Lambda Calculus

- Very general mathematical language.
- Simple syntax, powerful semantics.
- Provides 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

<expression> ::=
<variable>
| <constant>
| **(** <expression$_1$> <expression$_2$> **)**
| **(** λ <variable> **.** <expression> **)**

- Variables - lower case identifiers
- Constants - any predefined value
  - 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

(succ 1) where 'succ' is the successor function

(zerop 0) where 'zerop' is the zero predicate (test if zero)

(add 3 5) where 'add' is the addition operation

Abstractions

- "anonymous" functions:

(λx . x)

is similar to

f(x) = x

but has no name.

Examples

(λm . (add m 1)) ≈ f(m) = m + 1

(λm . (λn . (sub n m))) ≈ g(m, n) = n - m

where ≈ means "is similar to".

Notational Conventions

1. Uppercase: metavariables for expressions.

2. Function application associates to the left.

   $E_1\ E_2\ E_3$ means $((E_1\ E_2)\ E_3)$

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

   $\lambda x\ .\ E_1\ E_2\ E_3$ 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.

   $\lambda x\ y\ z\ .\ E$ 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

$(\lambda m\ .\ (\lambda n\ .\ (sub\ n\ m)))$

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

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

So

$((\lambda m\ .\ (\lambda n\ .\ (sub\ n\ m)))\ 1) = (\lambda n\ .\ (sub\ n\ 1))$

returns a function

whereas g is a function of *2* parameters

Using signatures (domain and codomain):

$g : N \times N \rightarrow N$

$(\lambda m\ .\ (\lambda n\ .\ (sub\ n\ m))) : N \rightarrow (N \rightarrow N)$

where 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 "< , >" is the tuple constructor, define

Curry = (λf . λx . λy . f <x, y>) and
Uncurry = (λf . λ<x, y> . f x y)

Then

Curry g = (λx . (λy . (y - x)))
= (λx . (λy . (sub y x)))

Uncurry (λx . (λy . (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**.

((λx . (y x)) x)

First two x's are bound.

Last x and the y are free.

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

a) $FV(c) = \varnothing$ for any constant c

b) $FV(x) = \{x\}$ for any variable x

c) $FV(E_1\ E_2) = FV(E_1) \cup FV(E_2)$

d) $FV(\lambda x\ .\ E) = FV(E) - \{x\}$

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

FV((λx . x) y) = { y }

FV((λx . 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→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 called **valid** or **safe** if no free variable in E$_1$ becomes bound as a result of the substitution E[v→E$_1$].

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

An Invalid Substitution

(λx . (sub x y))[y→x] = (λx . (sub x x))

First function: Subtract y from the argument.

Second function: Return zero.

E[v→E₁] is defined by:

a) $v[v \to E_1] = E_1$ for any variable v

b) $x[v \to E_1] = x$ for any variable $x \neq v$

c) $c[v \to E_1] = c$ for any constant c

d) $(E_{rator}\ E_{rand})[v \to E_1] = (E_{rator}[v \to E_1])\ (E_{rand}[v \to E_1])$

e) $(\lambda v\ .\ E)[v \to E_1] = (\lambda v\ .\ E)$

f) $(\lambda x\ .\ E)[v \to E_1] = \lambda x\ .\ (E[v \to E_1])$
when $x \neq v$ and $x \notin FV(E_1)$

g) $(\lambda x\ .\ E)[v \to E_1] = \lambda z\ .\ (E[x \to z][v \to E_1])$
when $x \neq v$ and $x \in FV(E_1)$,
where $z \neq v$ and $z \notin FV(E\ E_1)$

Substitution Examples

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

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

Reduction Rules

α-reduction:

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

$$\lambda v\ .\ E \Rightarrow_\alpha \lambda w\ .\ (E[v \to w])$$

provided that w does not occur at all in E, which makes the substitution E[v→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,

$$(\lambda v\ .\ E)\ E_1 \Rightarrow_\beta E[v \to E_1]$$

provided the substitution $E[v \to E_1]$ 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:

$$\lambda m\ .\ (sqr\ m) \Rightarrow_\eta sqr$$

$$\lambda n\ .\ (mul\ 2\ n) \Rightarrow_\eta (mul\ 2).$$

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.

(succ 13) $\Rightarrow_\delta$ 14

(mul 4 7) $\Rightarrow_\delta$ 28

(not false) $\Rightarrow_\delta$ true.

