CMPUT 325: Abstract Programming

Dr. B. Price and Dr. G. Greiner

19th October 2004
Abstract Programming

- λCalculus has precise semantics, simple syntax, simple evaluation
Abstract Programming

- \( \lambda \)Calculus has precise semantics, simple syntax, simple evaluation
- Its also extremely tedious
Abstract Programming

- λCalculus has precise semantics, simple syntax, simple evaluation
- It's also extremely tedious
- Standard idioms for many high-level control constructs
Abstract Programming

- $\lambda$-Calculus has precise semantics, simple syntax, simple evaluation
- It’s also extremely tedious
- Standard idioms for many high-level control constructs
- Use abstract idioms in place of $\lambda$-calculus
  - Easy to read
  - Guaranteed semantics and simple evaluation
Abstract Programming

- $\lambda$Calculus has precise semantics, simple syntax, simple evaluation
- It's also extremely tedious
- Standard idioms for many high-level control constructs
- Use abstract idioms in place of $\lambda$-calculus
  - Easy to read
  - Guaranteed semantics and simple evaluation
- Simple parser converts abstractions to idioms
Abstract Programming

- \( \lambda \)Calculus has precise semantics, simple syntax, simple evaluation
- It's also extremely tedious
- Standard idioms for many high-level control constructs
- Use abstract idioms in place of \( \lambda \)-calculus
  - Easy to read
  - Guaranteed semantics and simple evaluation
- Simple parser converts abstractions to idioms
- \( \lambda \)Calculus solves problem
Abstract Programming: Datatypes

- **Numbers**: use Church’s 2 arg function representation
  - **Integers**: \( n \equiv (\lambda \ s \ z \ | \ s^k \ z) \)
    where \( s^k \) is a string of \( k \ s \)’s
Abstract Programming: Datatypes

- Numbers: use Church’s 2 arg function representation
  - Integers: \( n \equiv (\lambda \ s \ z \ | \ s^k \ z) \)
    where \( s^k \) is a string of \( k \) s’s

- Boolean values: \( T \equiv (\lambda \ c \ d \ | \ c) \) and \( F \equiv (\lambda \ c \ d \ | \ d) \)
Abstract Programming: Datatypes

- **Numbers**: use Church’s 2 arg function representation
  - **Integers**: \( n \equiv (\lambda \, s \, z \mid s^k \, z) \)
    where \( s^k \) is a string of \( k \) \( s \)'s

- **Boolean values**: \( T \equiv (\lambda c \, d \mid c) \) and \( F \equiv (\lambda c \, d \mid d) \)

- **List**
  - **Cons cell** (\( M \cdot N \)): \( (\lambda z \mid z \, m \, n) \)
  - **List** (\( a \, b \, c \, 0 \)): \( (\lambda z \mid z \, a \, (\lambda z \mid z \, b \, (\lambda z \mid z \, c \, 0))) \)

- **String**: treat characters as an integer in base 256
  - Each character replaced by ASCII value
  - **HELLO** \( \equiv H \times 256^4 + E \times 256^3 + L \times 256^2 + L \times 256^1 + O \times 256^0 \)
Abstract Programming: Datatypes

- **Numbers**: use Church’s 2 arg function representation
  - **Integers**: \( n \equiv (\lambda \text{ s z } | \text{ s}^k \text{ z}) \)
    where \( \text{s}^k \) is a string of \( k \) s’s
- **Boolean values**: \( T \equiv (\lambda \text{ c d} | \text{ c}) \) and \( F \equiv (\lambda \text{ c d} | \text{ d}) \)
- **List**
  - Cons cell \((M . N) \equiv (\lambda \text{ z } | \text{ z m n})\)
  - List \((a b c 0)\): \((\lambda \text{ z } | \text{ z a} (\lambda \text{ z } | \text{ z b} (\lambda \text{ z } | \text{ z c 0})))\)
- **String**: treat chars as an integer in base 256
  - Each char replaced by ASCII value
  - HELLO \( \equiv H \times 256^4 + E \times 256^3 + L \times 256^2 + L \times 256^1 + O \times 256^0 \)
Abstract Programming: Functions

- Assume primitive operators on datatypes defined
  - Mathematical ops: add, sub, mul, div, zerop
  - List ops: cons, car, cdr
  - Boolean operators: and, or, not
Abstract Programming: Functions

- Assume primitive operators on datatypes defined
  - Mathematical ops: add, sub, mul, div, zerop
  - List ops: cons, car, cdr
  - Boolean operators: and, or, not

- Allow standard mathematical notations

\[ \lambda x y | (\lambda s z | x s (y s z)) \equiv 1 + 2 \equiv (\lambda x y | \cdot \cdot \cdot) x \]

\[ \text{square}(2) \equiv (\lambda y | (* y y)) 2 \equiv (\lambda y | \langle \text{multiplication-idiom} \rangle) 2 \]
Abstract Programming: Functions

- Assume primitive operators on datatypes defined
  - Mathematical ops: add, sub, mul, div, zeron
  - List ops: cons, car, cdr
  - Boolean operators: and, or, not

- Allow standard mathematical notations
  - Infix notation:
    \[
    1 + 2 \equiv (+ 1 2)
    \equiv (\lambda x y \mid (\lambda s z \mid x s (y s z)) ) 1 2
    \]
Abstract Programming: Functions

- Assume primitive operators on datatypes defined
  - Mathematical ops: add, sub, mul, div, zerop
  - List ops: cons, car, cdr
  - Boolean operators: and, or, not

- Allow standard mathematical notations
  - Infix notation:
    \[ 1+2 \equiv (+ \ 1 \ 2) \]
    \[ \equiv (\lambda x y \mid (\lambda s z \mid x s (y s z)) \ ) \ 1 \ 2 \]
  - Functional notation:
    \[ f(x) \equiv (\lambda y \mid \ldots \ ) \ x \]
    \[ \text{square}(2) \equiv (\lambda y \mid (* y y)) \ 2 \]
    \[ \equiv (\lambda y \mid \langle \text{multiplication-idiom} \rangle) \ 2 \]
Conditionals

IF x<0 THEN -x ELSE x

▶ λ-calculus translation?

NOTE: must have both THEN and ELSE clauses. Why?

λ-calculus predicates resolve to T or F

T chooses first argument
F chooses second argument

Must have an argument for each case or program will behave strangely
Conditionals

IF \( x < 0 \) THEN \(-x\) ELSE \(x\)

\[\lambda\text{-calculus translation?}\]

\((\lambda xyz | xyz) x < 0 \ -x \ x\)
Conditionals

IF \( x<0 \) THEN \(-x\) ELSE \(x\)

\(\lambda\)-calculus translation?

\((\lambda xyz \mid xyz) x<0 \ -x \ x\)

NOTE: must have both THEN and ELSE clauses. Why?
Conditionals

IF \( x < 0 \) THEN \(-x\) ELSE \(x\)

- \(\lambda\)-calculus translation?

\((\lambda xyz \mid xyz) x < 0 \ -x\ x\)

- NOTE: must have both THEN and ELSE clauses. Why?

- \(\lambda\)-calculus predicates resolve to T or F
  - T chooses first argument
  - F chooses second argument
Conditionals

IF x<0 THEN -x ELSE x

▶ λ-calculus translation?

(λxyz |xyz) x<0 -x x

▶ NOTE: must have both THEN and ELSE clauses. Why?

