

Lambda Calculus

- Very general mathematical language.
- Simple syntax, powerful semantics.
- Encapsulates function abstraction (definition) and application.
- Functions easily used as values.
- Used in denotational semantics.
- Functional programming languages can be viewed as syntactic variants.

Concrete Syntax for Lambda Calculus

$\langle \text{expression} \rangle ::=$
 $\langle \text{variable} \rangle$
 | $\langle \text{constant} \rangle$
 | $(\langle \text{expression}_1 \rangle \langle \text{expression}_2 \rangle)$
 | $(\lambda \langle \text{variable} \rangle . \langle \text{expression} \rangle)$

- Variables - lower case identifiers
- Constants - any predefined object
 - impure/applied lambda calculus
- Rule 3 - application of expr_1 to expr_2
 - operator (rator)
 - operand (rand)
- Rule 4 - lambda abstraction
 - bound variable
 - body

Application (Combinations)

$(E_1 E_2)$ means:
 apply the result of evaluating E_1 to E_2

- E_1 should be a function, either predefined (a constant) or an abstraction.
- In the pure lambda calculus, every (closed) expression *is* a function.

Examples

$(\text{succ } 1)$ where 'succ' is the successor function

$(\text{zerop } 0)$ where 'zerop' is the zero predicate (test if zero)

$(\text{add } 3 \ 5)$ where 'add' is the addition operation

Abstractions

- "anonymous" functions:

$(\lambda x . x)$

is similar to

$f(x) = x$

but has no name.

Examples

$(\lambda m . (\text{add } m \ 1)) \approx f(m) = m + 1$

$(\lambda m . (\lambda n . (\text{sub } n \ m))) \approx g(m, n) = n - m$

where \approx means "is similar to".

Notational Conventions

1. Uppercase: metavariables for expressions.

2. Function application associates to the left.

$$E_1 E_2 E_3 \text{ means } ((E_1 E_2) E_3)$$

3. The scope of “ λ <variable>” in an abstraction extends as far to the right as possible.

$$\lambda x . E_1 E_2 E_3 \text{ means } (\lambda x . (E_1 E_2 E_3))$$

Application has a higher precedence than abstraction.

Parentheses are needed for $(\lambda x . E_1 E_2) E_3$ where E_3 is an argument to the function $\lambda x . E_1 E_2$, not part of the body of the function.

4. An abstraction allows a list of variables that abbreviates a series of lambda abstractions.

$$\lambda x y z . E \text{ means } (\lambda x . (\lambda y . (\lambda z . E)))$$

5. Name functions using *define*.

Example

Given $\lambda x . x \lambda y . y x$, use the conventions above to parenthesize it.

1. $(\lambda x . x \lambda y . y x)$

2. $(\lambda x . x (\lambda y . y x))$

3. $(\lambda x . (x (\lambda y . (y x))))$

To be completely parenthesized, the expression should have parentheses around all applications, but we will normally stop at step 2.

Parenthesizing Examples

$$(\lambda x y z . x z (y z)) (\lambda x y . x) \lambda x y . x$$

$$\lambda x . (\lambda y . y y) z (\lambda w . w) x$$

Curried functions

Note differences between

$$(\lambda m . (\lambda n . (\text{sub } n \text{ } m)))$$

$$g(m, n) = n - m$$

- λ abstraction has no name
- λ abstraction is a function of 1 parameter

So

$$((\lambda m . (\lambda n . (\text{sub } n \text{ } m))) 1) = (\lambda n . (\text{sub } n \text{ } 1))$$

returns a function

whereas g is a function of 2 parameters

Using signatures (domain and codomain):

$$g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$(\lambda m . (\lambda n . (\text{sub } n \text{ } m))) : \mathbb{N} \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$$

where \mathbb{N} is the set of natural numbers.

The signature of g can be read:

g takes a tuple (pair) of natural numbers and returns a natural number.

The other signature can be read as

take a natural number, and return a *function* from natural numbers to natural numbers.

