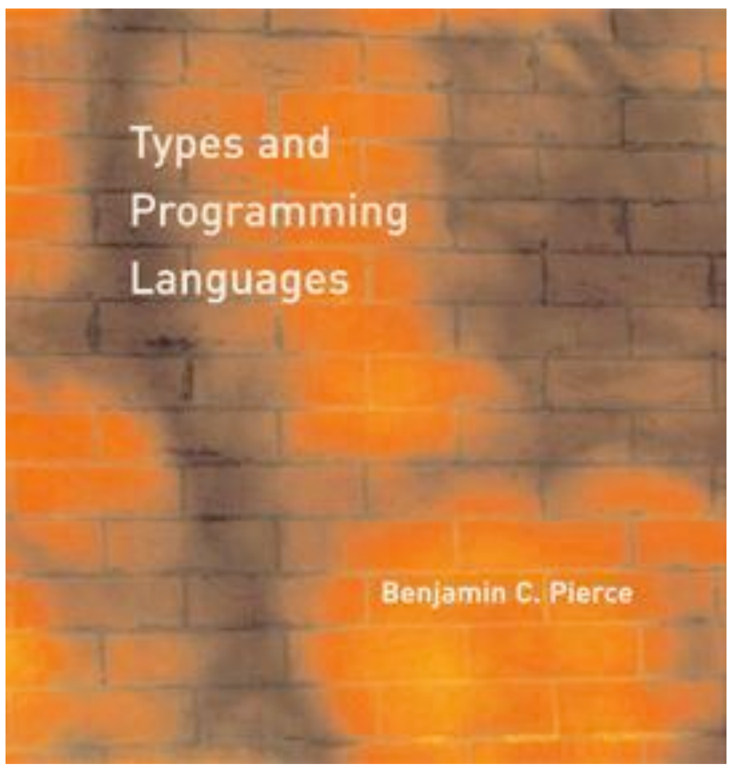
Programming Languages
Fall 2013



Lecture 4: Lambda-Calculus 1

Prof. Liang Huang

huang@qc.cs.cuny.edu

The Lambda Calculus

Alonzo Church

Alan Turing

The lambda-calculus

- If our previous language of arithmetic expressions was the simplest nontrivial programming language, then the lambda-calculus is the simplest interesting programming language...
  - Turing complete
  - higher order (functions as data)
  - main new feature: variable binding and lexical scope
- The e. coli of programming language research
- The foundation of many real-world programming language designs (including ML, Haskell, Scheme, Lisp, ...)

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x   =  succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

```
plus3 x  =  succ (succ (succ x))
```

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

$$\texttt{plus3} \quad = \quad \lambda\texttt{x. succ (succ (succ x))}$$

This function exists independent of the name `plus3`.

Intuitions

Suppose we want to describe a function that adds three to any number we pass it. We might write

$$\texttt{plus3 x} \quad = \quad \texttt{succ (succ (succ x))}$$

That is, "`plus3 x` is `succ (succ (succ x))`."

Q: What is `plus3` itself?

A: `plus3` is the function that, given `x`, yields `succ (succ (succ x))`.

$$\texttt{plus3} \quad = \quad \lambda\texttt{x. succ (succ (succ x))}$$

This function exists independent of the name `plus3`.

On this view, `plus3 (succ 0)` is just a convenient shorthand for "the function that, given `x`, yields `succ (succ (succ x))`, applied to `succ 0`."

$$\texttt{plus3 (succ 0)} \quad = \quad (\lambda\texttt{x. succ (succ (succ x))) (succ 0)}$$

Essentials

We have introduced two primitive syntactic forms:

- **abstraction** of a term `t` on some subterm `x`:

  `λx. t`

  "The function that, when applied to a value `v`, yields `t` with `v` in place of `x`."

- **application** of a function to an argument:

  t_1 t_2

  "the function t_1 applied to the argument t_2"

Recall that we wrote anonymous functions "`fun x → t`" in OCaml.

Abstractions over Functions

Consider the λ-abstraction

```
g  =  λf. f (f (succ 0))
```

Note that the parameter variable `f` is used in the function position in the body of `g`. Terms like `g` are called higher-order functions.