▶ λ-calculus predicates resolve to T or F
  ▶ T chooses first argument
  ▶ F chooses second argument

▶ Must have an argument for each case or program will behave strangely
Special Forms: LET by Examples

▶ In abstract programming we define “LET AND IN” special form

LET x=5 IN x+1 →
In abstract programming we define “LET AND IN” special form

LET x=5 IN x+1 → 6
Special Forms: LET by Examples

- In abstract programming we define “LET AND IN” special form
  
  LET x=5 IN x+1 → 6

  LET x=2 IN LET y=2 IN x+y →
Special Forms: LET by Examples

- In abstract programming we define "LET AND IN" special form

\[
\text{LET } x=5 \text{ IN } x+1 \rightarrow 6
\]

\[
\text{LET } x=2 \text{ IN LET } y=2 \text{ IN } x+y \rightarrow 4
\]
Special Forms: LET by Examples

- In abstract programming we define “LET AND IN” special form

\[
\text{LET } x = 5 \text{ IN } x + 1 \rightarrow 6
\]

\[
\text{LET } x = 2 \text{ IN LET } y = 2 \text{ IN } x + y \rightarrow 4
\]

\[
\text{LET } x = 2 \text{ AND } y = 2 \text{ IN } x + y \rightarrow
\]
Special Forms: LET by Examples

In abstract programming we define “LET AND IN” special form

LET x=5 IN x+1 → 6

LET x=2 IN LET y=2 IN x+y → 4

LET x=2 AND y=2 IN x+y → 4
In abstract programming we define “LET AND IN” special form

LET x=5 IN x+1 \rightarrow 6

LET x=2 IN LET y=2 IN x+y \rightarrow 4

LET x=2 AND y=2 IN x+y \rightarrow 4

LET f(x)=x*x AND y=3 IN
  LET x=f(y) IN
    x
  \rightarrow
Special Forms: LET by Examples

- In abstract programming we define “LET AND IN” special form

\[
\text{LET } x=5 \text{ IN } x+1 \rightarrow 6
\]

\[
\text{LET } x=2 \text{ IN LET } y=2 \text{ IN } x+y \rightarrow 4
\]

\[
\text{LET } x=2 \text{ AND } y=2 \text{ IN } x+y \rightarrow 4
\]

\[
\text{LET } f(x)=x\times x \text{ AND } y=3 \text{ IN}
\]
\[\text{LET } x=f(y) \text{ IN}
\] \[x\]
\[\rightarrow 9\]
Special Forms: LET Semantics I

\[ \text{LET } x = \langle E \rangle \text{ IN } \langle \text{BODY} \rangle \]

- \( \lambda \) calculus translation?
Special Forms: LET Semantics I

LET x = ⟨E⟩ IN ⟨BODY⟩
▶ λ calculus translation? (λx | ⟨BODY⟩) ⟨E⟩
Special Forms: LET Semantics I

\[
\text{LET } x = \langle E \rangle \text{ IN } \langle \text{BODY} \rangle
\]

▶ \( \lambda \) calculus translation? \((\lambda x | \langle \text{BODY} \rangle ) \langle E \rangle\)

\[
\text{LET } x = \langle E \rangle \text{ IN}
\]

\[
\text{LET } y = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle
\]

▶ \( \lambda \) calculus translation?
Special Forms: LET Semantics I

\[
\text{LET } x = \langle E \rangle \text{ IN } \langle \text{BODY} \rangle \\
\quad \lambda \text{ calculus translation? } (\lambda x | \langle \text{BODY} \rangle ) \langle E \rangle
\]

\[
\text{LET } x = \langle E \rangle \text{ IN } \\
\quad \text{LET } y = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle \\
\quad \lambda \text{ calculus translation? } \\
\quad (\lambda x | (\lambda y | \langle \text{BODY} \rangle ) \langle E \rangle ) \langle F \rangle
\]
Special Forms: LET Semantics I

\[
\text{LET } x = \langle E \rangle \text{ IN } \langle \text{BODY} \rangle \\
\quad \triangleright \lambda \text{ calculus translation? } (\lambda x \mid \langle \text{BODY} \rangle ) \langle E \rangle
\]

\[
\text{LET } x = \langle E \rangle \text{ IN } \\
\quad \text{LET } y = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle \\
\quad \triangleright \lambda \text{ calculus translation? } \\
\quad \quad (\lambda x \mid (\lambda y \mid \langle \text{BODY} \rangle ) \langle E \rangle ) \langle F \rangle
\]

\[
\text{LET } x = \langle E \rangle \text{ AND } y = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle \\
\quad \triangleright \lambda \text{ calculus translation? }
\]
LET $x = \langle E \rangle$ IN $\langle \text{BODY} \rangle$

- $\lambda$ calculus translation? $(\lambda x \mid \langle \text{BODY} \rangle) \langle E \rangle$

LET $x = \langle E \rangle$ IN

- LET $y = \langle F \rangle$ IN $\langle \text{BODY} \rangle$

- $\lambda$ calculus translation?
  $(\lambda x \mid (\lambda y \mid \langle \text{BODY} \rangle) \langle E \rangle) \langle F \rangle$

LET $x = \langle E \rangle$ AND $y = \langle F \rangle$ IN $\langle \text{BODY} \rangle$

- $\lambda$ calculus translation? Parallel substitution
  $(\lambda xy \mid \langle \text{BODY} \rangle)$
Special Forms: LET Semantics I

\[ \text{LET } x = \langle E \rangle \text{ IN } \langle \text{BODY} \rangle \]

\[ \text{\textit{\smaller \lambda calculus translation? } } (\lambda x | \langle \text{BODY} \rangle ) \langle E \rangle \]

\[ \text{LET } x = \langle E \rangle \text{ IN} \]

\[ \text{LET } y = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle \]

\[ \text{\textit{\smaller \lambda calculus translation? } } \]

\[ (\lambda x| (\lambda y| \langle \text{BODY} \rangle ) \langle E \rangle ) \langle F \rangle \]

\[ \text{LET } x = \langle E \rangle \text{ AND } y = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle \]

\[ \text{\textit{\smaller \lambda calculus translation? Parallel } } \]

\[ \text{\textit{\smaller substitution} } \]

\[ (\lambda xy| \langle \text{BODY} \rangle ) \langle E \rangle \langle F \rangle \]
Special Forms: LET Semantics II

\[
\text{LET } x = \langle E \rangle \text{ IN LET } x = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle
\]

▶ \(\lambda\) calculus translation?

\[
\begin{align*}
\lambda x&| (\lambda x| & \langle \text{BODY} \rangle) \\
\langle E \rangle &\quad &\langle F \rangle \\
\text{LET } f(x) = \langle E \rangle \text{ IN } &\langle \text{BODY} \rangle &\quad &\text{\(\lambda\)-calculus gives precise meaning to each case of LET}
\end{align*}
\]

Dr. B. Price and Dr. G. Greiner

CMPUT 325: Abstract Programming
Special Forms: LET Semantics II

\[
\text{LET } x = \langle E \rangle \text{ IN LET } x = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle
\]

▶ \(\lambda\)-calculus translation?

\[
(\lambda x| (\lambda x| \langle \text{BODY} \rangle) \langle E \rangle) \langle F \rangle
\]
Special Forms: LET Semantics II

\[\text{LET } x = \langle E \rangle \text{ IN LET } x = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle\]

\[\lambda \text{ calculus translation?}\]

\[(\lambda x | (\lambda x | \langle \text{BODY} \rangle | \langle E \rangle) | \langle F \rangle)\]

\[\text{LET } x = \langle E \rangle\]
\[\text{LET } x = \langle F \rangle \text{ AND } y = x \text{ IN } \langle \text{BODY} \rangle\]

\[\lambda \text{ calculus translation?}\]
Special Forms: LET Semantics II

LET x = ⟨E⟩ IN LET x = ⟨F⟩ IN ⟨BODY⟩

▶ λ calculus translation?
  (λx| (λx| ⟨BODY⟩) ⟨E⟩) ⟨F⟩

LET x =⟨E⟩
  LET x = ⟨F⟩ AND y = x IN ⟨BODY⟩

▶ λ calculus translation?
  (λx| (λxy| ⟨BODY⟩) ⟨F⟩ x) ⟨E⟩
Special Forms: LET Semantics II

\[
\text{LET } x = \langle E \rangle \text{ IN LET } x = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle
\]

▶ \(\lambda\) calculus translation?
\[
(\lambda x | (\lambda x | \langle \text{BODY} \rangle) \langle E \rangle) \langle F \rangle
\]

LET \(x = \langle E \rangle\)
LET \(x = \langle F \rangle\) AND \(y = x\) IN \(\langle \text{BODY} \rangle\)

▶ \(\lambda\) calculus translation?
\[
(\lambda x | (\lambda xy | \langle \text{BODY} \rangle) \langle F \rangle \langle x \rangle) \langle E \rangle
\]

LET \(f(x) = \langle E \rangle\) IN \(\langle \text{BODY} \rangle\)
Special Forms: LET Semantics II