Computationally though, these functions are the same. If " \langle , \rangle " is the tuple constructor, define

$\text{Curry} = (\lambda f . \lambda x . \lambda y . f \langle x, y \rangle)$ and
 $\text{Uncurry} = (\lambda f . \lambda \langle x, y \rangle . f x y)$

Then

$\text{Curry } g = (\lambda x . (\lambda y . (\text{sub } y x)))$

$\text{Uncurry } (\lambda x . (\lambda y . (\text{sub } y x))) = g$

Note that Curry and Uncurry have no effect on what is being computed.

Semantics (operational)

- Defined in terms of syntactic manipulations on lambda expressions
- Intuitively, applications are like function calls: we need to
 - bind values to the formal parameters
 - evaluate the body of the function

Bound and Free Variables

An *occurrence* of a variable v in a lambda expression is called **bound** if it is within the scope of a " λv "; otherwise it is called **free**.

$((\lambda x . (y x)) x)$

The **set of free variables** in an expression E , denoted by $FV(E)$, is defined by:

- $FV(c) = \emptyset$ for any constant c
- $FV(x) = \{x\}$ for any variable x
- $FV(E_1 E_2) = FV(E_1) \cup FV(E_2)$
- $FV(\lambda x . E) = FV(E) - \{x\}$

$FV(E)$ is the set of variable that have free occurrences in E .

$FV((\lambda x . x) y) = \{y\}$

$FV((\lambda x . x) x) = \{x\}$

Note the definition deals with occurrences of variables.

Here one occurrence of x is bound and one occurrence is free.

Substitution

$E[v \rightarrow E_1]$ denotes the lambda expression obtained by replacing each *free* occurrence of the variable v in E by the lambda expression E_1 .

Such a substitution is call **valid** or **safe** if no free variable in E_1 becomes bound as a result of the substitution $E[v \rightarrow E_1]$.

An invalid substitution involves a **variable capture** or a **name clash**

An Invalid Substitution

$(\lambda x . (\text{sub } x y))[y \rightarrow x] = (\lambda x . (\text{sub } x x))$

First function: Subtract y from the argument.

Second function: Return zero.

$E[v \rightarrow E_1]$ is defined by:

- a) $v[v \rightarrow E_1] = E_1$ for any variable v
- b) $x[v \rightarrow E_1] = x$ for any variable $x \neq v$
- c) $c[v \rightarrow E_1] = c$ for any constant c
- d) $(E_{\text{rator}} E_{\text{rand}})[v \rightarrow E_1] = (E_{\text{rator}}[v \rightarrow E_1]) (E_{\text{rand}}[v \rightarrow E_1])$
- e) $(\lambda v . E)[v \rightarrow E_1] = (\lambda v . E)$
- f) $(\lambda x . E)[v \rightarrow E_1] = \lambda x . (E[v \rightarrow E_1])$
when $x \neq v$ and $x \notin FV(E_1)$
- g) $(\lambda x . E)[v \rightarrow E_1] = \lambda z . (E[x \rightarrow z][v \rightarrow E_1])$
when $x \neq v$ and $x \in FV(E_1)$,
where $z \neq v$ and $z \notin FV(E E_1)$

Substitution Examples

$(\lambda x . (\lambda y . y z) (\lambda w . w) z x)[z \rightarrow y]$

$(\lambda x . (\lambda y . y y) z (\lambda w . w) x)[z \rightarrow f x]$

Reduction Rules

α -reduction

If v and w are variables and E is a lambda expression,

$$\lambda v . E \Rightarrow_{\alpha} \lambda w . (E[v \rightarrow w])$$

provided that w does not occur at all in E , which makes the substitution $E[v \rightarrow w]$ safe.

$$\lambda x . f x \Rightarrow_{\alpha} \lambda y . f y$$

$$\lambda f . \lambda x . f (f x) \Rightarrow_{\alpha} \lambda g . \lambda x . g (g x)$$

β -reduction

If v is a variable and E and E_1 are lambda expressions,

$$(\lambda v . E) E_1 \Rightarrow_{\beta} E[v \rightarrow E_1]$$

provided the substitution $E[v \rightarrow E_1]$ is carried out according to the rules for a safe substitution.

