{eval}[y, \ \{y \leftarrow 5, \ x \leftarrow 3\}] \\
\text{eval}[5] &\rightarrow 5
\end{align*}
\]
Multiple bindings are added to bindings list in order of occurrence

\[
\text{eval}(\lambda x \mid (\lambda y \mid (+ x y))) \ 3 \ 5, \ {} \]
\[
\text{eval}(\lambda y \mid (+ x y)) \ 5, \ {x \gets 3} \]
\[
\text{eval}(+ x y), \ {y \gets 5, \ x \gets 3} \]
\[
\text{eval}(x, \ {y \gets 5, \ x \gets 3})
\]
\[
\text{eval}[3] \rightarrow 3
\]
\[
\text{eval}[y, \ {y \gets 5, \ x \gets 3}]
\]
\[
\text{eval}[5] \rightarrow 5
\]
\[
\text{eval}[(+ 3 5)] \rightarrow 5
\]
Bindings and Shadowed Arguments

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

\[
eval[(\lambda x \mid (+ ((\lambda x \mid (+ x x)) 5) x)) 3, \{\} ]
\]
Bindings and Shadowed Arguments

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

\[
\text{eval} \left[ (\lambda x \mid ( + \ ( (\lambda x \mid (+ x x)) \ 5) \ x) \ 3, \ {} \right] \\
\text{eval} \left[ (+ \ ( (\lambda x \mid (+ x x)) \ 5) \ x), \ {x ← 3} \right] \\
\text{eval} \left[ x, \ {x ← 3} \right] \rightarrow 5 \\
\text{eval} \left[ x, \ {x ← 3} \right] \rightarrow 5 \\
\text{eval} \left[ (+ 10 x), \ {x ← 3} \right] \\
\text{eval} \left[ 10, \ {x ← 3} \right] \rightarrow 10 \\
\text{eval} \left[ x, \ {x ← 3} \right] \rightarrow 3 \\
\text{eval} \left[ (+ 10 x), \ {x ← 3} \right] \rightarrow 13
\]
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.

\[
\begin{align*}
\text{eval} \left[ (\lambda x \mid (+ \((\lambda x \mid (+ x x)) \ 5\)) \ x) \ 3, \ {} \right] \\
\text{eval} \left[ (+ \((\lambda x \mid (+ x x)) \ 5\)) \ x), \ \{x \leftarrow 3\} \right] \\
\text{eval} \left[ ((\lambda x \mid (+ x x)) \ 5), \ \{x \leftarrow 3\} \right] \\
\text{eval} \left[ (+ x x), \ \{x \leftarrow 5, \ x \leftarrow 3\} \right] \\
\end{align*}
\]
Bindings and Shadowed Arguments

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

\[
\text{eval}[ (\lambda x \mid (+ ((\lambda x \mid (+ x x)) 5) x) 3, \{} ]
\]
\[
\text{eval}[ (+ ((\lambda x \mid (+ x x)) 5) x), \{x←3\} ]
\]
\[
\text{eval}[ ((\lambda x \mid (+ x x)) 5), \{x←3\} ]
\]
\[
\text{eval}[ (+ x x), \{x←5, x←3\} ]
\]
\[
\text{eval}[x, \{x←5, x←3\} ] \rightarrow 5
\]
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

\[
eval[(\lambda x \mid (+ ((\lambda x \mid (+ x x)) 5) x) 3, \{\} ]
\]
\[
eval[(+ ((\lambda x \mid (+ x x)) 5) x), \{x←3\} ]
\]
\[
eval[((\lambda x \mid (+ x x)) 5), \{x←3\} ]
\]
\[
eval[(+ x x), \{x←5, x←3\} ]
\]
\[
eval[x, \{x←5, x←3\} ] \rightarrow 5
\]
\[
eval[x, \{x←5, x←3\} ] \rightarrow 5
\]
\[
\rightarrow 10
\]
Bindings and Shadowed Arguments

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

\[
\text{eval}\[ (\lambda x \mid (+ ((\lambda x \mid (+ x x)) 5) x)) 3, \{} \]
\[
\text{eval}\[ (+ ((\lambda x \mid (+ x x)) 5) x), \{x←3\} \]
\[
\text{eval}\[ ((\lambda x \mid (+ x x)) 5), \{x←3\} \]
\[
\text{eval}\[ (+ x x), \{x←5, x←3\} \]
\[
\text{eval}[x, \{x←5, x←3\}] \rightarrow 5
\[
\text{eval}[x, \{x←5, x←3\}] \rightarrow 5
\[
\rightarrow 10
\[
\rightarrow 10
\]
Bindings and Shadowed Arguments

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

```plaintext
eval[ (\lambda x \mid (+ ((\lambda x \mid (+ x x)) 5) x) 3, {} ]
eval[ (+ ((\lambda x \mid (+ x x)) 5) x), \{x←3\} ]
eval[ ((\lambda x \mid (+ 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

```plaintext
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.

\[
\text{eval}[ (\lambda x \mid (+ ((\lambda x \mid (+ x x)) 5) x)) 3, \{\} ] \\
\text{eval}[ (+ ((\lambda x \mid (+ x x)) 5) x), \{x \leftarrow 3\} ] \\
\text{eval}[ ((\lambda x \mid (+ x x)) 5), \{x \leftarrow 3\} ] \\
\text{eval}[ (+ x x), \{x \leftarrow 5, x \leftarrow 3\} ] \rightarrow 5 \\
\text{eval}[ x, \{x \leftarrow 5, x \leftarrow 3\} ] \rightarrow 5 \\
\rightarrow 10 \\
\rightarrow 10 \\
\text{eval}[ (+ 10 x), \{x \leftarrow 3\} ] \\
\text{eval}[ 10, \{x \leftarrow 3\} ] \rightarrow 10 \\
\text{eval}[ x, \{x \leftarrow 3\} ] \rightarrow 3
\]
Bindings looked up from left to right. First value found is used

\[
\text{eval}\left[ (\lambda x \mid (+ ((\lambda x \mid (+ x x)) 5) x)) 3, \{\} \right]
\]
\[
\text{eval}\left[ (+ ((\lambda x \mid (+ x x)) 5) x), \{x \leftarrow 3\} \right]
\]
\[
\text{eval}\left[ ((\lambda x \mid (+ x x)) 5), \{x \leftarrow 3\} \right]
\]
\[
\text{eval}\left[ (+ x x), \{x \leftarrow 5, x \leftarrow 3\} \right] \rightarrow 5
\]
\[
\text{eval}[x, \{x \leftarrow 5, x \leftarrow 3\}] \rightarrow 5
\]
\[
\rightarrow 10
\]
\[
\rightarrow 10
\]
\[
\text{eval}\left[ (+ 10 x), \{x \leftarrow 3\} \right]
\]
\[
\text{eval}[10, \{x \leftarrow 3\}] \rightarrow 10
\]
\[
\text{eval}[x, \{x \leftarrow 3\}] \rightarrow 3
\]
\[
\rightarrow 13
\]
Problems with Bindings and Free Variables I

eval[(\lambda y \mid (\lambda x \mid + \ x \ y)) \ 4, \ {}]
Problems with Bindings and Free Variables

\[
\text{eval}\left[\left(\lambda y \mid (\lambda x \mid + x y)\right), \{y \leftarrow 4\}\right]
\]

\[
\text{eval}\left[\left(\lambda x \mid + x y\right), \{y \leftarrow 4\}\right]
\]

Above solution breaks: See next slide!
 eval[(\(\lambda y \ | \ (\lambda x \ | \ + \ x \ y)\)) \ 4, \ {}]\n eval[(\(\lambda x \ | \ + \ x \ y\)), \ {y\leftarrow4}]\n
▶ No application here — cannot evaluate \((\lambda x| + x \ y)\) further
Problems with Bindings and Free Variables

\[
\text{eval} \left[ (\lambda y \mid (\lambda x \mid + x y)) \mid 4, \{} \right]
\]
\[
\text{eval} \left[ (\lambda x \mid + x y), \{y \leftarrow 4\} \right]
\]