LET \( x = \langle E \rangle \) IN LET \( x = \langle F \rangle \) IN \( \langle \text{BODY} \rangle \)

- \( \lambda \) calculus translation?
  \( (\lambda x| (\lambda x| \langle \text{BODY} \rangle) \langle E \rangle) \langle F \rangle \)

LET \( x = \langle E \rangle \)
LET \( x = \langle F \rangle \) AND \( y = x \) IN \( \langle \text{BODY} \rangle \)

- \( \lambda \) calculus translation?
  \( (\lambda x| (\lambda xy| \langle \text{BODY} \rangle) \langle F \rangle x) \langle E \rangle \)

LET \( f(x) = \langle E \rangle \) IN \( \langle \text{BODY} \rangle \)

- \( \lambda \) calculus translation?
Special Forms: LET Semantics II

LET $x = \langle E \rangle$ IN LET $x = \langle F \rangle$ IN $\langle$BODY$\rangle$

- $\lambda$ calculus translation?
  
  $$(\lambda x | (\lambda x | \langle$BODY$\rangle) \langle E \rangle) \langle F \rangle$$

LET $x = \langle E \rangle$

LET $x = \langle F \rangle$ AND $y = x$ IN $\langle$BODY$\rangle$

- $\lambda$ calculus translation?
  
  $$(\lambda x | (\lambda xy | \langle$BODY$\rangle) \langle F \rangle x) \langle E \rangle$$

LET $f(x) = \langle E \rangle$ IN $\langle$BODY$\rangle$

- $\lambda$ calculus translation?

- Closer:
LET x = \langle E \rangle \text{ IN } LET x = \langle F \rangle \text{ IN } \langle \text{BODY} \rangle \\
\quad \Rightarrow \lambda \text{ calculus translation?} \\
\quad \quad (\lambda x \mid (\lambda x \mid \langle \text{BODY} \rangle) \langle E \rangle) \langle F \rangle \\
LET x = \langle E \rangle \\
\quad LET x = \langle F \rangle \text{ AND } y = x \text{ IN } \langle \text{BODY} \rangle \\
\quad \Rightarrow \lambda \text{ calculus translation?} \\
\quad \quad (\lambda x \mid (\lambda xy \mid \langle \text{BODY} \rangle) \langle F \rangle x) \langle E \rangle \\
LET f(x) = \langle E \rangle \text{ IN } \langle \text{BODY} \rangle \\
\quad \Rightarrow \lambda \text{ calculus translation?} \\
\quad \quad \Rightarrow \text{ Closer: LET } f = (\lambda x \mid \langle E \rangle) \text{ IN } \langle \text{BODY} \rangle
Special Forms: LET Semantics II

LET x = ⟨E⟩ IN LET x = ⟨F⟩ IN ⟨BODY⟩

▶ λ calculus translation?
(λx| (λx| ⟨BODY⟩) ⟨E⟩) ⟨F⟩

LET x = ⟨E⟩
LET x = ⟨F⟩ AND y = x IN ⟨BODY⟩

▶ λ calculus translation?
(λx| (λxy| ⟨BODY⟩) ⟨F⟩ x) ⟨E⟩

LET f(x) = ⟨E⟩ IN ⟨BODY⟩

▶ λ calculus translation?
▶ Closer: LET f = (λx| ⟨E⟩) IN ⟨BODY⟩
(λf| ⟨BODY⟩) (λx |⟨E⟩)
Special Forms: LET Semantics II

LET \( x = \langle E \rangle \) IN LET \( x = \langle F \rangle \) IN \( \langle \text{BODY} \rangle \)

- \( \lambda \) calculus translation?
  \( (\lambda x \mid (\lambda x \mid \langle \text{BODY} \rangle) \langle E \rangle) \langle F \rangle \)

LET \( x = \langle E \rangle \)
LET \( x = \langle F \rangle \) AND \( y = x \) IN \( \langle \text{BODY} \rangle \)

- \( \lambda \) calculus translation?
  \( (\lambda x \mid (\lambda xy \mid \langle \text{BODY} \rangle) \langle F \rangle x) \langle E \rangle \)

LET \( f(x) = \langle E \rangle \) IN \( \langle \text{BODY} \rangle \)

- \( \lambda \) calculus translation?
  - Closer: LET \( f = (\lambda x \mid \langle E \rangle) \) IN \( \langle \text{BODY} \rangle \)
    \( (\lambda f \mid \langle \text{BODY} \rangle) (\lambda x \mid \langle E \rangle) \)

- \( \lambda \)-calculus gives precise meaning to each case of LET
Special Forms: LET and Self-reference

LET \( f(n) = \)
\[
\text{IF zerop(n) THEN 1 ELSE n*f(n-1)}
\]
IN \langle\text{BODY}\rangle
Special Forms: LET and Self-reference

LET f(n) =
    IF zerop(n) THEN 1 ELSE n*f(n-1)
IN ⟨BODY⟩

▶ λ calculus translation?

▶ What does the recursive call to f point to? It is a free variable!
▶ Is this correct? Yes.

Otherwise LET x=2 IN LET x=2*x IN ⟨BODY⟩ would fail:

(λ x | (λ x | ⟨BODY⟩) 2*x) 2
Special Forms: LET and Self-reference

LET f(n) =
    IF zerop(n) THEN 1 ELSE n*f(n-1)
IN ⟨BODY⟩

▶ λ calculus translation? Approximately:
(λf | ⟨BODY⟩)
( (λxyz|xyz) zerop(n) 1 n*f(n-1) )

▶ What does the recursive call to f point to? It is a free variable!
▶ Is this correct? Yes.

Otherwise LET x=2 IN LET x=2*x IN ⟨BODY⟩ would fail:
(λx| (λx| ⟨BODY⟩) 2*x) 2
Special Forms: LET and Self-reference

LET \( f(n) = \)

\[
\begin{align*}
\text{IF zero}(p(n)) & \text{ THEN } 1 & \text{ ELSE } n \cdot f(n-1)
\end{align*}
\]

IN ⟨BODY⟩

▶ \( \lambda \) calculus translation? Approximately:

\[
(\lambda f \mid \langle \text{BODY} \rangle)
\]

\[
( (\lambda xyz | xyz) \text{zero}(p(n)) 1 n \cdot f(n-1) )
\]

\[
(\lambda f \mid \langle \text{BODY} \rangle)
\]

\[
( (\lambda xyz | xyz) \text{zero}(p(n)) 1 n \cdot f(n-1) )
\]

▶ What does the recursive call to \( f \) point to? It is a free variable!
Special Forms: LET and Self-reference

LET f(n) =
    IF zerop(n) THEN 1 ELSE n*f(n-1)
IN ⟨BODY⟩

▶ λ calculus translation?  Approximately:
(λf | ⟨BODY⟩)
  ( (λxyz|xyz) zerop(n) 1 n*f(n-1) )

(λf | ⟨BODY⟩)
  ( (λxyz|xyz) zerop(n) 1 n*f(n-1) )

▶ What does the recursive call to f point to? It is a free variable!

▶ Is this correct?
Special Forms: LET and Self-reference

LET $f(n) =$
  IF zerop(n) THEN 1 ELSE n*f(n-1)
IN ⟨BODY⟩

▶ $\lambda$ calculus translation? Approximately:
$(\lambda f \mid ⟨BODY⟩) ($ $(\lambda xyz|xyz) \text{zerop}(n) 1 n*f(n-1)$ )

$(\lambda f \mid ⟨BODY⟩) ($ $(\lambda xyz|xyz) \text{zerop}(n) 1 n*f(n-1)$ )$

▶ What does the recursive call to $f$ point to? It is a free variable!

▶ Is this correct? Yes.