If we apply `g` to an argument like `plus3`, the "substitution rule" yields a nontrivial computation:

```
g plus3  =    (λf. f (f (succ 0))) (λx. succ (succ (succ x)))
         i.e. (λx. succ (succ (succ x)))
                ((λx. succ (succ (succ x))) (succ 0))
         i.e. (λx. succ (succ (succ x)))
                (succ (succ (succ (succ 0))))
         i.e. succ (succ (succ (succ (succ (succ (succ 0))))))
```

Abstractions Returning Functions

Consider the following variant of g:

$$\texttt{double} = \lambda\texttt{f}.\ \lambda\texttt{y}.\ \texttt{f (f y)}$$

I.e., double is the function that, when applied to a function f, yields a function that, when applied to an argument y, yields f (f y).

```
Prelude> let g = \f -> \y -> f (f y)
Prelude> g (+ 2) 3
7
```

Example

```
      double plus3 0
=     (λf. λy. f (f y))
        (λx. succ (succ (succ x)))
        0
i.e.  (λy. (λx. succ (succ (succ x)))
              ((λx. succ (succ (succ x))) y))
        0
i.e.  (λx. succ (succ (succ x)))
              ((λx. succ (succ (succ x))) 0)
i.e.  (λx. succ (succ (succ x)))
              (succ (succ (succ 0)))
i.e.  succ (succ (succ (succ (succ (succ 0)))))
```

The Pure Lambda-Calculus

As the preceding examples suggest, once we have λ-abstraction and application, we can throw away all the other language primitives and still have left a rich and powerful programming language.

In this language — the "pure lambda-calculus"— everything is a function.

- Variables always denote functions
- Functions always take other functions as parameters
- The result of a function is always a function

Formalities

Syntax

`t`	::=	terms
	`x`	variable
	`λx.t`	abstraction
	`t t`	application

Terminology:

- terms in the pure λ-calculus are often called λ-terms
- terms of the form `λx. t` are called λ-abstractions or just abstractions

Scope

The λ-abstraction term $\lambda x.t$ binds the variable x.

The scope of this binding is the body t.

Occurrences of x inside t are said to be bound by the abstraction.

Occurrences of x that are not within the scope of an abstraction binding x are said to be free.

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

Scope

The λ-abstraction term λx.t **binds** the variable x.

The **scope** of this binding is the **body** t.

Occurrences of x inside t are said to be **bound** by the abstraction.

Occurrences of x that are **not** within the scope of an abstraction binding x are said to be **free**.

```
λx. λy. x y z
λx. (λy. z y) y
```

Values

v	::=		values
		λx.t	abstraction value
t	::=		terms
		x	variable
		λx.t	abstraction
		t t	application

Operational Semantics

Computation rule:

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[x \mapsto v_2]t_{12}$ is "the term that results from substituting free occurrences of x in t_{12} with v_{12}."

Operational Semantics

Computation rule:

$$(\lambda x.t_{12})\ v_2 \longrightarrow [x \mapsto v_2]t_{12} \qquad \text{(E-AppAbs)}$$

Notation: $[x \mapsto v_2]t_{12}$ is "the term that results from substituting free occurrences of x in t_{12} with v_{12}."

Congruence rules:

call by name:

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}$$

$$(\lambda x.t_{12})\ t_2 \longrightarrow [x \mapsto t_2]t_{12}$$

big-step semantics

$$\lambda x.t \Downarrow \lambda x.t$$

$$\frac{t_1 \Downarrow \lambda x.t_{12} \qquad t_2 \Downarrow v_2 \qquad [x \mapsto v_2]t_{12} \Downarrow t'}{t_1\ t_2 \Downarrow t'}$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad \text{(E-App1)}$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad \text{(E-App2)}$$

Terminology

A term of the form $(\lambda x.t)\ v$ — that is, a λ-abstraction applied to a **value** — is called a **redex** (short for "reducible expression").

Alternative evaluation strategies

Strictly speaking, the language we have defined is called the pure, call-by-value lambda-calculus.

The evaluation strategy we have chosen — call by value — reflects standard conventions found in most mainstream languages.

Some other common ones:

- Call by name (cf. Haskell)
- Normal order (leftmost/outermost)
- Full (non-deterministic) beta-reduction

Programming in the Lambda-Calculus

Multiple arguments

Above, we wrote a function `double` that returns a function as an argument.

$$\texttt{double} = \lambda\texttt{f.}\ \lambda\texttt{y.}\ \texttt{f (f y)}$$

This idiom — a λ-abstraction that does nothing but immediately yield another abstraction — is very common in the λ-calculus.