- No application here — cannot evaluate \((\lambda x \mid + x y)\) further
- But, should have \(y\) bound to 4
Problems with Bindings and Free Variables I

\[
\text{eval}[(\lambda y \mid (\lambda x \mid + x y)) \ 4, \ {}]
\]
\[
\text{eval}[(\lambda x \mid + x y), \ {y \leftarrow 4}]
\]

▶ No application here — cannot evaluate \((\lambda x \mid + x y)\) further

▶ But, should have \(y\) bound to \(4\)

▶ Our simple interpreter actually handles this (but poorly):

\[
\begin{align*}
\text{evaluate} & \quad \lambda\text{-body: } + x y \text{ in environment } (y \leftarrow 4) \\
& \quad \text{create new function with evaluated body} \ (\lambda x \mid \langle B\text{ODY} \rangle) \\
& \quad \text{eval}[- x y, \ {y \leftarrow 4}] \\
& \quad \rightarrow (+ x 4) \\
& \quad \rightarrow (\lambda x \mid + x 4)
\end{align*}
\]

▶ Above solution breaks: See next slide!
Problems with Bindings and Free Variables I

\[
eval[(\lambda y \mid (\lambda x \mid + x y)) \ 4, \ {}]
eval[(\lambda x \mid + x y), \ {y←4}]
\]

- No application here — cannot evaluate \((\lambda x \mid + x y)\) further
- But, should have \(y\) bound to 4
- Our simple interpreter actually handles this (but poorly):
  - evaluate \(\lambda\)-body: \(+ x y\) in environment \((y←4)\)
Problems with Bindings and Free Variables I

\[
\text{eval}[ (\lambda y \ | \ (\lambda x \ | \ + \ x \ y)) \ 4, \ \{\}] \\
\text{eval}[ (\lambda x \ | \ + \ x \ y), \ \{y\leftarrow 4\}] \\
\]

- 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 \(\lambda\)-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

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
▶ Our simple interpreter actually handles this (but poorly):
  ▶ evaluate λ-body: + x y in environment (y←4)
  ▶ create new function with evaluated body (λx| ⟨BODY⟩)

eval[+ x y, {y←4}] → (+ x 4)
Problems with Bindings and Free Variables

\[
\text{eval}\left[\left(\lambda y \mid (\lambda x \mid + x y)\right) 4, \{\}\right]
\]
\[
\text{eval}\left[ (\lambda x \mid + x y), \{y \leftarrow 4\}\right]
\]

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

\[
\text{eval}[+ x y, \{y \leftarrow 4\}] \rightarrow (+ x 4) \\
\rightarrow (\lambda x \mid + x 4)
\]
Problems with Bindings and Free Variables I

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

- No application here — cannot evaluate (λx| + 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 (λx| ⟨BODY⟩)  

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

- Above solution breaks: See next slide!
Problems with Bindings and Free Variables II

\[
eval[(\lambda y \mid (\lambda y)(y \ y))] \ 4, \ {}\]

\[
\rightarrow (\lambda y \mid \eval[(y \ y)], \ {y \leftarrow 4})
\]

\[
\equiv (\lambda y \mid 4 \ 4)
\]

DO NOT DO THIS!

\[
\rightarrow (\lambda y \mid \eval[(y \ y), \ {y \leftarrow y, \ y \leftarrow 4})
\]

\[
\rightarrow (\lambda y \mid (y \ y))
\]

Dynamic binding results in wrong answer! The function problem

Could try to represent fact that \(y\) is bound in inner \(\lambda\)

\[
\rightarrow (\lambda y \mid \eval[(y \ y), \ {y \leftarrow y, \ y \leftarrow 4})
\]

\[
\rightarrow (\lambda y \mid (y \ y))
\]

Solution might break in more complex case - not sure at this point
Problems with Bindings and Free Variables II

\[
\text{eval}\left[\ (\lambda y\ | \ (\lambda y\ |\ (y\ y)))\ 4,\ {}\ \right]
\]
\[
\text{eval}\left[\ (\lambda y\ |\ (y\ y)),\ \{y\leftarrow 4\}\ \right]
\]
Problems with Bindings and Free Variables II

eval[ (\lambda y \mid (\lambda y \mid (y y))) 4, \{\} ]
eval[ (\lambda y \mid (y y)), \{y \leftarrow 4\} ]

DO NOT DO THIS!

\rightarrow (\lambda y \mid \text{eval[ ( y y), \{y \leftarrow 4\} ]})
Problems with Bindings and Free Variables II

eval[ (\lambda y \mid (\lambda y \mid (y \ y))) \ 4, \ {} ]
eval[ (\lambda y \mid (y \ y)), \ {y \leftarrow 4} ]

DO NOT DO THIS!

\Rightarrow (\lambda y \mid eval[ (y \ y), \ {y \leftarrow 4} ] )

\equiv (\lambda y \mid 4 \ 4)
Problems with Bindings and Free Variables II

\[
\begin{align*}
\text{eval} & \left[ (\lambda y \mid (\lambda y | (y \ y))) \right] 4, \{\} \] \\
\text{eval} & \left[ (\lambda y | (y \ y)) \right], \{y \leftarrow 4\} \]
\end{align*}
\]

\textbf{DO NOT DO THIS!}

\[
\rightarrow (\lambda y \mid \text{eval} \left[ (y \ y) \right], \{y \leftarrow 4\} \]
\]

\[
\equiv (\lambda y \mid 4 \ 4)
\]

- Dynamic binding results in wrong answer! The “funarg” problem
Problems with Bindings and Free Variables II

\[
\text{eval[ (λy \mid (λy\mid (y \; y))) \; 4, \{\} \; ]}
\]
\[
\text{eval[ (λy\mid (y \; y)), \{y←4\} \; ]}
\]

DO NOT DO THIS!

\[
\rightarrow (λy \mid \text{eval[ ( y \; y), \{y←4\} \; ]})
\]

\[
≡ (λy \mid 4 \; 4)
\]

- 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

\[
\text{eval} \left[ (\lambda y \mid (\lambda y \mid (y \ y))) \ 4, \ {} \right] \\
\text{eval} \left[ (\lambda y \mid (y \ y)), \ {y \leftarrow 4} \right] \\
\text{DO NOT DO THIS!} \\
\rightarrow (\lambda y \mid \text{eval} \left[ (y \ y), \ {y \leftarrow 4} \right]) \\
\equiv (\lambda y \mid 4 \ 4)
\]

» Dynamic binding results in wrong answer! The “funarg” problem

» Could try to represent fact that \( y \) is bound in inner \( \lambda \)
\[
\text{eval} \left[ (\lambda y \mid (y \ y)), \ {y \leftarrow 4} \right]
\]
Problems with Bindings and Free Variables II

\[
\text{eval}\left[ (\lambda y \mid (\lambda y \mid (y \ y))) \ 4, \ {} \right] \\
\text{eval}\left[ (\lambda y \mid (y \ y)), \ {y\leftarrow4} \right] \\
\text{DO NOT DO THIS!} \\
\rightarrow (\lambda y \mid \text{eval}\left[ (y \ y), \ {y\leftarrow4} \right]) \ ) \\
\equiv (\lambda y \mid 4 \ 4)
\]