Otherwise LET $x=2$ IN LET $x=2*x$ IN ⟨BODY⟩ would fail:
$(\lambda x\mid (\lambda x⟨BODY⟩)) 2*x) 2$
Special Forms: LETREC

LETREC  \( f(n) = \)

 IF \( \text{zerop}(n) \)  
 THEN 1  
 ELSE \( n*f(n-1) \)  
 IN \( \langle \text{BODY} \rangle \)
Special Forms: LETREC

LETREC \( f(n) = \)
\[
\begin{align*}
\text{IF } &\text{ zerop}(n) \\
\text{THEN } &1 \\
\text{ELSE } &n*f(n-1)
\end{align*}
\]
IN \( \langle \text{BODY} \rangle \)

- Sometimes we want vars in definition to refer to their labels

Dr. B. Price and Dr. G. Greiner

CMPUT 325: Abstract Programming
Special Forms: LETREC

LETREC \( f(n) = \)
\[
\text{IF zerop(n) THEN 1 ELSE } n \times f(n-1) \]
IN \( \langle \text{BODY} \rangle \)

- Sometimes we want vars in definition to refer to their labels
- Different semantics than LET — needs different name
Special Forms: LETREC

\[
\text{LETREC } f(n) = \\
\text{ IF zerop(n) }
\text{ THEN 1 }
\text{ ELSE } n*f(n-1) \\
\text{ IN } \langle \text{BODY} \rangle
\]

- Sometimes we want vars in definition to refer to their labels
- Different semantics than LET — needs different name
- \(\lambda\)-calculus translation?
Special Forms: LETREC

\[
\text{LETREC } f(n) = \\
\quad \text{IF zerop}(n) \\
\quad \quad \text{THEN } 1 \\
\quad \quad \text{ELSE } n* f(n-1) \\
\text{IN } \langle \text{BODY} \rangle
\]

- Sometimes we want vars in definition to refer to their labels
- Different semantics than LET — needs different name
- \(\lambda\)-calculus translation? Use combinator operator \(Y\)

\[
(\lambda f|\langle \text{BODY} \rangle \rangle \ (Y \ (\lambda f| \ (\lambda n| \ \text{zerop}(n) \ 1 \ n* f(n-1)) ))
\]
Special Forms: LETREC

LETREC \( f(n) = \)

\[ \begin{align*}
\text{IF} & \ \text{zerop}(n) \\
\text{THEN} & \ 1 \\
\text{ELSE} & \ n*f(n-1)
\end{align*} \]

\( \text{IN} \langle \text{BODY} \rangle \)

- Sometimes we want vars in definition to refer to their labels
- Different semantics than LET — needs different name
- \( \lambda \)-calculus translation?

Use combinator operator \( Y \)

\[
(\lambda f | \langle \text{BODY} \rangle) (Y (\lambda f | (\lambda n | \text{zerop}(n) \ 1 \ n*f(n-1)) )) \\
(\lambda f | \langle \text{BODY} \rangle) (Y (\lambda f | (\lambda n | \text{zerop}(n) \ 1 \ n*f(n-1)) ))
\]

\( \uparrow \)
Special Forms: LETREC

LETREC \( f(n) = \)
    IF zerop(n)
        THEN 1
    ELSE n*f(n-1)

IN ⟨BODY⟩

▶ Sometimes we want vars in definition to refer to their labels
▶ Different semantics than LET — needs different name
▶ \( \lambda \)-calculus translation? Use combinator operator \( Y \)

\[
(\lambda f. \langle \text{BODY} \rangle) \left( Y (\lambda f. (\lambda n. \text{zerop}(n) \cdot 1 \cdot n \cdot f(n-1))) \right)
\]

▶ Are 2 \( f \)'s the same?
Special Forms: LETREC

\[
\text{LETREC } f(n) = \\
\begin{array}{l}
\text{IF } \text{zerop}(n) \\
\text{THEN 1 }\\
\text{ELSE } n*f(n-1)
\end{array}
\]
\text{IN } \langle \text{BODY} \rangle

- Sometimes we want vars in definition to refer to their labels
- Different semantics than LET — needs different name
- \(\lambda\)-calculus translation? Use combinator operator \(Y\)

\[
(\lambda f \mid \langle \text{BODY} \rangle) (Y (\lambda f \mid (\lambda n \mid \text{zerop}(n) 1 n*f(n-1))))
\]
\[
(\lambda f \mid \langle \text{BODY} \rangle) (Y (\lambda f \mid (\lambda n \mid \text{zerop}(n) 1 n*f(n-1))))
\]

- Are 2 \(f\)'s the same? No. \(f\) in function def is not free!
Special Forms: Nested LETREC

LETREC
\[
f(n) = \text{IF } \text{zerop}(n) \text{ THEN } 1 \text{ ELSE } n*f(n-1) \text{ IN}
\]
LETREC
\[
g(n) = \text{IF } \text{zerop}(n) \text{ THEN } 0 \text{ ELSE } f(n)+g(n-1) \text{ IN}
\]
LETREC
  f(n) = IF zerop(n) THEN 1 ELSE n*f(n-1) IN
LETREC
  g(n) = IF zerop(n) THEN 0 ELSE f(n)+g(n-1) IN
⟨BODY⟩

What does this do?
Special Forms: Nested LETREC

LETREC
\[ f(n) = \text{IF } \text{zerop}(n) \text{ THEN } 1 \text{ ELSE } n*f(n-1) \text{ IN} \]
LETREC
\[ g(n) = \text{IF } \text{zerop}(n) \text{ THEN } 0 \text{ ELSE } f(n)+g(n-1) \text{ IN} \]
⟨BODY⟩

▶ What does this do? Sums first n factorials.
Special Forms: Nested LETREC

LETREC
  f(n) = IF zerop(n) THEN 1 ELSE n*f(n-1) IN
LETREC
  g(n) = IF zerop(n) THEN 0 ELSE f(n)+g(n-1) IN
⟨BODY⟩

▷ What does this do? Sums first n factorials. Translation?
Special Forms: Nested LETREC

\begin{verbatim}
LETREC
  f(n) = IF zerop(n) THEN 1 ELSE n*f(n-1) IN
LETREC
  g(n) = IF zerop(n) THEN 0 ELSE f(n)+g(n-1) IN
\end{verbatim}

What does this do? Sums first n factorials. Translation?

\begin{verbatim}
(\lambda f | \\
  (\lambda g | \\
    \langle BODY \rangle \\
      ) (Y (\lambda g | (\lambda n | zerop(n) 0 f(n)+g(n-1)) )) \\
  ) (Y (\lambda f | (\lambda n | zerop(n) 1 n*f(n-1)) ))
\end{verbatim}
Special Forms: Nested LETREC

LETREC
  f(n) = IF zerop(n) THEN 1 ELSE n*f(n-1) IN
LETREC
  g(n) = IF zerop(n) THEN 0 ELSE f(n)+g(n-1) IN

What does this do? Sums first n factorials. Translation?

(\lambda f | 
  (\lambda g | 
    (\lambda g | (BODY) 
      (Y (\lambda g | (\lambda n | zerop(n) 0 f(n)+g(n-1)) )))
    ) (Y (\lambda f | (\lambda n | zerop(n) 1 n*f(n-1)) )))

What does each f in this definition refer to?
Special Forms: Nested LETREC

```
LETREC
  f(n) = IF zerop(n) THEN 1 ELSE n*f(n-1) IN

LETREC
  g(n) = IF zerop(n) THEN 0 ELSE f(n)+g(n-1) IN

⟨BODY⟩
```

► What does this do? Sums first $n$ factorials. Translation?

$$(\lambda f \mid
  (\lambda g \mid
    ⟨BODY⟩
    ) (Y (\lambda g \mid (\lambda n \mid \text{zerop}(n) 0 f(n)+g(n-1))) ))
  ) (Y (\lambda f \mid (\lambda n \mid \text{zerop}(n) 1 n*f(n-1))) ))$$

► What does each $f$ in this definition refer to?