η -reduction

If v is a variable, E is a lambda expression (denoting a function), and v has no free occurrence in E ,

$$\lambda v . (E v) \Rightarrow_{\eta} E.$$

In the pure lambda calculus every expression is a function.

The rule fails when E represents some predefined constants:

if 5 is a predefined numeral, $\lambda x . (5 x)$ and 5 are not equivalent or even related.

If E stands for a predefined function, the rule remains valid as shown by these examples:

$$\lambda m . (\text{sqr } m) \Rightarrow_{\eta} \text{sqr}$$

$$\lambda n . (\text{mul } 2 n) \Rightarrow_{\eta} (\text{mul } 2).$$

Take $E \Leftrightarrow_{\eta} F$ to mean $E \Rightarrow_{\eta} F$ or $F \Rightarrow_{\eta} E$.

Extensionality of functions:

If $f(x) = g(x)$ for all x , then $f = g$.

Extensionality Theorem If $F_1 x \Rightarrow^* E$ and $F_2 x \Rightarrow^* E$ where $x \notin FV(F_1 F_2)$, then $F_1 \Leftrightarrow^* F_2$ where \Leftrightarrow^* includes η -reductions.

Proof: $F_1 \Leftrightarrow_{\eta} \lambda x . (F_1 x) \Leftrightarrow^* \lambda x . E$
 $\Leftrightarrow^* \lambda x . (F_2 x) \Leftrightarrow_{\eta} F_2$.

δ -reduction

In an applied lambda calculus (not pure), rules associated with predefined values and functions are called δ rules.

$(\text{succ } 13) \Rightarrow_{\delta} 14$

$(\text{mul } 4 \ 7) \Rightarrow_{\delta} 28$

$(\text{not false}) \Rightarrow_{\delta} \text{true}$.

Reduction Example

Thrice $(\lambda m . (\text{mul } 2 \ m)) \ 7$

where Thrice = $\lambda f . \lambda x . f \ (f \ (f \ x))$

Thrice $(\lambda m . (\text{mul } 2 \ m)) \ 7$

$\Rightarrow (\lambda f . \lambda x . f \ (f \ (f \ x))) \ (\lambda m . (\text{mul } 2 \ m)) \ 7$

$\Rightarrow_{\beta} (\lambda x . (\lambda m . (\text{mul } 2 \ m)))$

$((\lambda m . (\text{mul } 2 \ m)) \ ((\lambda m . (\text{mul } 2 \ m)) \ x))) \ 7$

$\Rightarrow_{\beta} (\lambda m . (\text{mul } 2 \ m))$

$((\lambda m . (\text{mul } 2 \ m)) \ ((\lambda m . (\text{mul } 2 \ m)) \ 7))$

$\Rightarrow_{\beta} \text{mul } 2 \ ((\lambda m . (\text{mul } 2 \ m)) \ ((\lambda m . (\text{mul } 2 \ m)) \ 7))$

$\Rightarrow_{\beta} \text{mul } 2 \ (\text{mul } 2 \ ((\lambda m . (\text{mul } 2 \ m)) \ 7))$

$\Rightarrow_{\beta} \text{mul } 2 \ (\text{mul } 2 \ (\text{mul } 2 \ 7)) \Rightarrow_{\delta} \text{mul } 2 \ (\text{mul } 2 \ 14)$

$\Rightarrow_{\delta} \text{mul } 2 \ 28 \Rightarrow_{\delta} 56$

Normal Form

A lambda expression that contains no β -redexes and no δ -redexes.

It cannot be further reduced using the β -rule or the δ -rule (no function applications to evaluate).

Questions

1. Can every lambda expression be reduced to a normal form?
2. Is there more than one way to reduce a particular lambda expression?
3. If there is more than one reduction strategy, does every one (that terminates) lead to the same normal form expression?
4. Is there a reduction strategy that will guarantee that a normal form expression will be produced?