- Dynamic binding results in wrong answer! The “funarg” problem

- Could try to represent fact that \( y \) is bound in inner \( \lambda \)

\[
\text{eval}\left[ (\lambda y \mid (y \ y)), \ {y\leftarrow4} \right] \\
\text{DO NOT DO THIS!} \\
\rightarrow (\lambda y \mid \text{eval}\left[ (y \ y), \ {y\leftarrow y, \ y\leftarrow4} \right]) \ )
\]
Problems with Bindings and Free Variables II

\[
\text{eval}\left[ (\lambda y \mid (\lambda y \mid (y \ y))) \ 4, \ {} \right] \\
\text{eval}\left[ (\lambda y \mid (y \ y)), \ {y\leftarrow 4} \right] \\
\text{DO NOT DO THIS!} \\
\quad \rightarrow (\lambda y \mid \text{eval}\left[ (y \ y), \ {y\leftarrow 4} \right] \ ) \\
\quad \equiv (\lambda y \mid 4 \ 4)
\]

- Dynamic binding results in wrong answer! The “funarg” problem

- Could try to represent fact that \( y \) is bound in inner \( \lambda \)

\[
\text{eval}\left[ (\lambda y \mid (y \ y)), \ {y\leftarrow 4} \right] \\
\text{DO NOT DO THIS!} \\
\quad \rightarrow (\lambda y \mid \text{eval}\left[ (y \ y), \ {y\leftarrow y, \ y\leftarrow 4} \right] \ ) \\
\quad \rightarrow (\lambda y \mid (y \ y))
\]
Problems with Bindings and Free Variables II

\[
\text{eval}[ (\lambda y \mid (\lambda y \mid (y \ y))) \ 4, \ {} ] \\
\text{eval}[ (\lambda y \mid (y \ y)), \ {y\leftarrow}4\ ] \\
\text{DO NOT DO THIS!} \\
\rightarrow (\lambda y \mid \text{eval}[ ( y \ y), \ {y\leftarrow}4\ ] \ ) \\
\equiv (\lambda y \mid 4 \ 4)
\]

- Dynamic binding results in wrong answer! The “funarg” problem

- Could try to represent fact that y is bound in inner \( \lambda 

\text{eval}[ (\lambda y \mid (y \ y)), \ {y\leftarrow}4\ ] \\
\text{DO NOT DO THIS!} \\
\rightarrow (\lambda y \mid \text{eval}[ ( y \ y), \ {y\leftarrow}y, \ y\leftarrow}4\ ] \ ) \\
\rightarrow (\lambda y \mid ( 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 \textit{environment} or \textit{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

\( \text{eval\[ (\lambda x \mid x) \ 2 \ ,\{}]\] \)
Simple Application with Closures

\[ \text{eval}[(\lambda x \mid x) \ 2, \{\}] \]

*Regular apply:* \text{eval} \ f1, \ \text{eval} \ a1, \ \text{apply} \ f1 \ \text{to} \ a1
Simple Application with Closures

\[
\text{eval} \left( \lambda x \mid x \right) 2 , \{\} \right]
\]

Regular apply: \text{eval } f1, \text{ eval } a1, \text{ apply } f1 \text{ to } a1

\[
f1 = \text{eval} \left( \lambda x \mid x \right) , \{\} \right]
\]
Simple Application with Closures

\[
\text{eval}[(\lambda x \mid x)^2, \{\}]
\]

Regular apply: \(\text{eval} \ f_1, \ \text{eval} \ a_1, \ \text{apply} \ f_1 \ \text{to} \ a_1\)

\[
f_1 = \text{eval}[(\lambda x \mid x), \{\}]
\]

Definition: make closure
Simple Application with Closures

eval[ (λx | x) 2 ,{}]  

*Regular apply:* eval $f_1$, eval $a_1$, apply $f_1$ to $a_1$

$$f_1 = \text{eval[ (λx | x) ,{}] }$$  
*Definition:* make closure

$$f_1 = <(λx | x) ,{}>$$
Simple Application with Closures

\[ \text{eval}[ (\lambda x \mid x)\,2\,{},{}] \]

Regular apply: \( \text{eval } f1, \text{ eval } a1, \text{ apply } f1 \text{ to } a1 \)

\[ f1 = \text{eval}[ (\lambda x \mid x),{}{}] \]

Definition: make closure

\[ f1 = \langle (\lambda x \mid x),{}\rangle \]

\[ a1 = \text{eval}[ 2 \,] = 2 \]
Simple Application with Closures

\[ \text{eval[ } (\lambda x \ | \ x) \ 2 ,\{\}] \]

Regular apply: \text{eval } f1, \text{eval } a1, \text{apply } f1 \text{ to } a1

\[ f1 = \text{eval[ } (\lambda x \ | \ x) ,\{\}] \]

Definition: make closure
\[ f1 = <(\lambda x \ | \ x) ,\{\}> \]

\[ a1 = \text{eval[ } 2 ] = 2 \]

\[ \text{apply[ } f1, \ a1 \ ] \]
Simple Application with Closures

\[ \text{eval} \left[ \lambda x \mid x \right] 2, \{\} \]

*Regular apply:* eval \( f_1 \), eval \( a_1 \), apply \( f_1 \) to \( a_1 \)

\[ f_1 = \text{eval} \left[ \lambda x \mid x \right], \{\} \]

*Definition:* make closure

\[ f_1 = < \lambda x \mid x >, \{\} > \]

\[ a_1 = \text{eval} \left[ 2 \right] = 2 \]

apply\[ f_1, a_1 \]

*Eval \( f_1 \) body in environment with \( x=a_1 \) and context of \( f_1=\{\} \)
Simple Application with Closures

\[ \text{eval[ (λx | x) 2 ,{}}] \]

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

\[ f1 = \text{eval[ (λx | x) ,{}}] \]

*Definition: make closure

\[ f1 = \langle(λx | x) ,{}}\rangle \]

\[ a1 = \text{eval[ 2 ] = 2} \]

\[ \text{apply[ f1, a1 ]} \]

*Eval } f1 body in environment

  * with } x=a1 and context of } f1={}

\[ \text{eval[ x, {x←2}+{}}] \]
Simple Application with Closures

\[
eval[(\lambda x \mid x) \, 2, \{\}] \]

Regular apply: \( \text{eval } f1, \text{ eval } a1, \text{ apply } f1 \text{ to } a1 \)

\[
f1 = \text{eval}[(\lambda x \mid x), \{\}] \]

Definition: make closure
\[
f1 = \langle (\lambda x \mid x), \{\} \rangle \]

\[
a1 = \text{eval}[2] = 2 \]

\[
\text{apply}[f1, a1] \]

Eval \( f1 \) body in environment
with \( x = a1 \) and context of \( f1 = \{\} \)
\[\text{eval}[x, \{x \leftarrow 2\}+\{\}] \]

\[\rightarrow 2\]
Simple Application with Closures

\[ \text{eval} \left[ \left( \lambda x \mid x \right) \mid 2 \right] \]

**Regular apply:** eval \( f1 \), eval \( a1 \), apply \( f1 \) to \( a1 \)

\[ f1 = \text{eval} \left[ \left( \lambda x \mid x \right) \mid \right] \]

*Definition:* make closure

\[ f1 = \langle \left( \lambda x \mid x \right) \mid \rangle \]

\[ a1 = \text{eval} \left[ 2 \right] = 2 \]

\[ \text{apply} \left[ f1, a1 \right] \]

*Eval \( f1 \) body in environment
  with \( x=a1 \) and context of \( f1=\{} \)

\[ \text{eval} \left[ x, \{x\leftarrow 2\}+\{} \right] \]

\[ \rightarrow 2 \]

- Seems like extra machinery, but useful in complex cases
Forming and Applying Closures

- Forming closures

Given definition \((\lambda p \mid \langle \text{BODY} \rangle)\) defined in environment \(E\)

We form the closure \(<(\lambda p \mid \langle \text{BODY} \rangle), E)>\)

To apply closure \(<(\lambda p \mid \langle \text{BODY} \rangle), E)>\) to argument \(A\) in context \(G\)

evaluate \(\langle \text{BODY} \rangle\) in an environment \(\{p \leftarrow A + E + G\}\)
Forming and Applying Closures