► Functions can refer to themselves and to earlier definitions
Special Forms: Parallel LETREC

LETREC
  even(n) IF zerop n THEN T ELSE odd(n-1) AND
  odd(n) IF zerop n THEN F ELSE even(n-1) IN
⟨BODY⟩
Special Forms: Parallel LETREC

LETREC
  even(n) IF zerop n THEN T ELSE odd( n-1 )
  AND odd(n) IF zerop n THEN F ELSE even( n-1 ) IN
  ⟨BODY⟩

In mutually recursive functions, earlier functions also refer to later functions
Special Forms: Parallel LETREC

LETREC
    even(n) IF zerop n THEN T ELSE odd(n-1) AND
    odd(n) IF zerop n THEN F ELSE even(n-1) IN
⟨BODY⟩

► In mutually recursive functions, earlier functions also refer to later functions

► Translation? Need pair of combinators that generate either function
Special Forms: Parallel LETREC

LETREC
   even(n) IF zerop n THEN T ELSE odd(n-1) AND
   odd(n) IF zerop n THEN F ELSE even(n-1) IN
⟨BODY⟩

▶ In mutually recursive functions, earlier functions also refer to later functions

▶ Translation? Need pair of combinators that generate either function

\[ Y_1 = (\lambda fg|RRS) \quad Y_2 = (\lambda fg|SRS) \]
Where \( R = (\lambda rs|f(rrs)(srs)) \), \( S = (\lambda rs|g(rrs)(srs)) \)
Special Forms: Parallel LETREC

LETREC
  even(n) IF zerop n THEN T ELSE odd(n-1) AND
  odd(n) IF zerop n THEN F ELSE even(n-1) IN
⟨BODY⟩

▶ In mutually recursive functions, earlier functions also refer to later functions

▶ Translation? Need pair of combinators that generate either function

Y1=\(\lambda fg|RRS\) Y2=\(\lambda fg|SRS\)
Where \(R=(\lambda rs|f(rrs)(srs))\), \(S=(\lambda rs|g(rrs)(srs))\)

▶ Combinator Properties

\[ Y1 \ F \ G = F (Y1 \ F \ G) (Y2 \ F \ G) \]
Special Forms: Parallel LETREC

LETREC

\begin{align*}
even(n) & \text{ IF } \text{zerop } n \text{ THEN } T \text{ ELSE } odd(n-1) \\
\text{AND } odd(n) & \text{ IF } \text{zerop } n \text{ THEN } F \text{ ELSE } even(n-1) \text{ IN}
\end{align*}

▶ In mutually recursive functions, earlier functions also refer to later functions

▶ Translation? Need pair of combinators that generate either function

\[ Y_1 = (\lambda fg| RRS) \quad Y_2 = (\lambda fg| SRS) \]
\[ \text{Where } R = (\lambda rs| f(rrs)(srs)) , \quad S = (\lambda rs| g(rrs)(srs)) \]

▶ Combinator Properties

\[ Y_1 \ F \ G \ = \ F \ (Y_1 \ F \ G) \ (Y_2 \ F \ G) \]
\[ Y_2 \ F \ G \ = \ G \ (Y_1 \ F \ G) \ (Y_2 \ F \ G) \]
Special Forms: Parallel LETREC

▶ Given the following definitions for F and G

\[
F \equiv \text{even}(n) \text{ IF zero? } n \text{ THEN } T \text{ ELSE odd( } n-1 \text{ )}
\]
\[
G \equiv \text{odd}(n) \text{ IF zero? } n \text{ THEN } F \text{ ELSE even( } n-1 \text{ )}
\]

▶ Why can't I use 2 independent combinators?

▶ Each copy of the function F has to also be able to reference G.
Given the following definitions for F and G

\[ \text{F} \equiv \text{even}(n) \ \text{IF} \ \text{zerop} \ n \ \text{THEN} \ T \ \text{ELSE} \ \text{odd}(n-1) \]
\[ \text{G} \equiv \text{odd}(n) \ \text{IF} \ \text{zerop} \ n \ \text{THEN} \ F \ \text{ELSE} \ \text{even}(n-1) \]

LETREC Expansion using pair of combinators

\[ (\lambda fg \langle \text{BODY} \rangle) \]
\[ (Y1 (\lambda fg \mid F)(\lambda fg \mid G)) \]
\[ (Y2 (\lambda fg \mid F)(\lambda fg \mid G)) \]
Given the following definitions for \( F \) and \( G \)

\[
F \equiv \text{even}(n) \ \text{IF} \ \text{zerop} \ n \ \text{THEN} \ T \ \text{ELSE} \ \text{odd}(n-1)
\]

\[
G \equiv \text{odd}(n) \ \text{IF} \ \text{zerop} \ n \ \text{THEN} \ F \ \text{ELSE} \ \text{even}(n-1)
\]

LETREC Expansion using pair of combinators

\[
(\lambda f g | \langle \text{BODY} \rangle) \\
(Y_1 (\lambda f g | F)(\lambda f g | G)) \\
(Y_2 (\lambda f g | F)(\lambda f g | G))
\]

Why can’t I use 2 independent combinators?
Given the following definitions for F and G

\[ F \equiv \text{even}(n) \text{ IF zerop } n \text{ THEN } T \text{ ELSE } \text{odd}(n-1) \]
\[ G \equiv \text{odd}(n) \text{ IF zerop } n \text{ THEN } F \text{ ELSE } \text{even}(n-1) \]

LETREC Expansion using pair of combinators

\[ (\lambda fg | \langle \text{BODY} \rangle) \]
\[ (Y1 (\lambda fg | F)(\lambda fg | G)) \]
\[ (Y2 (\lambda fg | F)(\lambda fg | G)) \]

Why can't I use 2 independent combinators?

Each copy of the function F has to also be able to reference G
Abstract Programming: BNF

\[
\langle \text{identifier} \rangle ::= \langle \text{alpha-char} \rangle \{ \langle \text{alpha-char} \rangle \mid \langle \text{number} \rangle \} \\
\langle \text{constant} \rangle ::= \langle \text{number} \rangle \mid \langle \text{boolean} \rangle \mid \langle \text{char-string} \rangle \\
\langle \text{expression} \rangle ::= \langle \text{constant} \rangle \mid \langle \text{identifier} \rangle \\
\quad \mid (\lambda \langle \text{identifier} \rangle \ "|" \langle \text{expression} \rangle ) \\
\quad \mid (\langle \text{expression} \rangle ^+ ) \\
\quad \mid \langle \text{identifier} \rangle (\langle \text{expression} \rangle \{ , \langle \text{expression} \rangle \} ^* ) \\
\quad \mid \text{let} \ \langle \text{definition} \rangle \ \text{in} \ \langle \text{expression} \rangle \\
\quad \mid \text{letrec} \ \langle \text{definition} \rangle \ \text{in} \ \langle \text{expression} \rangle \\
\quad \mid \text{if} \ \langle \text{expression} \rangle \ \text{then} \ \langle \text{expression} \rangle \ \text{else} \ \langle \text{expression} \rangle \\
\quad \mid \langle \text{arithmetic expression} \rangle \\
\langle \text{definition} \rangle ::= \langle \text{header} \rangle = \langle \text{expression} \rangle \\
\quad \mid \langle \text{definition} \rangle \ \{ \text{and} \ \langle \text{definition} \rangle \} ^* \\
\langle \text{header} \rangle ::= \langle \text{identifier} \rangle \\
\quad \mid \langle \text{identifier} \rangle \ (\langle \text{identifier} \rangle \ \{ , \ \langle \text{identifier} \rangle \} ^* ) \\
\langle \text{abstract-program} \rangle ::= \langle \text{expression} \rangle
\]
Convenience: WHERE and WHEREREC

- Sometimes convenient to put definitions after usage

\( \langle \text{BODY} \rangle \) WHERE \( \langle \text{DEFINITION} \rangle \)
Convenience: WHERE and WHEREREC

- Sometimes convenient to put definitions after usage
  
  \[\langle \text{BODY} \rangle \text{WHERE} \langle \text{DEFINITION} \rangle\]

- Example

  ```
  LET a(r) = pi * r IN
  a(10)
  WHERE pi = 3.1415
  ```

- Do we need brackets? No.