Answer 1: No

$(\lambda x . x \ x) \ (\lambda x . x \ x)$

$\Rightarrow_{\beta} (\lambda x . x \ x) \ (\lambda x . x \ x)$

$\Rightarrow_{\beta} (\lambda x . x \ x) \ (\lambda x . x \ x)$

$\Rightarrow_{\beta} \dots$

Answer 2: Yes

Example

$(\lambda x . \lambda y . y \ 5 \ x) \ ((\lambda m . \text{add } m \ 2) \ 6) \ \text{mul}$

Path One

$(\lambda x . \lambda y . y \ 5 \ x) \ ((\lambda m . \text{add } m \ 2) \ 6) \ \text{mul}$

$\Rightarrow_{\beta} (\lambda y . y \ 5 \ ((\lambda m . \text{add } m \ 2) \ 6)) \ \text{mul}$

$\Rightarrow_{\beta} \text{mul } 5 \ ((\lambda m . \text{add } m \ 2) \ 6)$

$\Rightarrow_{\beta} \text{mul } 5 \ (\text{add } 6 \ 2)$

$\Rightarrow_{\delta} \text{mul } 5 \ 8 \Rightarrow_{\delta} 40$

Path Two

$(\lambda x . \lambda y . y \ 5 \ x) \ ((\lambda m . \text{add } m \ 2) \ 6) \ \text{mul}$

$\Rightarrow_{\beta} (\lambda x . \lambda y . y \ 5 \ x) \ (\text{add } 6 \ 2) \ \text{mul}$

$\Rightarrow_{\delta} (\lambda x . \lambda y . y \ 5 \ x) \ 8 \ \text{mul}$

$\Rightarrow_{\beta} (\lambda y . y \ 5 \ 8) \ \text{mul}$

$\Rightarrow_{\beta} \text{mul } 5 \ 8 \Rightarrow_{\delta} 40$

Example $(\lambda y . 5) ((\lambda x . x x) (\lambda x . x x))$

Path One

$(\lambda y . 5) ((\lambda x . x x) (\lambda x . x x)) \Rightarrow_{\beta} 5$

Path Two

$(\lambda y . 5) ((\lambda x . x x) (\lambda x . x x))$

$\Rightarrow_{\beta} (\lambda y . 5) ((\lambda x . x x) (\lambda x . x x))$

$\Rightarrow_{\beta} (\lambda y . 5) ((\lambda x . x x) (\lambda x . x x)) \dots$

Normal order reduction

Reduce the leftmost *outermost* β -redex first.

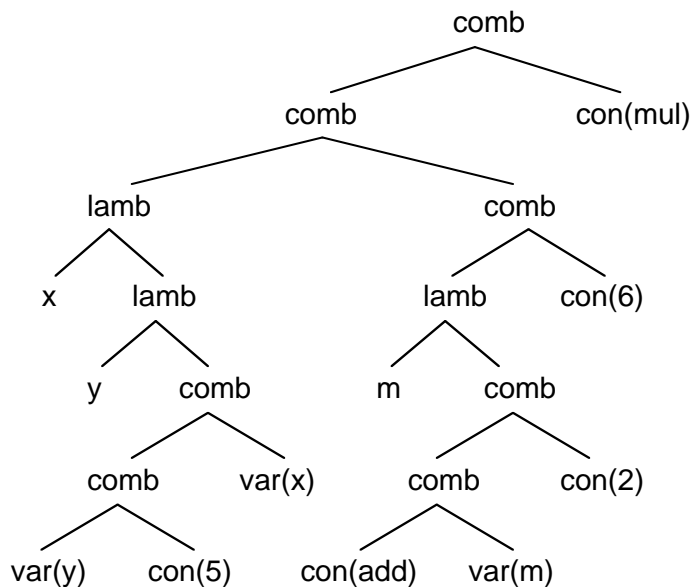
Applicative order reduction

Reduce the leftmost *innermost* β -redex first.