- Forming closures
  - Given definition \((\lambda p \mid \langle BODY \rangle)\) defined in environment E
Forming and Applying Closures

- Forming closures
  - Given definition \((\lambda p | \langle \text{BODY} \rangle)\) defined in environment \(E\)
  - We form the closure \(<(\lambda p | \langle \text{BODY} \rangle), E>\)
Forming and Applying Closures

- Forming closures
  - Given definition \((\lambda \mathbf{p} \mid \langle \text{BODY} \rangle)\) defined in environment \(E\)
  - We form the closure \(<(\lambda \mathbf{p} \mid \langle \text{BODY} \rangle), E>\)

- To apply closure \(<(\lambda \mathbf{p} \mid \langle \text{BODY} \rangle), E>\) to argument \(A\) in context \(G\)
Forming and Applying Closures

- Forming closures
  - Given definition \((\lambda p \mid \langle \text{BODY} \rangle)\) defined in environment \(E\)
  - We form the closure \(<(\lambda p \mid \langle \text{BODY} \rangle), E>\)

- To apply closure \(<(\lambda p \mid \langle \text{BODY} \rangle), E>\) to argument \(A\) in context \(G\)
  - evaluate \(\langle \text{BODY} \rangle\)
Forming and Applying Closures

- Forming closures
  - Given definition \((\lambda p \langle \text{BODY} \rangle)\) defined in environment \(E\)
  - We form the closure \(<(\lambda p \langle \text{BODY} \rangle),E)>\)

- To apply closure \(<(\lambda p \langle \text{BODY} \rangle),E)>\) to argument \(A\) in context \(G\)
  - evaluate \(\langle \text{BODY} \rangle\)
  - in an environment = \(\{ p \leftarrow A + E + G\}\)
Trickier Application with Closures I

LET x=1 IN LET y=(\lambda z | z+x) IN y(3)
Trickier Application with Closures I

\[
\text{LET } x=1 \text{ IN LET } y=\lambda z \mid z+x \text{ IN } y(3) \\
eval[(\lambda x \mid (\lambda y \mid (y \ 3)) \ (\lambda z \mid z+x)) \ 1, \ {}]
\]

Dr. B. Price and Dr. R. Greiner

COMPUT325: Meta-interpretation
Trickier Application with Closures I

LET \( x=1 \) IN LET \( y=(\lambda z | z+x) \) IN \( y(3) \)

\[
\text{eval}[(\lambda x | (\lambda y | (y \ 3)) \ (\lambda z | z+x)) \ 1, \ {}]
\]

*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)), {}]
LET x=1 IN LET y=(\lambda z\mid z+x) IN y(3)
eval[(\lambda x\mid (\lambda y\mid (y\ 3))\ (\lambda z\mid z+x))\ 1, \{\}]

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

f1=eval[(\lambda x\mid (\lambda y\mid (y\ 3))\ (\lambda z\mid z+x)), \{\}]

*Definition: make closure*
**Trickier Application with Closures**

LET \( x=1 \) IN LET \( y=(\lambda z|z+x) \) IN \( y(3) \)

\[
eval[(\lambda x|((\lambda y|(y \ 3)) \ (\lambda z|z+x))) \ 1, \ \{\}]
\]

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

\[
f1=eval[(\lambda x|((\lambda y|(y \ 3)) \ (\lambda z|z+x))), \ \{\}]
\]

*Definition: make closure*

\[
f1=<(\lambda x|((\lambda y|(y \ 3)) \ (\lambda z|z+x)),\{\}>
\]
Trickier Application with Closures I

LET $x=1$ IN LET $y=(\lambda z|z+x)$ IN $y(3)$

$eval[(\lambda x|(\lambda y|(y\ 3))\ (\lambda z|z+x))\ 1, \{\}]$

Regular apply, eval $f_1$, eval $a_1$, apply $f_1$ to $a_1$

$f_1=eval[(\lambda x|(\lambda y|(y\ 3))\ (\lambda z|z+x)), \{\}]$

Definition: make closure

$f_1=<(\lambda x|(\lambda y|(y\ 3))\ (\lambda z|z+x)),\{}>$

$a_1=eval[\ 1, \{\}] = 1$
Trickier Application with Closures I

LET \( x=1 \) IN LET \( y=(\lambda z\mid z+x) \) IN \( y(3) \)

\[
eval[(\lambda x\mid (\lambda y\mid (y\ 3))\ (\lambda z\mid z+x))\ 1, \ {}]
\]

Regular apply, \( \text{eval } f1, \text{ eval } a1, \text{ apply } f1 \text{ to } a1 \)

\[
f1=\text{eval}[(\lambda x\mid (\lambda y\mid (y\ 3))\ (\lambda z\mid z+x)), \ {}]
\]

Definition: make closure

\[
f1=<(\lambda x\mid (\lambda y\mid (y\ 3))\ (\lambda z\mid z+x)),{}> 
\]

\[
a1=\text{eval}[\ 1, \ {}] = 1 
\]

apply\((f1,a1)\)
Trickier Application with Closures

\[ \text{LET } x=1 \text{ IN LET } y=(\lambda z|z+x) \text{ IN } y(3) \]
\[ \text{eval}\left[\left(\lambda x\left(\lambda y\left(y 3\right)\right)\left(\lambda z|z+x\right)\right), 1, {}\right] \]

Regular apply, eval \( f_1 \), eval \( a_1 \), apply \( f_1 \) to \( a_1 \)

\[ f_1=\text{eval}\left[\left(\lambda x\left(\lambda y\left(y 3\right)\right)\left(\lambda z|z+x\right)\right), {}\right] \]

Definition: make closure

\[ f_1=<\left(\lambda x\left(\lambda y\left(y 3\right)\right)\left(\lambda z|z+x\right)\right),{}> \]
\[ a_1=\text{eval}\left[ 1, {}\right] = 1 \]
\[ \text{apply}(f_1, a_1) \]

Eval \( f_1 \) body with \( a_1 \) and context of \( f_1 \)
LET  x=1  IN LET  y=(\lambda z|z+x)  IN  y(3)
eval[(\lambda x|(\lambda y|(y 3))  (\lambda z|z+x))  1,  {}]

Regular apply,  eval f1,  eval a1,  apply f1 to  a1
f1=eval[(\lambda x|(\lambda y|(y 3))  (\lambda z|z+x)),  {}]

Definition:make closure
f1=<(\lambda x|(\lambda y|(y 3))  (\lambda z|z+x)),{}>
a1=eval[ 1,  {}] = 1
apply(f1,a1)

Eval f1 body with  a1 and context of  f1
eval[(\lambda y|(y 3))  (\lambda z|z+x),{x=1}]
Trickier Application with Closures II

\[ \text{eval}[(\lambda y|y\ 3)\ (\lambda z|z+x),\{x=1\}] \]
Trickier Application with Closures II

\[ \text{eval}[(\lambda y | y \ 3)) \ (\lambda z | z + x), \{x=1\}] \]

*Regular apply, eval f2, eval a2, apply f2 to a2*
Trickier Application with Closures II

\[ \text{eval}[(\lambda y \mid (y + 3)) \ (\lambda z \mid z + x), \{x=1\}] \]

*Regular apply, eval \(f2\), eval \(a2\), apply \(f2\) to \(a2\)*

\(f2=\text{eval}[(\lambda y \mid (y + 3)), \{x=1\}]\)

\(a2=\text{eval}[(\lambda z \mid z + x), \{x=1\}]\)

apply\(f2, a2\)
Trickier Application with Closures II

\[
eval[(\lambda y|(y \ 3)) (\lambda z|z+x), \{x=1\}] \\
\text{Regular apply, eval } f_2, \text{ eval } a_2, \text{ apply } f_2 \text{ to } a_2 \\
f_2=eval[(\lambda y|(y \ 3)), \{x=1\}] \\
\text{Definition: make closure}
\]
eval[(\(y\) | \((y\ 3)\)) \(\ (\lambda z\mid z+x)\) ,\{x=1\}] 