In general, λ`x.` λ`y.` `t` is a function that, given a value `v` for `x`, yields a function that, given a value `u` for `y`, yields `t` with `v` in place of `x` and `u` in place of `y`.

That is, λ`x.` λ`y.` `t` is a two-argument function.

(Recall the discussion of currying in OCaml.)

Syntactic conventions

Since λ-calculus provides only one-argument functions, all multi-argument functions must be written in curried style.

The following conventions make the linear forms of terms easier to read and write:

- Application associates to the left

  E.g., `t u v` means `(t u) v`, not `t (u v)`

- Bodies of λ- abstractions extend as far to the right as possible

  E.g., `λx. λy. x y` means `λx. (λy. x y)`, not `λx. (λy. x) y`

The "Church Booleans"

```
tru = λt. λf. t
fls = λt. λf. f
```

	tru v w	
$=$	(λt.λf.t) v w	by definition
$\longrightarrow$	(λf. v) w	reducing the underlined redex
$\longrightarrow$	v	reducing the underlined redex

	fls v w	
$=$	(λt.λf.f) v w	by definition
$\longrightarrow$	(λf. f) w	reducing the underlined redex
$\longrightarrow$	w	reducing the underlined redex

Functions on Booleans

$$\texttt{not} \quad = \quad \lambda\texttt{b. b fls tru}$$

That is, `not` is a function that, given a boolean value `v`, returns `fls` if `v` is `tru` and `tru` if `v` is `fls`.

Functions on Booleans

$$\texttt{and} \;=\; \lambda\texttt{b.}\ \lambda\texttt{c.}\ \texttt{b c fls}$$

That is, `and` is a function that, given two boolean values `v` and `w`, returns `w` if `v` is `tru` and `fls` if `v` is `fls` (short-circuit ?)

Thus `and v w` yields `tru` if both `v` and `w` are `tru` and `fls` if either `v` or `w` is `fls`.

what about or?

Pairs

```
pair = λf.λs.λb. b f s
fst = λp. p tru
snd = λp. p fls
```

That is, `pair v w` is a function that, when applied to a boolean value `b`, applies `b` to `v` and `w`.

By the definition of booleans, this application yields `v` if `b` is `tru` and `w` if `b` is `fls`, so the first and second projection functions `fst` and `snd` can be implemented simply by supplying the appropriate boolean.

	fst (pair v w)	
=	fst ((λf. λs. λb. b f s) v w)	by definition
→	fst ((λs. λb. b v s) w)	reducing the underlined redex
→	fst (λb. b v w)	reducing the underlined redex
=	(λp. p tru) (λb. b v w)	by definition
→	(λb. b v w) tru	reducing the underlined redex
→	tru v w	reducing the underlined redex
$\longrightarrow^*$	v	as before.

Example

	`fst (pair v w)`	
$=$	`fst ((λf. λs. λb. b f s) v w)`	by definition
$\longrightarrow$	`fst ((λs. λb. b v s) w)`	reducing the underlined redex
$\longrightarrow$	`fst (λb. b v w)`	reducing the underlined redex
$=$	`(λp. p tru) (λb. b v w)`	by definition
$\longrightarrow$	`(λb. b v w) tru`	reducing the underlined redex
$\longrightarrow$	`tru v w`	reducing the underlined redex
$\longrightarrow^*$	`v`	as before.

Church numerals

Idea: represent the number n by a function that "repeats some action n times."

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

what about "fls"? maybe C is right...

That is, each number n is represented by a term c_n that takes two arguments, s and z (for "successor" and "zero"), and applies s, n times, to z.

Functions on Church Numerals

Successor:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

another solution?

```
scc2 = λn. λs. λz. n s (s z);
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
scc2 = λn. λs. λz. n s (s z);
```

Addition:

c_0 = λs. λz. z
c_1 = λs. λz. s z
c_2 = λs. λz. s (s z)
c_3 = λs. λz. s (s (s z))

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
scc2 = λn. λs. λz. n s (s z);
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
scc2 = λn. λs. λz. n s (s z);
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
scc2 = λn. λs. λz. n s (s z);
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c0
```

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
scc2 = λn. λs. λz. n s (s z);
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c0
```

Zero test:

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
scc2 = λn. λs. λz. n s (s z);
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c0
```

Zero test:

```
iszro = λm. m (λx. fls) tru
```

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
scc2 = λn. λs. λz. n s (s z);
```

Addition:

```
plus = λm. λn. λs. λz. m s (n s z)
```

Multiplication:

```
times = λm. λn. m (plus n) c0
```

Zero test:

```
iszro = λm. m (λx. fls) tru
```

```
c0 = λs. λz. z
c1 = λs. λz. s z
c2 = λs. λz. s (s z)
c3 = λs. λz. s (s (s z))
```

```
times2 = λm. λn. λs. λz. m (n s) z;
```

Or, more compactly:

```
times3 = λm. λn. λs. m (n s);
```

```
power1 = λm. λn. m (times n) c1;
power2 = λm. λn. m n;
```

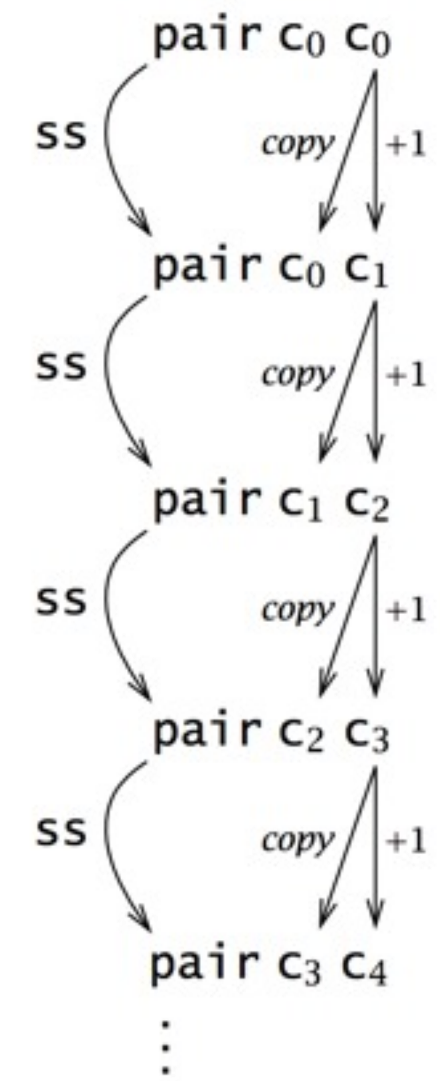
What about predecessor?

Predecessor

```
zz = pair c0 c0

ss = λp. pair (snd p) (scc (snd p))
```

Predecessor

```
zz = pair c0 c0

ss = λp. pair (snd p) (scc (snd p))

prd = λm. fst (m ss zz)
```



Questions:

1. what's the complexity of prd?
2. how to define equal?
3. how to define subtract?

Normal forms

Recall:

- A normal form is a term that cannot take an evaluation step.
- A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus?

Prove it.

Normal forms

Recall:

- A normal form is a term that cannot take an evaluation step.
- A stuck term is a normal form that is not a value.

Are there any stuck terms in the pure λ-calculus?

Prove it.

Does every term evaluate to a normal form?

Prove it.

Divergence

$$\texttt{omega} \quad = \quad \texttt{(}\lambda\texttt{x. x x) (}\lambda\texttt{x. x x)}$$

Note that `omega` evaluates in one step to itself!

So evaluation of `omega` never reaches a normal form: it diverges.

Divergence

$$\texttt{omega} \quad = \quad \texttt{(}\lambda\texttt{x. x x) (}\lambda\texttt{x. x x)}$$

Note that `omega` evaluates in one step to itself!

So evaluation of `omega` never reaches a normal form: it diverges.

Being able to write a divergent computation does not seem very useful in itself. However, there are variants of `omega` that are very useful...

Iterated Application

Suppose $\mathtt{f}$ is some λ-abstraction, and consider the following term:

$$\mathtt{Y_f} \quad = \quad \mathtt{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}$$

Iterated Application

Suppose $\mathtt{f}$ is some λ-abstraction, and consider the following term:

$$\mathtt{Y_f} \quad = \quad \mathtt{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}$$

Now the "pattern of divergence" becomes more interesting:

$$\mathtt{Y_f}$$
$$=$$
$$\underline{\mathtt{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}}$$
$$\longrightarrow$$
$$\mathtt{f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})}$$
$$\longrightarrow$$
$$\mathtt{f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))}))}$$
$$\longrightarrow$$
$$\mathtt{f\ (f\ (f\ (\underline{(\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))})))}$$
$$\longrightarrow$$
$$\cdots$$

Y_f is still *not* very useful, since (like omega), all it does is diverge.