Outermost and Innermost Redexes

Example:

$(\lambda x . \lambda y . y 5 x) ((\lambda m . \text{add } m 2) 6) \text{ mul}$



Answer 3: Yes

Church-Rosser Theorem I:

For any lambda expressions E, F, and G, if $E \Rightarrow^* F$ and $E \Rightarrow^* G$, there is a lambda expression Z such that $F \Rightarrow^* Z$ and $G \Rightarrow^* Z$.

Corollary For any lambda expressions E, M, and N, if $E \Rightarrow^* M$ and $E \Rightarrow^* N$ where M and N are in normal form, then M and N are variants of each other (with respect to α -reduction).

Proof The only reduction possible for an expression in normal form is an α -reduction. Therefore the lambda expression Z in the theorem must be a variant of M and N by α -reduction only.

Answer 4: Yes

Church-Rosser Theorem II:

For any lambda expression E, if $E \Rightarrow^* N$ where N is in normal form, there is a normal order reduction from E to N.

Outcomes of a normal order reduction:

1. It reaches a unique (up to α -conversion) normal form lambda expression,
- or
2. It never terminates.

Applicative order reduction similar to call by value.

Normal order reduction similar to call by name.

Call by values applicative order reduction except that no β -redex or δ -redex within an abstraction is reduced. The argument to a function is evaluated before the function is applied.

Example:

$$\begin{aligned}
 & (\lambda x . (\lambda g . g x) \\
 & \quad ((\lambda f . \lambda y . \lambda z . ((\lambda n . \text{mul } n (f x y)) z)) \text{add})) \\
 & \quad \quad \quad ((\lambda m . \text{sqr } m) 3) \\
 \Rightarrow_{\beta} & (\lambda x . (\lambda g . g x) \\
 & \quad ((\lambda f . \lambda y . \lambda z . ((\lambda n . \text{mul } n (f x y)) z)) \text{add})) \\
 & \quad \quad \quad (\mathbf{\text{sqr } 3}) \\
 \Rightarrow_{\delta} & (\lambda \mathbf{x} . (\lambda g . g x) \\
 & \quad ((\lambda f . \lambda y . \lambda z . ((\lambda n . \text{mul } n (f x y)) z)) \text{add})) \mathbf{9} \\
 \Rightarrow_{\beta} & (\lambda g . g 9) \\
 & \quad ((\lambda f . \lambda y . \lambda z . ((\lambda n . \text{mul } n (f 9 y)) z)) \mathbf{\text{add}}) \\
 \Rightarrow_{\beta} & (\lambda \mathbf{g} . g 9) \\
 & \quad (\lambda \mathbf{y} . \lambda \mathbf{z} . ((\lambda n . \text{mul } n (\mathbf{\text{add } 9 y})) z)) \\
 \Rightarrow_{\delta} & (\lambda \mathbf{y} . \lambda \mathbf{z} . ((\lambda n . \text{mul } n (\mathbf{\text{add } 9 y})) z)) \mathbf{9} \\
 \Rightarrow_{\beta} & \lambda \mathbf{z} . ((\lambda n . \text{mul } n (\mathbf{\text{add } 9 9})) z)
 \end{aligned}$$

Constants in Pure Lambda Calculus

Constructor:

$$\text{define Pair} = \lambda a . \lambda b . \lambda f . f a b$$

Selectors:

$$\text{define Head} = \lambda g . g (\lambda a . \lambda b . a)$$

$$\text{define Tail} = \lambda g . g (\lambda a . \lambda b . b)$$

Correctness of the definitions:

Tail (Pair p q)