Regular apply,  eval f2,  eval a2,  apply f2 to a2
f2=eval[(\(y\) | \((y\ 3)\)) ,\{x=1\}] 

Definition: make closure
f2=<(\(y\) | \((y\ 3)\)) ,\{x=1\}>
Trickier Application with Closures II

\[ \text{eval}\left[(\lambda y|(y \ 3)) \ (\lambda z|z+x)\ ,\{x=1\}\right] \]

*Regular apply,  eval \(f_2\),  eval \(a_2\), apply \(f_2\) to \(a_2\)*

\(f_2=\text{eval}\left[(\lambda y|(y \ 3))\ ,\{x=1\}\right]\)

*Definition: make closure*

\(f_2=<(\lambda y|(y \ 3))\ ,\{x=1\}>\)

\(a_2=\text{eval}\left[(\lambda z|z+x)\ ,\{x=1\}\right]\)
Trickier Application with Closures II

eval[\((\lambda y \, (y \, 3)) \ (\lambda z \, z+x)\),\{x=1\}]

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

f2=eval[\((\lambda y \, (y \, 3))\),\{x=1\}]

*Definition: make closure*

f2=\((\lambda y \, (y \, 3)),\{x=1\}\rangle

a2=eval[\((\lambda z \, z+x)\),\{x=1\}]

*Definition: make closure*
Trickier Application with Closures II

\[
eval[(\lambda y|(y \ 3)) \ (\lambda z|z+x) \ ,\{x=1\}]
\]

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

\[
f2=eval[(\lambda y|(y \ 3))\ ,\{x=1\}]
\]

Definition: make closure

\[
f2=<(\lambda y|(y \ 3))\ ,\{x=1\}>
\]

\[
a2=eval[(\lambda z|z+x) \ ,\{x=1\}]
\]

Definition: make closure

\[
a2=<(\lambda z|z+x) \ ,\{x=1\}>
\]
Trickier Application with Closures II

\[
eval[(\lambda y\,(y\ 3))\ (\lambda z\,z+x)\ ,\{x=1\}]
\]

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

\[
f2=eval[(\lambda y\,(y\ 3))\ ,\{x=1\}]
\]

*Definition: make closure*

\[
f2=<(\lambda y\,(y\ 3))\ ,\{x=1\}>
\]

\[
a2=eval[(\lambda z\,z+x)\ ,\{x=1\}]
\]

*Definition: make closure*

\[
a2=<(\lambda z\,z+x)\ ,\{x=1\}>
\]

apply(f2,a2)
Trickier Application with Closures II

\[ \text{eval}\left[ (\lambda y| (y \ 3)) \ (\lambda z| z+x) , \{x=1\} \right] \]

Regular apply, eval \( f_2 \), eval \( a_2 \), apply \( f_2 \) to \( a_2 \)

\( f_2 = \text{eval}\left[ (\lambda y| (y \ 3)) , \{x=1\} \right] \)

Definition: make closure

\( f_2 = (\lambda y| (y \ 3)) , \{x=1\} \)

\( a_2 = \text{eval}\left[ (\lambda z| z+x) , \{x=1\} \right] \)

Definition: make closure

\( a_2 = (\lambda z| z+x) , \{x=1\} \)

apply\( (f_2 , a_2) \)

Eval \( f_2 \) body with \( a_2 \) and context of \( f_2 \)
Trickier Application with Closures II

\[
eval[(\lambda y|(y 3)) (\lambda z|z+x),\{x=1\}]
\]

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

f2=\(\text{eval}[(\lambda y|(y 3)),\{x=1\}]\)

**Definition: make closure**

f2=<(\lambda y|(y 3)),\{x=1}\>

a2=\(\text{eval}[(\lambda z|z+x),\{x=1\}]\)

**Definition: make closure**

a2=<(\lambda z|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

evall[(\lambda y | (y \ 3)) \ (\lambda z | z+x) , \{x=1\}]

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

\[f2=\text{eval}[(\lambda y | (y \ 3)) , \{x=1\}]\]

*Definition: make closure*

\[f2=<(\lambda y | (y \ 3)), \{x=1\}>\]

\[a2=\text{eval}[(\lambda z | z+x) , \{x=1\}]\]

*Definition: make closure*

\[a2=<(\lambda z | z+x), \{x=1\}>\]

apply(f2,a2)

*Eval f2 body with a2 and context of f2*

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

\[\text{eval}[(y \ 3), \{y=<(\lambda z | z+x), \{x=1\}>}, x=1]\]
Trickier Application with Closures III

\[
eval[(y \ 3), \{y=\langle\lambda z | z+x\rangle, \{x=1\}\}, x=1]\]

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

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

Regular apply, \(eval\ f3,\ eval\ a3,\ apply\ f3\ to\ a3\)

\[f3=eval[y,\{y=<(\lambda z | z+x),\{x=1}\>,x=1}]\]
Trickier Application with Closures III

\[ \text{eval}[y \ 3], \{y=<\lambda z | z+x>, \{x=1\}, x=1}] \]

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

f3 = eval[y, \{y=<\lambda z | z+x>, \{x=1\}, x=1}\]

f3 = <\lambda z | 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

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

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

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

\[
f3=<(\lambda z | z+x),\{x=1}\>\]

\[
a3=eval[3]=3\]

\[
apply[f3,a3]\]

\[
eval[z+x,\{z=3,x=1}\]
\]

*Regular apply...

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

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

\[
apply[f4,a4]\]