- LET's BODY is a single term

- Abstract Programming is Left-associative

- WHERE and WHEREREC do not add expressive power, just convenience
Convenience: WHERE and WHEREREC

- Sometimes convenient to put definitions after usage
  \( \langle \text{BODY} \rangle \text{ WHERE } \langle \text{DEFINITION} \rangle \)

- Example
  
  ```
  LET a(r) = pi * r \text{ IN}
  a(10)
  WHERE pi = 3.1415
  ```

- Do we need brackets? No.
  - LET’s \( \langle \text{BODY} \rangle \) is a single term
  - Abstract Programming is Left-associative
Convenience: WHERE and WHEREREC

- Sometimes convenient to put definitions after usage
  \[ \langle \text{BODY} \rangle \text{ WHERE } \langle \text{DEFINITION} \rangle \]
- Example
  
  \[
  \text{LET } a(r) = \pi \times r \text{ IN }
  \
  a(10)
  \]
  \[ \text{WHERE } \pi = 3.1415 \]
- Do we need brackets? No.
  - LET’s \langle \text{BODY} \rangle is a single term
  - Abstract Programming is Left-associative
- WHEREREC is analogous to LETREC
Convenience: WHERE and WHERERECC

- Sometimes convenient to put definitions after usage
  \(\langle \text{BODY} \rangle \text{ WHERE } \langle \text{DEFINITION} \rangle\)

- Example
  
  \[
  \text{LET } a(r) = \pi \times r \text{ IN } \\
  a(10) \\
  \text{WHERE } \pi = 3.1415
  \]

- Do we need brackets? No.
  
  - LET’s \(\langle \text{BODY} \rangle\) is a single term
  - Abstract Programming is Left-associative

- WHERERECC is analogous to LETREC

- WHERE and WHERERECC do not add expressive power, just convenience
Performance Considerations

- LET and LETREC mean different things
  
  \[
  \text{LET } x = x+2 \ \text{IN} \ \langle \text{BODY} \rangle \not\equiv \text{LETREC } x = x+2 \ \text{IN} \ \langle \text{BODY} \rangle
  \]
Performance Considerations

- LET and LETREC mean different things
  \[ \text{LET } x = x + 2 \text{ IN } \langle \text{BODY} \rangle \not\equiv \text{LETREC } x = x + 2 \text{ IN } \langle \text{BODY} \rangle \]

- Meaning overlaps when there is no self-reference
  \[ \text{LET } x = 2 \text{ IN } \langle \text{BODY} \rangle \equiv \text{LETREC } x = 2 \text{ IN } \langle \text{BODY} \rangle \]
Performance Considerations

- LET and LETREC mean different things
  \[ \text{LET } x = x + 2 \text{ IN } \langle \text{BODY} \rangle \neq \text{LETREC } x = x + 2 \text{ IN } \langle \text{BODY} \rangle \]

- Meaning overlaps when there is no self-reference
  \[ \text{LET } x = 2 \text{ IN } \langle \text{BODY} \rangle \equiv \text{LETREC } x = 2 \text{ IN } \langle \text{BODY} \rangle \]
  \[ \text{LET } y = 2 \text{ IN } \langle \text{BODY} \rangle \equiv \text{LETREC } y = 2 \text{ IN } \langle \text{BODY} \rangle \]
  \[ \text{LET } x = y \text{ IN } \langle \text{BODY} \rangle \equiv \text{LETREC } x = y \text{ IN } \langle \text{BODY} \rangle \]
Performance Considerations

- LET and LETREC mean different things

\[
\text{LET } x=x+2 \text{ IN } \langle \text{BODY} \rangle \neq \text{LETREC } x=x+2 \text{ IN } \langle \text{BODY} \rangle
\]

- Meaning overlaps when there is no self-reference

\[
\text{LET } x=2 \text{ IN } \langle \text{BODY} \rangle \equiv \text{LETREC } x=2 \text{ IN } \langle \text{BODY} \rangle
\]

\[
\text{LET } y=2 \text{ IN } \langle \text{BODY} \rangle \equiv \text{LETREC } y=2 \text{ IN } \langle \text{BODY} \rangle
\]

\[
\text{LET } x=y \text{ IN } \langle \text{BODY} \rangle \equiv \text{LETREC } x=y \text{ IN } \langle \text{BODY} \rangle
\]

- Depending on compiler, may be more efficient to use LET when possible
Higher-order Functions

- Abstract language looks like traditional languages

LET map(f,L) = IF null(L) THEN nil ELSE cons(f(car(L)), map(cdr(L)) ) IN
LET square(x)=x*x IN
map(square, [1 2 3 4])
→ [1 4 9 16]
Higher-order Functions

- Abstract language looks like traditional languages
- Underlying semantics does not distinguish data and functions
Higher-order Functions

- Abstract language looks like traditional languages
- Underlying semantics does not distinguish data and functions
- Higher-order function has at least one of these properties
  - Accepts a function as an argument
  - Returns a function as its value

```latex
LET map(f,L) =
  IF null(L) THEN nil
  ELSE cons(f(car(L)), map(cdr(L)))
IN
LET square(x) = x*x IN
map(square, [1 2 3 4])
→ [1 4 9 16]
```
Higher-order Functions

- Abstract language looks like traditional languages
- Underlying semantics does not distinguish data and functions
- Higher-order function has at least one of these properties
  - Accepts a function as an argument
  - Returns a function as its value
- Can treat functions as arguments or return values

```
LET map(f,L) =
  IF null(L)
  THEN nil
  ELSE cons(f(car(L)), map(cdr(L)))
IN

LET square(x) = x*x IN
map(square, [1 2 3 4])
→ [1 4 9 16]
```
Higher-order Functions

- Abstract language looks like traditional languages
- Underlying semantics does not distinguish data and functions
- Higher-order function has at least one of these properties
  - Accepts a function as an argument
  - Returns a function as its value
- Can treat functions as arguments or return values

```plaintext
LET map(f,L) =
  IF null(L)
  THEN nil
  ELSE cons( f(car(L)) , map(cdr(L)) )
IN

LET square(x)=x*x

LET map(square, [1 2 3 4])
```
Other Traditional Higher-order Functions

- **Filter**: apply a predicate to each item and return those items that satisfy

  \[(\text{filter 'even [1 2 3 4]}) \rightarrow [2 4]\]
Other Traditional Higher-order Functions

- **Filter:** apply a predicate to each item and return those items that satisfy
  
  \[
  \text{filter 'even [1 2 3 4]} \rightarrow [2 4]
  \]

- **Reduce:** combine elements of list with given function left associatively
  (common Lisp: reduce)

  \[
  (\text{reduce #'- '(1 2 3 4)}) \equiv (-4 -4) \equiv -8
  \]
Other Traditional Higher-order Functions

- Filter: apply a predicate to each item and return those items that satisfy
  \[(\text{filter 'even } [1 2 3 4])\rightarrow [2 4]\]

- Reduce: combine elements of list with given function left associatively
  (common Lisp: reduce)
  \[(\text{reduce #'- '}(1 2 3 4))\]
  \[\equiv (((1 - 2) - 3) - 4)\]
Other Traditional Higher-order Functions

- **Filter**: apply a predicate to each item and return those items that satisfy
  \[
  \text{(filter 'even [1 2 3 4])} \rightarrow [2 4]
  \]

- **Reduce**: combine elements of list with given function left associatively
  \[
  \text{(common Lisp: reduce)}
  \]
  \[
  \text{(reduce #'- '(1 2 3 4))}
  \equiv (((1 - 2) - 3) - 4)
  \equiv ((-1 - 3) - 4)
  \]
Other Traditional Higher-order Functions

► Filter: apply a predicate to each item and return those items that satisfy

\[(\text{filter 'even} \ [1 \ 2 \ 3 \ 4]) \rightarrow [2 \ 4]\]

► Reduce: combine elements of list with given function left associatively
(common Lisp: reduce)

\[(\text{reduce} \ #'- \ '(1 \ 2 \ 3 \ 4)) = ((1 - 2) - 3) - 4 = \ldots = -8\]
Other Traditional Higher-order Functions

- **Filter**: apply a predicate to each item and return those items that satisfy
  
  
  (filter 'even [1 2 3 4]) → [2 4]

- **Reduce**: combine elements of list with given function left associatively
  
  (common Lisp: reduce)

  (reduce #'- '(1 2 3 4))
  ≡ (((1 - 2) - 3) - 4)
  ≡ ((-1 - 3) - 4)
  ≡ (-4 -4)
  ≡ -8
Global Definitions

- In principle, there aren’t any: no DEFUN or SETF
Global Definitions

- In principle, there aren’t any: no DEFUN or SETF
- There are only nested LET statements
Global Definitions