$$\begin{aligned}
 & \Rightarrow (\lambda \mathbf{g} . g (\lambda a . \lambda b . b)) \\
 & \quad \quad \quad (\lambda \mathbf{a} . \lambda \mathbf{b} . \lambda \mathbf{f} . f a b) \mathbf{p q} \\
 & \Rightarrow_{\beta} ((\lambda \mathbf{a} . \lambda \mathbf{b} . \lambda \mathbf{f} . f a b) \mathbf{p q}) (\lambda a . \lambda b . b) \\
 & \Rightarrow_{\beta} ((\lambda \mathbf{b} . \lambda \mathbf{f} . f p b) \mathbf{q}) (\lambda a . \lambda b . b) \\
 & \Rightarrow_{\beta} (\lambda \mathbf{f} . f p q) (\lambda \mathbf{a} . \lambda \mathbf{b} . \mathbf{b}) \\
 & \Rightarrow_{\beta} (\lambda \mathbf{a} . \lambda \mathbf{b} . b) \mathbf{p q} \\
 & \Rightarrow_{\beta} (\lambda \mathbf{b} . b) \mathbf{q} \Rightarrow_{\beta} \mathbf{q}
 \end{aligned}$$

Lists in Lambda Calculus

$$\text{define Nil} = \lambda x . \lambda a . \lambda b . a,$$

$$\begin{aligned}
 \text{define } [1, 2, 3, 4] \\
 = \text{Pair } 1 (\text{Pair } 2 (\text{Pair } 3 (\text{Pair } 4 \text{Nil}))).
 \end{aligned}$$

$$\text{Head (Tail (Tail [1, 2, 3, 4]))} \Rightarrow 3.$$

Natural Numbers

$$\text{define } 0 = \lambda f . \lambda x . x$$

$$\text{define } 1 = \lambda f . \lambda x . f x$$

$$\text{define } 2 = \lambda f . \lambda x . f (f x)$$

$$\text{define } 3 = \lambda f . \lambda x . f (f (f x))$$

$$: \quad \quad \quad :$$

Successor function: $\text{Succ} : \mathbb{N} \rightarrow \mathbb{N}$

$$\text{define Succ} = \lambda n . \lambda f . \lambda x . f (n f x)$$

Addition operation: $\text{Add} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}$

$$\text{define Add} = \lambda m . \lambda n . \lambda f . \lambda x . m f (n f x)$$

A Computation

Add 2 2

$$\begin{aligned}
 & \Rightarrow (\lambda \mathbf{m} . \lambda \mathbf{n} . \lambda \mathbf{f} . \lambda \mathbf{x} . m f (n f x)) \\
 & \quad \quad \quad (\lambda \mathbf{g} . \lambda \mathbf{y} . \mathbf{g} (\mathbf{g} \mathbf{y})) (\lambda \mathbf{h} . \lambda \mathbf{z} . \mathbf{h} (\mathbf{h} \mathbf{z})) \\
 & \Rightarrow (\lambda \mathbf{n} . \lambda \mathbf{f} . \lambda \mathbf{x} . (\lambda \mathbf{g} . \lambda \mathbf{y} . \mathbf{g} (\mathbf{g} \mathbf{y})) f (n f x)) \\
 & \quad \quad \quad (\lambda \mathbf{h} . \lambda \mathbf{z} . \mathbf{h} (\mathbf{h} \mathbf{z})) \\
 & \Rightarrow \lambda \mathbf{f} . \lambda \mathbf{x} . (\lambda \mathbf{g} . \lambda \mathbf{y} . \mathbf{g} (\mathbf{g} \mathbf{y})) \mathbf{f} \\
 & \quad \quad \quad ((\lambda \mathbf{h} . \lambda \mathbf{z} . \mathbf{h} (\mathbf{h} \mathbf{z})) f x) \\
 & \Rightarrow \lambda \mathbf{f} . \lambda \mathbf{x} . (\lambda \mathbf{y} . f (f y)) ((\lambda \mathbf{h} . \lambda \mathbf{z} . \mathbf{h} (\mathbf{h} \mathbf{z})) f x) \\
 & \Rightarrow \lambda \mathbf{f} . \lambda \mathbf{x} . (f (f ((\lambda \mathbf{h} . \lambda \mathbf{z} . \mathbf{h} (\mathbf{h} \mathbf{z})) f x))) \\
 & \Rightarrow \lambda \mathbf{f} . \lambda \mathbf{x} . (f (f ((\lambda \mathbf{z} . f (f z)) x))) \\
 & \Rightarrow \lambda \mathbf{f} . \lambda \mathbf{x} . (f (f (f (f x)))) = 4
 \end{aligned}$$

Functional Programming Languages

Functional programming languages, such as Lisp, Scheme, ML, Miranda, and Haskell, can be viewed as implementations of the lambda calculus.