\[
eval[+ 3 1] → 4\]
Trickier Application with Closures III

```
eval[(y 3), {y=<\(\lambda z \mid z+x\), {x=1}>, x=1}]

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

Dr. B. Price and Dr. R. Greiner
COMPUT325: Meta-interpretation
Trickier Application with Closures III

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

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

Dr. B. Price and Dr. R. Greiner

COMPUT325: Meta-interpretation


Trickier Application with Closures III

\[
\text{eval[(y \ 3), \{y=<(\lambda z\mid z+x), \{x=1}\}, x=1}]
\]

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

\[
f3=\text{eval}[y, \{y=<(\lambda z\mid z+x), \{x=1}\}, x=1}]
\]

\[
f3=<(\lambda z\mid z+x), \{x=1}\>
\]

\[
a3=\text{eval}[3]=3
\]

\[
\text{apply}[f3, a3]
\]

*Eval f2 body with a2 and context of f2*

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

*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
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

\[ \text{eval}\left((y \ 3), \{y=\langle (\lambda z | z+x), \{x=1\}\rangle, x=1\}\right) \]

*Regular apply, eval \( f3 \), eval \( a3 \), apply \( f3 \) to \( a3 \)

\( f3 = \text{eval}\left(y, \{y=\langle (\lambda z | z+x), \{x=1\}\rangle, x=1\}\right) \)

\( f3 = \langle (\lambda z | z+x), \{x=1\}\rangle \)

\( a3 = \text{eval}\left[3\right]=3 \)

\( \text{apply}[f3,a3] \)

*Eval \( f2 \) body with \( a2 \) and context of \( f2 \)

\( \text{eval}[z+x, \{z=3, x=1\}] \)

*Regular apply...

\( f4 = \text{eval}[z, \{z=3, x=1\}] = 3 \)

\( a4 = \text{eval}[x, \{z=3, x=1\}] = 1 \)

\( \text{apply}[f4,a4] \)
Trickier Application with Closures III

\[ \text{eval}[(y \ 3), \{y = \langle \lambda z \mid z + x \rangle, \{x = 1\}\}, \{x = 1\}] \]

\textit{Regular apply, eval f3, eval a3, apply f3 to a3}

\[ f3 = \text{eval}[y, \{y = \langle \lambda z \mid z + x \rangle, \{x = 1\}\}, \{x = 1\}] \]
\[ f3 = \langle \lambda z \mid z + x \rangle, \{x = 1\} \]
\[ a3 = \text{eval}[3] = 3 \]
\[ \text{apply}[f3, a3] \]

\textit{Eval f2 body with a2 and context of f2}

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

\textit{Regular apply...}

\[ f4 = \text{eval}[z, \{z = 3, x = 1\}] = 3 \]
\[ a4 = \text{eval}[x, \{z = 3, x = 1\}] = 1 \]
\[ \text{apply}[f4, a4] \]
\[ \text{eval} [+ 3 1] \rightarrow 4 \]
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 correctly

\[ F \equiv (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \]
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 correctly

\[
F \equiv (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \\
Y \equiv (\lambda f \mid (\lambda x \mid f (x \ x)) \ (\lambda x \mid f (x \ x)))
\]
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 correctly

\[ F \equiv (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \]
\[ Y \equiv (\lambda f \mid (\lambda x\mid f (x x)) \ (\lambda x\mid f (x x))) \]
\[ \text{eval}[YF,{}] \]
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 correctly

\[ F \equiv (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \]
\[ Y \equiv (\lambda f \mid (\lambda x \mid f (x \ x)) \ (\lambda x \mid f (x \ x))) \]
\[ \text{eval} [YF,{}] \]
\[ \text{eval} [(\lambda f \mid (\lambda x \mid f (x \ x)) \ (\lambda x \mid f (x \ x))) F,{}] \]
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 correctly

\[ F \equiv (\lambda f \mid (\lambda n \mid \text{zerop}(n) \; 0 \; f(n-1))) \]
\[ Y \equiv (\lambda f \mid (\lambda x \mid f(x \; x)) \; (\lambda x \mid f(x \; x))) \]
\[ \text{eval}[YF,{}] \]
\[ \text{eval}[(\lambda f \mid (\lambda x \mid f(x \; x)) \; (\lambda x \mid f(x \; x))) \; F,{}] \]
\[ \text{eval}[(\lambda x \mid f(x \; x)) \; (\lambda x \mid f(x \; x)), \{f\leftarrow F\}] \]
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 correctly

\[ F \equiv (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \]
\[ Y \equiv (\lambda f \mid (\lambda x \mid f(x \ x)) \ (\lambda x \mid f(x \ x))) \]
\[ \text{eval}[YF,{}] \]
\[ \text{eval}[((\lambda f \mid (\lambda x \mid f(x \ x)) \ (\lambda x \mid f(x \ x))) \ F,{}] \]
\[ \text{eval}[(\lambda x \mid f(x \ x)) \ (\lambda x \mid f(x \ x)), \ {f \leftarrow F}] \]
\[ \rightarrow (\lambda x \mid F(x \ x)) \ (\lambda x \mid F(x \ x)) \]
Applicative order reduction blows up with Combinator \( \text{Y} \)

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

Bindings evaluate Fixed-Point Combinator correctly

\[
F \equiv (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)))
\]

\[
Y \equiv (\lambda f \mid (\lambda x \mid f (x \ x)) \ (\lambda x \mid f (x \ x)))
\]

\[
\text{eval}[\text{Y}F, \{}\]
\]

\[
\text{eval}[(\lambda f \mid (\lambda x \mid f (x \ x)) \ (\lambda x \mid f (x \ x))) \ F, \{}\]
\]

\[
\text{eval}[(\lambda x \mid f (x \ x)) \ (\lambda x \mid f (x \ x)), \ {f \leftarrow F}]\]
\]

\[
\rightarrow (\lambda x \mid F (x \ x)) \ (\lambda x \mid F (x \ x))
\]

\[
\equiv \langle \text{Y}F \rangle
\]
Bindings and Recursion II

eval[ (\lambda f \ | \ (\lambda n \ | \ zerop(n) \ 0 \ f(n-1))) \ \langle YF \rangle \ 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⟩}]
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[ (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[ (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \langle YF \rangle \ 1, \ {}]\]
eval[ (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)) \ 1, \ {f \leftarrow \langle YF \rangle } ]
eval[ \text{zerop}(n) \ 0 \ f(n-1), \ {n \leftarrow 1,f \leftarrow \langle YF \rangle } ]
  \quad \text{eval[ zerop}(n) , \ {n \leftarrow 1,f \leftarrow \langle YF \rangle } ] \rightarrow \ F
\quad \text{eval[ f}(n-1), \ {n \leftarrow 1,f \leftarrow \langle YF \rangle } ]
Bindings and Recursion II

\[
\text{eval}\left[ \left( \lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)) \right) \langle YF \rangle \right] 1, \ \{\}\]
\[
\text{eval}\left[ (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)) \right] 1, \ \{f \leftarrow \langle YF \rangle\}]
\[
\text{eval}\left[ \text{zerop}(n) \ 0 \ f(n-1), \ \{n \leftarrow 1,f \leftarrow \langle YF \rangle\}\right]
\]
\[
\text{eval}\left[ \text{zerop}(n), \ \{n \leftarrow 1,f \leftarrow \langle YF \rangle\}\right] \rightarrow F
\]
\[
\text{eval}\left[ f(n-1), \ \{n \leftarrow 1,f \leftarrow \langle YF \rangle\}\right]
\]
\[
\text{eval}\left[ \langle YF \rangle, \ \{n \leftarrow 1,f \leftarrow \langle YF \rangle\}\right] \rightarrow F \langle YF \rangle
\]
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[(\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \langle YF \rangle \ 1, \ {}]

eval[(\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)) \ 1, \ {f \leftarrow \langle YF \rangle}]

eval[\text{zerop}(n) \ 0 \ f(n-1), \ {n \leftarrow 1, f \leftarrow \langle YF \rangle}]
  
  eval[\text{zerop}(n), \ {n \leftarrow 1, f \leftarrow \langle YF \rangle}] \rightarrow F

  eval[f(n-1), \ {n \leftarrow 1, f \leftarrow \langle YF \rangle}]
    