Reduction Example

Thrice (λm . (mul 2 m)) 7
where Thrice = λf . λx . f (f (f x))

Thrice (λm . (mul 2 m)) 7

$\Rightarrow$ (**λf** . λx . f (f (f x))) **(λm . (mul 2 m))** 7

$\Rightarrow_\beta$ (**λx** . (λm . (mul 2 m)) ((λm . (mul 2 m)) ((λm . (mul 2 m)) x))) **7**

$\Rightarrow_\beta$ (**λm** . (mul 2 m)) **((λm . (mul 2 m)) ((λm . (mul 2 m)) 7))**

$\Rightarrow_\beta$ mul 2 ((**λm** . (mul 2 m)) **((λm . (mul 2 m)) 7)**)

$\Rightarrow_\beta$ mul 2 (mul 2 ((**λm** . (mul 2 m)) **7**))

$\Rightarrow_\beta$ mul 2 (mul 2 (mul 2 7)) $\Rightarrow_\delta$ mul 2 (mul 2 14)

$\Rightarrow_\delta$ 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

(λ**x** . x x) **(λx . x x)**
$\Rightarrow_\beta$ (λ**x** . x x) **(λx . x x)**
$\Rightarrow_\beta$ (λ**x** . x x) **(λx . x x)**
$\Rightarrow_\beta$ …

Answer 2: Yes

Example:
(λx . λy . y 5 x) ((λm . add m 2) 6) mul

Path One:
(λ**x** . λy . y 5 x) **((λm . add m 2) 6)** mul

$\Rightarrow_\beta$ (**λy** . y 5 ((λm . add m 2) 6)) **mul**

$\Rightarrow_\beta$ mul 5 ((**λm** . add m 2) **6**)

$\Rightarrow_\beta$ mul 5 (add 6 2)

$\Rightarrow_\delta$ mul 5 8 $\Rightarrow_\delta$ 40

Path Two:
(λx . λy . y 5 x) ((**λm** . add m 2) **6**) mul

$\Rightarrow_\beta$ (λx . λy . y 5 x) (add 6 2) mul

$\Rightarrow_\delta$ (**λx** . λy . y 5 x) **8** mul

$\Rightarrow_\beta$ (**λy** . y 5 8) **mul**

$\Rightarrow_\beta$ mul 5 8 $\Rightarrow_\delta$ 40

Example: (λy . 5) ((λx . x x) (λx . x x))

Path One:

(**λy** . 5) **((λx . x x) (λx . x x))** $\Rightarrow_\beta$ 5

Path Two:
(λy . 5) ((**λx** . x x) **(λx . x x)**)

$\Rightarrow_\beta$ (λy . 5) ((**λx** . x x) **(λx . x x)**)

$\Rightarrow_\beta$ (λy . 5) ((**λx** . x x) **(λx . x x)**) …

Normal order reduction

Reduce the leftmost *outermost* β-redex first.

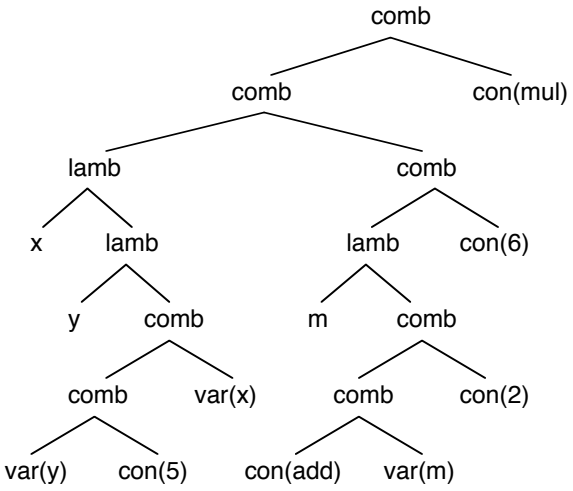
Applicative order reduction

Reduce the leftmost *innermost* β-redex first.

Outermost and Innermost Redexes

Example:

(λx . λy . y 5 x) ((λm . add m 2) 6) 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 value is 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:

(λx . (λg . g x)
((λf . λy . λz . ((λn . mul n (f x y)) z)) add))
((**λm** . sqr m) **3**)

$\Rightarrow_\beta$ (λx . (λg . g x)
((λf . λy . λz . ((λn . mul n (f x y)) z)) add))
(**sqr 3**)

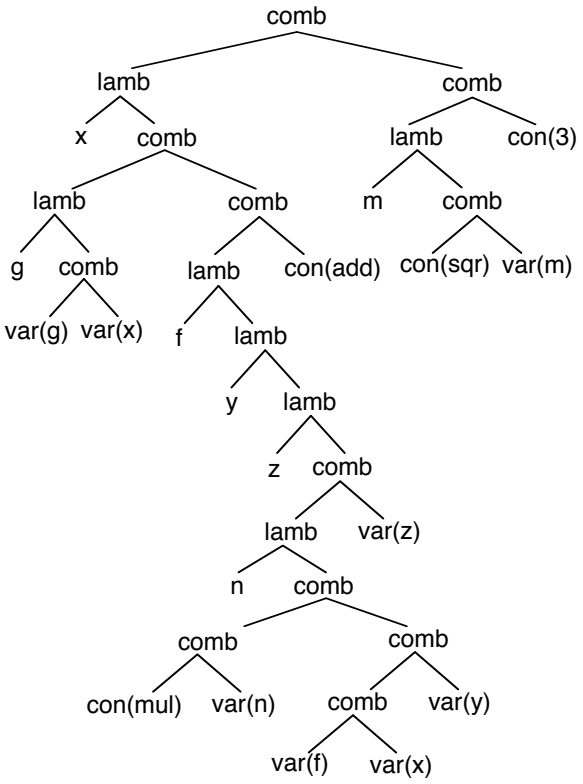
$\Rightarrow_\delta$ (**λx** . (λg . g x)
((λf . λy . λz . ((λn . mul n (f x y)) z)) add)) **9**