- In principle, there aren’t any: no DEFUN or SETF
- There are only nested LET statements
- In principle, integers and primitives defined by LET

```lambda
LET T = (λxy|x)
AND F = (λxy|y)
AND + = ...
:
IN ⟨BODY⟩
```
Abstract Programming

- Can be used to implement any functional language
Abstract Programming

- Can be used to implement any functional language
- Is equivalent in power to a Turing machine
Abstract Programming

- Can be used to implement any functional language
- Is equivalent in power to a Turing machine
- Abstract programming language approximately equivalent to Pure Lisp
  - Parallel LET \( \approx \) Lisp LET
  - Nested LET’s \( \approx \) Lisp LET*
  - Parallel LETREC’s \( \approx \) Lisp LABELS
Partial application / Currying

- In principle, λ’s can be used anywhere in abstract programming

\[
\text{map}\left((\lambda x \mid 2+x), \ [1\ 2\ 3]\right) \rightarrow \ [3\ 4\ 5]
\]
Partial application / Currying

- In principle, λ’s can be used anywhere in abstract programming
  \[
  \text{map}( (\lambda x . 2 + x), [1 \ 2 \ 3] ) \rightarrow [3 \ 4 \ 5]
  \]

- A more elegant method:
Partial application / Currying

- In principle, λ’s can be used anywhere in abstract programming

\[
\text{map}( (\lambda x | 2+x), [1 2 3] ) \rightarrow [3 4 5]
\]

- A more elegant method:

- Let \( \text{pa} \) be the partial application operator

\[
\text{LET } \text{pa} = (\lambda f \ x | (\lambda y | f \ x \ y)) \text{ IN } \langle \text{BODY} \rangle
\]
In principle, \( \lambda \)'s can be used anywhere in abstract programming

\[
\text{map}( \lambda x \mid 2 + x), [1 \ 2 \ 3] \to [3 \ 4 \ 5]
\]

A more elegant method:

Let \( pa \) be the partial application operator

\[
\text{LET } pa = (\lambda f \ x \mid (\lambda y \mid f \ x \ y)) \text{ IN } \langle \text{BODY} \rangle
\]

Allows us to write:

\[
\text{LET } inc = pa \ ' + 1 \text{ IN } \ ;; \text{ i.e. } inc = (\lambda y \mid (+ \ 1 \ y)) \\
inc(1) \to 2
\]
Partial application / Currying

- In principle, λ’s can be used anywhere in abstract programming

  \[
  \text{map}( (\lambda x | 2+x) , \ [1 \ 2 \ 3] ) \rightarrow [3 \ 4 \ 5]
  \]

- A more elegant method:

  - Let \( \text{pa} \) be the partial application operator

    \[
    \text{LET } \text{pa} = (\lambda f \ x | (\lambda y | f \ x \ y)) \ \text{IN } \langle \text{BODY} \rangle
    \]

  - Allows us to write:

    \[
    \text{LET inc} = \text{pa }' + \ 1 \ \text{IN } ; ; \ i.e. \ inc = (\lambda y | (+ \ 1 \ y))
    \]

    \[
    \text{inc}(1) \rightarrow 2
    \]

- Or more impressively:

  \[
  \text{map}( \text{pa} \ + \ 2, \ [1 \ 2 \ 3] ) \rightarrow [3 \ 4 \ 5]
  \]
Partial application / Currying

- In principle, λ’s can be used anywhere in abstract programming

\[
\text{map}( (\lambda x | 2+x), [1 \ 2 \ 3] ) \to [3 \ 4 \ 5]
\]

- A more elegant method:

- Let \(pa\) be the partial application operator

\[
\text{LET } \text{pa} = (\lambda f \ x | (\lambda y | f \ x \ y)) \text{ IN } \langle \text{BODY} \rangle
\]

- Allows us to write:

\[
\text{LET } \text{inc} = \text{pa }' + 1 \text{ IN } ;; \text{ i.e. } \text{inc} = (\lambda y | (+ 1 \ y)) \\
\text{inc}(1) \to 2
\]

- Or more impressively:

\[
\text{map}( \text{pa } + 2, [1 \ 2 \ 3] ) \to [3 \ 4 \ 5]
\]

- Partial application \(\equiv\) currying
Combinators as a Calculus

- The central operation in $\lambda$-calculus is the $\beta$-substitution
Combinators as a Calculus

- The central operation in $\lambda$-calculus is the $\beta$-substitution
- It requires
  - scanning expressions for variables
  - analyzing free vs. bound variables
  - renaming when conflicts are discovered
  - rebuilding substituted copies of expressions repeatedly
  - $\lambda$-parameters just steer copies of expressions to places in code

"combinators" which move, copy and delete arguments
Combinators as a Calculus

- The central operation in $\lambda$-calculus is the $\beta$-substitution
- It requires
  - scanning expressions for variables

Dr. B. Price and Dr. G. Greiner
CMPUT 325: Abstract Programming
Combinators as a Calculus

- The central operation in $\lambda$-calculus is the $\beta$-substitution
- It requires
  - scanning expressions for variables
  - analyzing free vs. bound variables
Combinators as a Calculus

- The central operation in λ-calculus is the β-substitution
- It requires
  - scanning expressions for variables
  - analyzing free vs. bound variables
  - renaming when conflicts are discovered
Combinators as a Calculus

The central operation in $\lambda$-calculus is the $\beta$-substitution

It requires

- scanning expressions for variables
- analyzing free vs. bound variables
- renaming when conflicts are discovered
- rebuilding substituted copies of expressions repeatedly
Combinators as a Calculus

- The central operation in $\lambda$-calculus is the $\beta$-substitution

- It requires
  - scanning expressions for variables
  - analyzing free vs. bound variables
  - renaming when conflicts are discovered
  - rebuilding substituted copies of expressions repeatedly

- $\lambda$-parameters just “steer” copies of expressions to places in code
Combinators as a Calculus

- The central operation in λ-calculus is the β-substitution.

- It requires:
  - scanning expressions for variables
  - analyzing free vs. bound variables
  - renaming when conflicts are discovered
  - rebuilding substituted copies of expressions repeatedly

- λ-parameters just “steer” copies of expressions to places in code

- Define "combinators" which move, copy and delete arguments
Combinators as Special Functions

- Suppose we had a library of useful combinators: X, Y, Z

- Intuitive example: Program ≡ string of combinators: ZXY ZYY...

- Suppose 2 argument combinator Z reverses its arguments: ZXY ZYY... → YX ZYY...

- Suppose 1 argument combinator Y duplicates its arguments: YX ZYY... → XX ZYY...

- Suppose 1 argument combinator X deletes its second argument: XX ZYY... → X ZYY...

- Combinators can be defined using λ-calculus: X ≡ (λxy|x)

- Given combinators, no λ's, formal parameters or substitution required
Combinators as Special Functions

- Suppose we had a library of useful combinators: X, Y, Z
- Intuitive example:

  Program ≡ string of combinators: ZXY ZYY . . .

  Suppose 2 argument combinator Z reverses its arguments:
  ZXY ZYY . . . → YX ZYY . . .

  Suppose 1 argument combinator Y duplicates its arguments:
  YX ZYY . . . → XX ZYY . . .

  Suppose 1 argument combinator X deletes its second argument:
  XX ZYY . . . → X ZYY . . .

Combinators can be defined using λ-calculus: X ≡ (λxy|x)

Given combinators, no λ’s, formal parameters or substitution required.
Combinators as Special Functions

- Suppose we had a library of useful combinators: \(X,Y,Z\)

- Intuitive example:
  - Program \(\equiv\) string of combinators: \(ZXYZYY\ldots\)
Combinators as Special Functions

- Suppose we had a library of useful combinators: X, Y, Z

- Intuitive example:
  - Program ≡ string of combinators: ZXYZYY...
  - Suppose 2 argument combinator Z reverses its arguments
    ZXYZYY... → YXZYY...