    eval[\langle YF \rangle, \ {n \leftarrow 1, f \leftarrow \langle YF \rangle}] \rightarrow F \langle YF \rangle

    eval[n-1, \ {n \leftarrow 1, f \leftarrow \langle YF \rangle} \ ] \rightarrow 0

  eval[F <YF> \ 0, \ {n \leftarrow 1, f \leftarrow \langle YF \rangle} \ ]
Process repeats

\[
eval[ (\lambda f \mid (\lambda n \mid \text{zerop}(n) 0 f(n-1))) \langle YF \rangle 0, \\
\{n\leftarrow 1, f\leftarrow\langle YF \rangle\} ]
\]
Process repeats

\[
\text{eval}\left[\left(\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))\right) \langle YF \rangle \ 0, \{n\leftarrow 1, f\leftarrow \langle YF \rangle\}\right]
\]

\[
\text{eval}\left[\left(\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)\right) \ 0, \{f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle\}\right]
\]
Process repeats

\[
\text{eval} \left[ (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \ 0, \{n\leftarrow 1, f\leftarrow \langle YF \rangle \} \right]
\]

\[
\text{eval} \left[ (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)) \ 0, \{f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle \} \right]
\]

\[
\text{eval} \left[ \text{zerop}(n) \ 0 \ f(n-1) \ 0, \{n\leftarrow 0, f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle \} \right]
\]
Bindings and Recursion III

- Process repeats

\[
\text{eval}
(\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \ \langle YF \rangle \ 0,
\{n\leftarrow 1, f\leftarrow \langle YF \rangle \}
\]

\[
\text{eval}
(\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)) \ 0,
\{f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle \}
\]

\[
\text{eval}
\text{zerop}(n) \ 0 \ f(n-1) \ 0,
\{n\leftarrow 0, f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle \}
\]

\[
\text{eval}
\text{zerop}(n), \ {n\leftarrow 0, f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle}
\]

Dr. B. Price and Dr. R. Greiner
COMPUT325: Meta-interpretation
Process repeats

\[
\text{eval}[ (λf | (λn | \text{zerop}(n) 0 f(n-1))) \langle YF \rangle 0, \\
{n←1,f←\langle YF \rangle} ]
\]

\[
\text{eval}[ (λn | \text{zerop}(n) 0 f(n-1)) 0, \\
{f←\langle YF \rangle,n←1,f←\langle YF \rangle}]\]

\[
\text{eval}[ \text{zerop}(n) 0 f(n-1) 0, \\
{n←0,f←\langle YF \rangle,n←1,f←\langle YF \rangle}]\]
\[
\text{eval}[ \text{zerop}(n), {n←0,f←\langle YF \rangle,n←1,f←\langle YF \rangle}]\]
\[
\text{eval}[ \text{zerop}(0), \\
{n←0,f←\langle YF \rangle,n←1,f←\langle YF \rangle}] → 0
\]
Bindings and Recursion III

- Process repeats

\[
\text{eval}\left[ (\lambda f \mid (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1))) \ YF \ 0, \{n\leftarrow 1, f\leftarrow \langle YF \rangle\} \right]
\]

\[
\text{eval}\left[ (\lambda n \mid \text{zerop}(n) \ 0 \ f(n-1)) \ 0, \{f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle\}\right]
\]

\[
\text{eval}\left[ \text{zerop}(n) \ 0 \ f(n-1) \ 0, \{n\leftarrow 0, f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle\}\right]
\]

\[
\text{eval}\left[ \text{zerop}(n), \{n\leftarrow 0, f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle\}\right]
\]

\[
\text{eval}\left[ \text{zerop}(0), \{n\leftarrow 0, f\leftarrow \langle YF \rangle, n\leftarrow 1, f\leftarrow \langle YF \rangle\}\right] \rightarrow 0
\]

\[
\text{eval}[0] \rightarrow 0
\]
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

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

- Normal order passes unreduced arguments to functions
- 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
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

\[
(\lambda x \, | \, x) \, (\lambda y \, | \, 3) \, 1 \\
≡ \ [\,(\lambda y \, | \, 3)/x\,] \, x \, 1 \\
≡ \ (\lambda y \, | \, 3) \, 1 \\
≡ \ 3
\]

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
    \[(\lambda x \mid x) (\lambda y \mid 3) 1\]
    \[\equiv [(\lambda y \mid 3)/x] x 1\]
    \[\equiv (\lambda y \mid 3) 1\]
    \[\equiv 3\]
  - Normal order passes unreduced arguments to functions
    \[\text{'+ } f(y)=3 1\]
  - 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
    \[
    (\lambda x \ | \ x) (\lambda y \ | \ 3) 1 \\
    \equiv [(\lambda y \ | \ 3)/x] x 1 \\
    \equiv (\lambda y \ | \ 3) 1 \\
    \equiv 3
    \]
  - Normal order passes unreduced arguments to functions
    \'+ f(y)=3 1
  - Efficient specialized functions cannot accept arbitrary expressions as arguments
  - 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:
    \[
    (\lambda x \ | \ x) (\lambda y \ | \ 3) 1 \\
    \equiv [(\lambda y \ | \ 3)/x] \ x \ 1 \\
    \equiv (\lambda y \ | \ 3) 1 \\
    \equiv 3
    \]
  - Normal order passes unreduced arguments to functions
    \[
    ' + f(y)=3 \ 1
    \]
  - Efficient specialized functions cannot accept arbitrary expressions as arguments
  - 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 recurring 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 recurring 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 recurring 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
Assume we change each application to a closure before application
Assume we change each application to a closure before application

\[ E \equiv \text{LETREC } f = \langle \text{BODY} \rangle \text{ IN } \langle \text{EXPR} \rangle \]
Assume we change each application to a closure before application

\[ E \equiv \text{LETREC } f = \langle \text{BODY} \rangle \text{ IN } \langle \text{EXPR} \rangle \]

\[ C \equiv \langle \text{BODY} \rangle , \{ f \leftarrow C \} \]
Assume we change each application to a closure before application

\[ E \equiv \text{LETREC } f = \langle \text{BODY} \rangle \text{ IN } \langle \text{EXPR} \rangle \]

\[ C \equiv \langle \langle \text{BODY} \rangle, \{ f \leftarrow C \} \rangle \]

\[ E \equiv \langle \langle \text{EXPR} \rangle, \{ f \leftarrow C \} \rangle \]
Closures and Recursion IV

\[
\text{LETREC } z(n) = \text{IF zerop}(n) \ 0 \ z(n-1) \ \text{IN} \ z(1)
\]
Closures and Recursion IV

LETREC \( z(n) = \text{if } \text{zerop}(n) 0 z(n-1) \text{ in } z(1) \)

\( C \equiv \langle (\lambda n. \text{if } \text{zerop}(n) 0 z(n-1)), \{z\leftarrow C\} \rangle \)

\( \text{Dr. B. Price and Dr. R. Greiner} \)

COMPUT325: Meta-interpretation
Closures and Recursion IV

LETREC $z(n) = \text{IF zero}(n) 0 z(n-1) \text{ IN } z(1)$

$C \equiv \langle (\lambda n|\text{IF zero}(n) 0 z(n-1)), \{z \leftarrow C}\rangle$

$E \equiv \langle (z \ 1), \{z \leftarrow C}\rangle$
Closures and Recursion IV

\[
\text{LETREC } z(n) = \text{IF zerop}(n) 0 z(n-1) \text{ IN } z(1)
\]

\[
C \equiv <(\lambda n | \text{IF zerop}(n) 0 z(n-1)), \{z \leftarrow C}\>
\]

\[
E \equiv <(z \ 1), \{z \leftarrow C}\>
\]

\[
\text{eval}[E, \{\}] \]

\[
\text{eval}[ <(z \ 1), \{z \leftarrow C}\> , \{\}]
\]

\[
\text{eval}[\text{IF zerop}(n) 0 z(n-1), \{n \leftarrow 1, z \leftarrow C\}]
\]
Closures and Recursion IV

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