$\Rightarrow_\beta$ (λg . g 9)
((**λf** . λy . λz . ((λn . mul n (f 9 y)) z)) **add**)

$\Rightarrow_\beta$ (**λg** . g 9)
(λy . λz . ((λn . mul n (add 9 y)) z))

$\Rightarrow_\beta$ (**λy** . λz . ((λn . mul n (add 9 y)) z)) **9**

$\Rightarrow_\beta$ λz . ((λn . mul n (add 9 9)) z)



Constants in Pure Lambda Calculus

Constructor:

define Pair = λa . λb . λf . f a b

Selectors:

define Head = λg . g (λa . λb . a)

define Tail = λg . g (λa . λb . b)

Correctness of the definitions:

Tail (Pair p q)

⇒ (**λg** . g (λa . λb . b))
((λa . λb . λf . f a b) p q)

$\Rightarrow_\beta$ ((**λa** . λb . λf . f a b) **p** q) (λa . λb . b)

$\Rightarrow_\beta$ ((**λb** . λf . f p b) **q**) (λa . λb . b)

$\Rightarrow_\beta$ (**λf** . f p q) **(λa . λb . b)**

$\Rightarrow_\beta$ (**λa** . λb . b) **p** q

$\Rightarrow_\beta$ (**λb** . b) **q** $\Rightarrow_\beta$ q

Lists in Lambda Calculus

define Nil = λx . λa . λb . a,

define [1, 2, 3, 4]
= Pair 1 (Pair 2 (Pair 3 (Pair 4 Nil))).

Head (Tail (Tail [1, 2, 3, 4])) ⇒ 3.

Natural Numbers

define 0 = λf . λx . x
define 1 = λf . λx . f x
define 2 = λf . λx . f (f x)
define 3 = λf . λx . f (f (f x))
: :

Successor function: Succ : N → N

define Succ = λn . λf . λx . f (n f x)

Addition operation: Add : N → N → N

define Add = λm . λn . λf . λx . m f (n f x)

A Computation

Add 2 2

⇒ (**λm** . λn . λf . λx . m f (n f x))
(λg . λy . g (g y)) (λh . λz . h (h z))

⇒ (**λn** . λf . λx . (λg . λy . g (g y)) f (n f x))
(λh . λz . h (h z))

⇒ λf . λx . (**λg** . λy . g (g y)) **f**
((λh . λz . h (h z)) f x)

⇒ λf . λx . (**λy** . f (f y)) **((λh . λz . h (h z)) f x)**

⇒ λf . λx . (f (f ((**λh** . λz . h (h z)) **f** x)))

⇒ λf . λx . (f (f ((**λz** . f (f z)) **x**)))

⇒ λf . λx . (f (f (f (f x)))) = 4

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 (λx . (add x 3)) 5

2. where-expression
   (add x 3) **where** x=5
   means (λx . (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

<expression> ::= <identifier> | <constant>
| **(** **L** <identifier>+ <expression> **)**
| **(** <expression>+ <expression> **)**

<constant> ::= <numeral> | **true** | **false**
|**succ** | **sqr** | **add** | **sub** | **mul**

Two abbreviations

(L x y E) means (L x (L y E)), which stands
λx . λy . E

(E1 E2 E3) means ((E1 E2) E3).

Outermost parentheses are never omitted.

Abstract Syntax

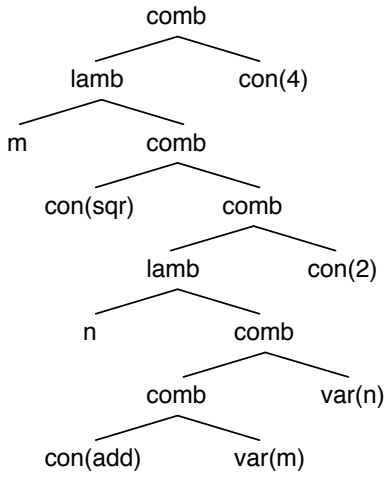
Expression ::= var(Identifier)
| con(Constant)
| lamb(Identifier,Expression)
| comb(Expression,Expression)

Lambda Expression:
(λm . sqr ((λn . (add m n)) 2)) 4

Concrete Syntax:
((L m (sqr ((L n (add m n)) 2))) 4)

Abstract Syntax Tree:

```
comb(lamb(m,comb(con(sqr),
                 comb(lamb(n,comb(comb(con(add),
                                       var(m)),
                                  var(n))),
                      con(2)))),
     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 .