- Combinators can be defined using λ-calculus: X ≡ (λxy|x)

- Given combinators, no λ’s, formal parameters or substitution required.
Suppose we had a library of useful combinators: X, Y, Z

Intuitive example:

- Program ≡ string of combinators: ZXYZYY...
- Suppose 2 argument combinator Z reverses its arguments
  ZXYZYY... → YXZYY...
- Suppose 1 argument combinator Y duplicates its arguments
  YXZYY... → XXZYY...
Combinators as Special Functions

- Suppose we had a library of useful combinators: X, Y, Z

- Intuitive example:
  - Program $\equiv$ string of combinators: $ZXYZYY\ldots$
  - Suppose 2 argument combinator Z reverses its arguments
    $ZXYZYY\ldots \rightarrow YXZYY\ldots$
  - Suppose 1 argument combinator Y duplicates its arguments
    $YXZYY\ldots \rightarrow XXZYY\ldots$
  - Suppose 1 argument combinator X deletes its second argument
    $XXZYY\ldots \rightarrow XZYY\ldots$
Combinators as Special Functions

- Suppose we had a library of useful combinators: X, Y, Z

- Intuitive example:
  - Program ≡ string of combinators: ZXYZYY...
  - Suppose 2 argument combinator Z reverses its arguments
    \[ ZXYZYY\ldots \rightarrow YXZYY\ldots \]
  - Suppose 1 argument combinator Y duplicates its arguments
    \[ XYZYY\ldots \rightarrow XXZYY\ldots \]
  - Suppose 1 argument combinator X deletes its second argument
    \[ XXZYY\ldots \rightarrow XZYY\ldots \]

- Combinators can be defined using \( \lambda \)-calculus: \( X \equiv (\lambda xy | x) \)
Combinators as Special Functions

- Suppose we had a library of useful combinators: \( X, Y, Z \)

- Intuitive example:
  - Program \( \equiv \) string of combinators: \( ZXYZYY\ldots \)
  - Suppose 2 argument combinator \( Z \) reverses its arguments
    \( ZXYZYY\ldots \rightarrow YXZYY\ldots \)
  - Suppose 1 argument combinator \( Y \) duplicates its arguments
    \( YXZYY\ldots \rightarrow XXZYY\ldots \)
  - Suppose 1 argument combinator \( X \) deletes its second argument
    \( XXZYY\ldots \rightarrow XZYY\ldots \)

- Combinators can be defined using \( \lambda \)-calculus: \( X \equiv (\lambda xy|x) \)

- Given combinators, no \( \lambda \)'s, formal parameters or substitution required
Combinators as a Calculus

- Left-associative like $\lambda$-calculus: $ABCD \ldots \equiv (((AB)C)D)$
Combinators as a Calculus

- Left-associative like $\lambda$-calculus: $ABCD \ldots \equiv (((AB)C)D)$

- Proved that two combinators can generate all others

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>A $\lambda$Calculus Def</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>distribute</td>
<td>$(\lambda xyz</td>
<td>xz(yz))$</td>
</tr>
<tr>
<td>k</td>
<td>constant</td>
<td>$(\lambda xy</td>
<td>x)$</td>
</tr>
</tbody>
</table>
Combinators as a Calculus

- Left-associative like $\lambda$-calculus: $ABCD \ldots \equiv (((AB)C)D)$

- Proved that two combinators can generate all others

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>$\lambda$Calculus Def</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s$</td>
<td>distribute</td>
<td>$(\lambda xyz</td>
<td>xz(yz) )$</td>
</tr>
<tr>
<td>$k$</td>
<td>constant</td>
<td>$(\lambda xy</td>
<td>x)$</td>
</tr>
</tbody>
</table>

- Identity function: $I \equiv S \ K \ K \ A \equiv K \ A \ (K \ A) \equiv A$
# Common Combinators

Common combinators can be defined using S and K

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Name</th>
<th>$\lambda$Calculus Def</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>compose</td>
<td>$(\lambda xyz</td>
<td>x(yz))$</td>
</tr>
<tr>
<td>C</td>
<td>reversal</td>
<td>$(\lambda xyz</td>
<td>xzy)$</td>
</tr>
<tr>
<td>W</td>
<td>duplicate</td>
<td>$(\lambda xy</td>
<td>xyy)$</td>
</tr>
</tbody>
</table>
Common Combinators

- There is a mechanical mapping between \(\lambda\)-calculus and the minimal SKI combinator language consisting of only S, K and I combinators
Common Combinators

- There is a mechanical mapping between $\lambda$-calculus and the minimal SKI combinator language consisting of only S, K and I combinators

- $\text{cons} \equiv \text{B C (C I)}$
Common Combinators

- There is a mechanical mapping between $\lambda$-calculus and the minimal SKI combinator language consisting of only S, K and I combinators
  - \( \text{cons} \equiv B \ C \ (C \ I) \)
  - \( \text{car} \equiv C \ I \ K \)
Common Combinators

- There is a mechanical mapping between $\lambda$-calculus and the minimal SKI combinator language consisting of only S,K and I combinators
- $\text{cons} \equiv B\ C\ (C\ I)$
- $\text{car} \equiv C\ I\ K$
- $\text{cdr} \equiv C\ I\ (K\ I)$
Common Combinators

- There is a mechanical mapping between $\lambda$-calculus and the minimal SKI combinator language consisting of only S, K and I combinators

- cons ≡ B C (C I)

- car ≡ C I K

- cdr ≡ C I (K I)

- integers: zero ≡ K I, $Z_i = (s (s \ldots (s z) \ldots))$ with $i$ copies of $s$
Common Combinators

- There is a mechanical mapping between $\lambda$-calculus and the minimal SKI combinator language consisting of only S, K and I combinators

- $\text{cons} \equiv B \ C \ (C \ I)$

- $\text{car} \equiv C \ I \ K$

- $\text{cdr} \equiv C \ I \ (K \ I)$

- integers: $\text{zero} \equiv K I$, $Z_i = (s \ (s \ldots(s \ z)\ldots))$ with $i$ copies of $s$

- $\text{successor}(n) \equiv (S \ B \ Z_i) \ s \ z$
Common Combinators

- There is a mechanical mapping between $\lambda$-calculus and the minimal SKI combinator language consisting of only S, K and I combinators
  
  - \text{cons} \equiv B\ C\ (C\ I)
  
  - \text{car} \equiv C\ I\ K
  
  - \text{cdr} \equiv C\ I\ (K\ I)
  
  - \text{integers: zero} \equiv Kl,\ Z_i = (s\ (s\ldots(s\ z)\ldots))\ with\ i\ copies\ of\ s
  
  - \text{successor(n)} \equiv (S\ B\ Z_i)\ s\ z
  
  - \text{Factorial: } f(n) = (if\ (=\ n\ 0)\ 1\ (\times n\ (f\ (-\ n\ 1))))
  \equiv (S(C(B\ \text{if}\ (C=0))\ 1)\ (S\times\ (B\ f\ (C\ -\ 1))))
Combinator Notes

- Common subsequences can be compiled into super-combinators
Combinator Notes

- Common subsequences can be compiled into super-combinators

- VLSI chips have been fabricated to directly implement combinator logic

Another example
- Divide every number in L by 2
  \[
  \text{MAP (/ SWAP 2) L}
  \]
- SWAP reverses arguments to / so we get each number divided by 2 instead of 2 divided by each number
Combinator Notes

▶ Common subsequences can be compiled into super-combinators

▶ VLSI chips have been fabricated to directly implement combinator logic

▶ Haskell and Miranda define many high-level combinators
Combinator Notes

- Common subsequences can be compiled into super-combinators
- VLSI chips have been fabricated to directly implement combinator logic
- Haskell and Miranda define many high-level combinators
- Another example

Divide every number in L by 2

MAP (/ SWAP 2) L

SWAP reverses arguments to / so we get

each number divided by 2

instead of 2 divided by each number
Combinator Notes

- Common subsequences can be compiled into super-combinators

- VLSI chips have been fabricated to directly implement combinator logic

- Haskell and Miranda define many high-level combinators

- Another example

- Divide every number in L by 2
  \[
  \text{MAP} \left( / \ SWAP \ 2 \right) L
  \]
Combinator Notes

- Common subsequences can be compiled into super-combinators

- VLSI chips have been fabricated to directly implement combinator logic

- Haskell and Miranda define many high-level combinators

- Another example

- Divide every number in $L$ by 2
  $$\text{MAP } (/ \text{ SWAP } 2 ) \ L$$

- SWAP reverses arguments to / so we get
  - each number divided by 2
  - instead of 2 divided by each number