\[ C \equiv <(\lambda n.|IF \ \text{zerop}(n) \ 0 \ z(n-1)), \{z\leftarrow C}\> \]

\[ E \equiv <(z \ 1), \{z\leftarrow C}\> \]

\[
\text{eval}[E,\{}\]
\[
\text{eval}[<(z \ 1), \{z\leftarrow C}\>, \{}\]
\]
Closures and Recursion IV

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

\[ C \equiv \langle (\lambda n. \text{IF zerop}(n) \ 0 \ z(n-1)), \{z\leftarrow C}\rangle \]

\[ E \equiv \langle (z \ 1), \{z\leftarrow C}\rangle \]

\[
\text{eval}[E,{}]
\text{eval}[ \langle (z \ 1), \{z\leftarrow C}\rangle, \{\}\]
\text{eval}[ (z \ 1), \{z\leftarrow C}\] ;; application \((f1 \ a1)\)
Closures and Recursion IV

\[
\text{LETREC } z(n) = \text{IF } \text{zerop}(n) \ 0 \ z(n-1) \ \text{IN} \ z(1)
\]

\[
C \equiv <(\lambda n | \text{IF} \ \text{zerop}(n) \ 0 \ z(n-1)), \ {z←C}>
\]

\[
E \equiv <(z \ 1), \ {z←C}>
\]

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

\[
f1 = \text{eval}[<(\lambda n | \text{IF} \ \text{zerop}(n) \ 0 \ z(n-1)), \ {z←C}>, \ {z←C}>]
\]
Closures and Recursion IV

\[
\text{LETREC } z(n) = \text{IF} \ \text{zerop}(n) \ 0 \ z(n-1) \ \text{IN} \ z(1)
\]

\[
C \equiv \langle \lambda n \mid \text{IF} \ \text{zerop}(n) \ 0 \ z(n-1) \rangle, \ \{z \leftarrow C\rangle
\]

\[
E \equiv \langle z \ 1 \rangle, \ \{z \leftarrow C\rangle
\]

\[
\text{eval\[E,\{}\]}\]
\[
\text{eval\[ \langle z \ 1 \rangle, \ \{z \leftarrow C\rangle, \ \{\}\]}\]
\[
\text{eval\[ (z \ 1), \ \{z \leftarrow C\}] \ ;; \ application \ (f1 \ a1)\]
\[
\quad f1=\text{eval\[ \langle \lambda n \mid \text{IF} \ \text{zerop}(n) \ 0 \ z(n-1) \rangle, \ \{z \leftarrow C\rangle, \ \{z \leftarrow C\rangle\]
\[
\quad f1=\langle \lambda n \mid \text{IF} \ \text{zerop}(n) \ 0 \ z(n-1) \rangle, \ \{z \leftarrow C\rangle
\]
Closures and Recursion IV

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

\[ C \equiv <(\lambda n \mid \text{IF zerop}(n) 0 z(n-1)), \{z \leftarrow C}\> \]

\[ E \equiv <(z 1), \{z \leftarrow C}\> \]

\[
\text{eval}[E,\{\}]
\]
\[
\text{eval}[ <(z 1), \{z \leftarrow C}\>, \{\}]
\]
\[
\text{eval}[ (z 1), \{z \leftarrow C\}] \; ;; \; \text{application (f1 a1)}
\]
\[
f1=\text{eval}[ <(\lambda n \mid \text{IF zerop}(n) 0 z(n-1)), \{z \leftarrow C\}, \{z \leftarrow C\}]
\]
\[
f1=<(\lambda n \mid \text{IF zerop}(n) 0 z(n-1)), \{z \leftarrow C\}>
\]
\[
a1=\text{eval}[1]=1
\]
Closures and Recursion IV

\[\text{LETREC } z(n) = \text{IF zerop}(n) \ 0 \ z(n-1) \ \text{IN } z(1)\]

\[C \equiv (\lambda n | \text{IF zerop}(n) \ 0 \ z(n-1)), \{z \leftarrow C\}\]

\[E \equiv (z \ 1), \{z \leftarrow C\}\]

\[\text{eval}[E,{}]\]
\[\text{eval}[ (z \ 1), \{z \leftarrow C\}, \{\} ]
\[\text{eval}[ (z \ 1), \{z \leftarrow C\} ] \quad \text{;; application (f1 a1)}
\quad f1 = \text{eval}[ (\lambda n | \text{IF zerop}(n) \ 0 \ z(n-1)), \{z \leftarrow C\}, \{z \leftarrow C\}]
\quad f1 = (\lambda n | \text{IF zerop}(n) \ 0 \ z(n-1)), \{z \leftarrow C\}
\quad a1 = \text{eval}[1] = 1
\quad \text{apply}[f1, a1]
\]

\textit{applying a closure, get body, add parm to env}
Closures and Recursion IV

\text{LETREC } z(n) = \text{IF zerop(n) 0 } z(n-1) \text{ IN } z(1)

\text{C } \equiv \langle (\lambda n | \text{IF zerop(n) 0 } z(n-1)), \{ z \leftarrow \text{C} \} \rangle

\text{E } \equiv \langle (z \ 1), \{ z \leftarrow \text{C} \} \rangle

\text{eval[E,\{\}]} \text{; application (f1 a1)}

f1 = \text{eval[ } \langle (\lambda n | \text{IF zerop(n) 0 } z(n-1)), \{ z \leftarrow \text{C} \} \rangle, \{ z \leftarrow \text{C} \} \rangle

a1 = \text{eval[1]} = 1

\text{apply[f1,a1]}

\text{applying a closure, get body, add parm to env}

\text{eval[IF zerop(n) 0 } z(n-1),\{ n \leftarrow 1, z \leftarrow \text{C} \} \rangle
eval[IF zerop(n) 0 z(n-1),{n←1,z←C} ]

*Application:* evaluate the arguments

Since zerop evaluates to false, 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}>]
Closures and Recursion V

\[ \text{eval}\left[ \text{IF } \text{zerop}(n)\ 0\ \text{z}(n-1),\{n\leftarrow 1,z\leftarrow C}\right] \]

*Application: evaluate the arguments*

\[ \text{eval}\left[ \text{zerop}(n),\{n\leftarrow 1,z\leftarrow C}\right] \rightarrow F \]
Closures and Recursion V

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

*Application: evaluate the arguments*

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

\textit{Application: evaluate the arguments}

\begin{align*}
\text{eval[ zerop(n) , \{} & n ← 1, z ← C \}\} → F \\
\text{eval[ 0,\{} & n ← 1, z ← C \}\} → 0 \\
\text{eval[ z(n-1) ,\{} & n ← 1, z ← C}\]
\end{align*}

\textit{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
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}> ]
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

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

**Application: evaluate the arguments**

\[ \text{eval[ zerop(n) ,}\{\text{n←1, z←C}\} ] \rightarrow F \]
\[ \text{eval[ 0,}\{\text{n←1, z←C}\} ] \rightarrow 0 \]
\[ \text{eval[ z(n-1) ,}\{\text{n←1, z←C}\}] \]

**Application: evaluate the arguments**

\[ \text{eval[ n-1,}\{\text{n←1, z←C}\} ] \rightarrow 0 \]
\[ \rightarrow <z,\{\text{n←0, n←1, z←C}\}> \]

*Since zerop evaluates to false, evaluate recursive term*

\[ \text{eval[<z,}\{\text{n←0, n←1, z←C}\}> ] \]

*z will get value from context again.*

**Recursion emerges from self-reference**

\[ \text{eval[IF zerop(n) 0 z(n-1),}\{\text{n←0, n←1, z←C}\}> ] \]
Meta-interpretation of Lisp

- Abstract Programming gives structure to a $\lambda$-calculus formula
  (IF zerop(n) THEN 1 ELSE (2*n))
Meta-interpretation of Lisp

- Abstract Programming gives structure to a $\lambda$-calculus formula
  \[(\text{IF } \text{zerop}(n) \text{ THEN } 1 \text{ ELSE } (2*n))\]

- Structure is illusory: translated into a large tangled $\lambda$-calculus expression
Meta-interpreteration of Lisp

- Abstract Programming gives structure to a $\lambda$-calculus formula
  \[(\text{IF zero}(n) \text{ THEN } 1 \text{ ELSE } (2*n))\]
- Structure is illusory: translated into a large tangled $\lambda$-calculus expression
- So we need general mechanisms like closures to help with recursion
Meta-interpretation of Lisp

- Abstract Programming gives structure to a $\lambda$-calculus formula
  \[
  (\text{IF } \text{zerop}(n) \text{ THEN } 1 \text{ ELSE } (2*n))
  \]
- Structure is illusory: translated into a large tangled $\lambda$-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 $\lambda$-calculus formula
  
  \[
  \text{IF zero\text{p}(n) THEN 1 ELSE (2*n)}
  \]

- Structure is illusory: translated into a large tangled $\lambda$-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 \( \lambda \)-calculus formula
  \[
  \text{IF zerop}(n) \ \text{THEN} \ 1 \ \text{ELSE} \ (2 \ast n)
  \]
- Structure is illusory: translated into a large tangled \( \lambda \)-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 reserve closures in LISP for
  - function definitions to preserve lexical scope
  - implementation of lazy evaluation