These languages usually provide one or two syntactic expressions for readability:

1. let-expression

let $x=5$ **in** (add x 3)
 means $(\lambda x . (\text{add } x \ 3)) \ 5$

2. where-expression

(add x 3) **where** $x=5$
 means $(\lambda x . (\text{add } x \ 3)) \ 5$

They also provide a mechanism for defining (naming) functions, recursive functions in particular.

We justify the recursive definitions in Chapter 10.

A Lambda Calculus Evaluator

Concrete Syntax

$\langle \text{expression} \rangle ::= \langle \text{identifier} \rangle \mid \langle \text{constant} \rangle$
 $\mid (L \langle \text{identifier} \rangle^+ \langle \text{expression} \rangle)$
 $\mid (\langle \text{expression} \rangle^+ \langle \text{expression} \rangle)$

$\langle \text{constant} \rangle ::= \langle \text{numeral} \rangle \mid \text{true} \mid \text{false}$
 $\mid \text{succ} \mid \text{sqr} \mid \text{add} \mid \text{sub} \mid \text{mul}$

Two abbreviations

$(L \ x \ y \ E)$ means $(L \ x \ (L \ y \ E))$, which stands for $\lambda x . \lambda y . E$

$(E1 \ E2 \ E3)$ means $((E1 \ E2) \ E3)$.

Outermost parentheses are never omitted.

Abstract Syntax

Expression $::=$ var(Identifier)
 \mid con(Constant)
 \mid lamb(Identifier,Expression)
 \mid comb(Expression,Expression)

Lambda Expression:

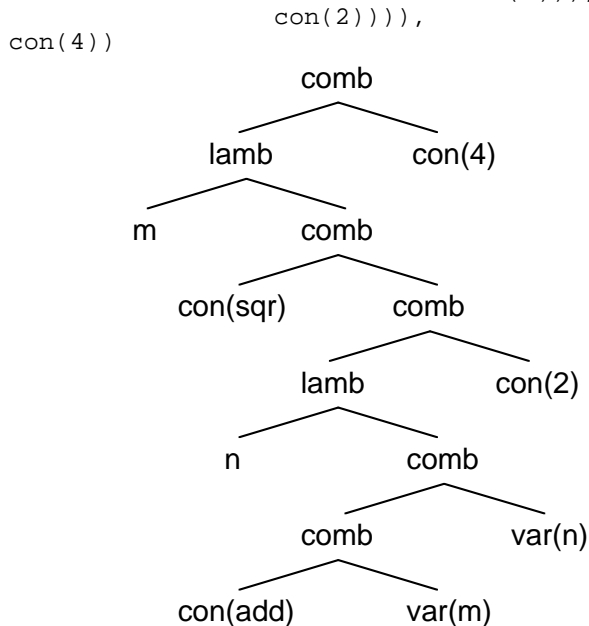
$(\lambda m . \text{sqr} ((\lambda n . (\text{add } m \ n)) \ 2)) \ 4$

Concrete Syntax:

$((L \ m \ (\text{sqr} ((L \ n \ (\text{add } m \ n)) \ 2))) \ 4)$

Abstract Syntax Tree:

$\text{comb}(\text{lamb}(m, \text{comb}(\text{con}(\text{sqr}), \text{comb}(\text{lamb}(n, \text{comb}(\text{comb}(\text{con}(\text{add}), \text{var}(m)), \text{var}(n))), \text{con}(2))))), \text{con}(4))$



Logic Grammar for Lambda Calculus

program(expr(E)) --> expr(E).

expr(lamb(X,E)) --> [lparen],[L],[var(X)],expr(E1),restlamb(E1,E).

restlamb(E,E) --> [rparen].
 restlamb(var(Y),lamb(Y,E)) --> expr(E1), restlamb(E1,E).

expr(E) --> [lparen],expr(E1),expr(E2),restcomb(E1,E2,E).

restcomb(E1,E2,comb(E1,E2)) --> [rparen].
 restcomb(E1,E2,E) --> expr(E3), restcomb(comb(E1,E2),E3,E).

expr(var(X)) --> [var(X)].

expr(con(X)) --> [num(X)].

expr(con(true)) --> [true].
 expr(con(false)) --> [false].

expr(con(succ)) --> [succ].
 expr(con(sqr)) --> [sqr].
 expr(con(add)) --> [add].
 expr(con(sub)) --> [sub].
 expr(con(mul)) --> [mul].

Lambda Calculus Evaluator

evaluate(E,NewE) :- reduce(E,TempE),
 evaluate(TempE,NewE).

evaluate(E,E).

Utilities

freevars(Exp,FV).

FV is a list containing the free variables that occur in Exp.

subst(Exp,Var,Exp1,NewExp).

NewExp is the expression Exp with each free occurrence of variable Var replace by Exp1.

variant(Var,VarList,NewVar).

NewVar is a “variant” of variable Var different from all the variables in VarList.

prime(Var,PrimeVar).

PrimeVar is variable Var with an apostrophe.

pp(Exp).

“Pretty-print” the expression Exp.

Substitution

subst(var(V),V,E1,E1). % a)

subst(var(X),V,E1,var(X)). % b)

subst(con(C),V,E1,con(C)). % c)

subst(comb(Rator,Rand),V,E1, % d)
 comb(NewRator,NewRand)) :-
 subst(Rator,V,E1,NewRator),
 subst(Rand,V,E1,NewRand).

subst(lamb(V,E),V,E1,lamb(V,E)). % e)

subst(lamb(X,E),V,E1,lamb(Z,NewE)) :-
 freevars(E1,F1),
 (member(X,F1), freevars(E,F), % g)
 union(F,[V],F2),
 union(F1,F2,FV),
 variant(X,FV,Z),
 subst(E,X,var(Z),TempE),
 subst(TempE,V,E1,NewE)
 ; subst(E,V,E1,NewE), Z=X) . % f)

Free Variables

freevars(var(X),[X]).

freevars(con(C),[]).

freevars(comb(Rator,Rand),FV) :-
 freevars(Rator,RatorFV),
 freevars(Rand,RandFV),
 union(RatorFV,RandFV,FV).

freevars(lamb(X,E),FV) :-
 freevars(E,F),delete(X,F,FV).

Variants

prime(X,PrimeX) :- name(X,L),
 concat(L,[39],NewL),
 name(PrimeX,NewL).

variant(X,L,NewX) :- member(X,L),
 prime(X,PrimeX),
 variant(PrimeX,L,NewX).

variant(X,L,X).

Reduction

reduce(comb(lamb(X,Body),Arg),R) :- % 1
 subst(Body,X,Arg,R).

reduce(comb(con(C),con(Arg)),R) :- % 2
 compute(C,Arg,R).

reduce(comb(Rator,Rand), % 3
 comb(NRator,Rand)) :-
 reduce(Rator,NRator).

reduce(comb(Rator,Rand), % 4
 comb(Rator,NRand)) :-
 reduce(Rand,NRand).

reduce(lamb(X,Body),lamb(X,NBody)) :- % 5
 reduce(Body,NBody).

Evaluating Constants

```
compute(succ,N,con(R)) :- R is N+1.  
compute(sqr,N,con(R)) :- R is N*N.  
compute(add,N,con(add(N))).  
compute(add(M),N,con(R)) :- R is M+N.  
compute(sub,N,con(sub(N))).  
compute(sub(M),N,con(R)) :- R is M-N.  
compute(mul,N,con(mul(N))).  
compute(mul(M),N,con(R)) :- R is M*N.
```

Top Level

```
evaluate(Exp,Result), nl,  
    write('Result = '), pp(Result), nl.
```

Try It `cp ~slonnegr/public/plf/normal .`
 `cp ~slonnegr/public/plf/twice .`