Is there any way we could "slow it down"?

Delaying Divergence

$$\texttt{poisonpill} \quad = \quad \lambda\texttt{y. omega}$$

Note that `poisonpill` is a value — it it will only diverge when we actually apply it to an argument. This means that we can safely pass it as an argument to other functions, return it as a result from functions, etc.

$$\underline{(\lambda\texttt{p. fst (pair p fls) tru) poisonpill}}$$
$$\longrightarrow$$
$$\texttt{fst (pair poisonpill fls) tru}$$
$$\longrightarrow^*$$
$$\underline{\texttt{poisonpill tru}}$$
$$\longrightarrow$$
$$\texttt{omega}$$
$$\longrightarrow$$
$$\ldots$$

Cf. thunks in OCaml.

A delayed variant of omega

Here is a variant of omega in which the delay and divergence are a bit more tightly intertwined:

$$\texttt{omegav} \;=\; \lambda y.\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ y$$

Note that omegav is a normal form. However, if we apply it to any argument v, it diverges:

$$\begin{array}{c}
\texttt{omegav v} \\
= \\
\underline{(\lambda y.\ \ (\lambda x.\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ y)\ v} \\
\longrightarrow \\
\underline{(\lambda x.\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ (\lambda y.\ x\ x\ y))}\ v \\
\longrightarrow \\
(\lambda y.\ \ (\lambda x.\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ (\lambda y.\ x\ x\ y))\ y)\ v \\
= \\
\texttt{omegav v}
\end{array}$$

Another delayed variant

Suppose f is a function. Define

$$Z_f = \lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y$$

This term combines the "added f" from Y_f with the "delayed divergence" of omegav.

If we now apply Z_f to an argument v, something interesting happens:

$$Z_f\ v$$
$$=$$
$$\underline{(\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y)\ v}$$
$$\longrightarrow$$
$$\underline{(\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))}\ v$$
$$\longrightarrow$$
$$f\ (\lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y)\ v$$
$$=$$
$$f\ Z_f\ v$$

Since Z_f and v are both values, the next computation step will be the reduction of $f\ Z_f$ — that is, before we "diverge," f gets to do some computation.

Now we are getting somewhere.

Recursion

Let

```
f  =  λfct.
        λn.
          if n=0 then 1
          else n * (fct (pred n))
```

`f` looks just the ordinary factorial function, except that, in place of a recursive call in the last time, it calls the function `fct`, which is passed as a parameter.

N.b.: for brevity, this example uses “real” numbers and booleans, infix syntax, etc. It can easily be translated into the pure lambda-calculus (using Church numerals, etc.).

Z_f = λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y

We can use Z to "tie the knot" in the definition of f and obtain a real recursive factorial function:

```
f  =  λfct.
        λn.
          if n=0 then 1
          else n * (fct (pred n))
```

Z_f 3

$\longrightarrow^*$

f Z_f 3

=

(λfct. λn. ...) Z_f 3

$\longrightarrow$ $\longrightarrow$

if 3=0 then 1 else 3 * (Z_f (pred 3))

$\longrightarrow^*$

3 * (Z_f (pred 3)))

$\longrightarrow$

3 * (Z_f 2)

$\longrightarrow^*$

3 * (f Z_f 2)

...

A Generic Z

If we define

$$Z = \lambda f.\ Z_f$$

i.e.,

$$Z = \lambda f.\ \lambda y.\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ (\lambda x.\ f\ (\lambda y.\ x\ x\ y))\ y$$

then we can obtain the behavior of Z_f for any f we like, simply by applying Z to f.

$$Z\ f \longrightarrow Z_f$$

For example:

```
fact   =   Z  ( λfct.
                 λn.
                   if n=0 then 1
                   else n * (fct (pred n)) )
```

Z_f = λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y

Technical note:

The term Z here is essentially the same as the fix discussed the book.

```
Z    =  λf. λy. (λx. f (λy. x x y)) (λx. f (λy. x x y)) y
fix  =  λf. (λx. f (λy. x x y)) (λx. f (λy. x x y))
```

Z is hopefully slightly easier to understand, since it has the property that Z f v $\longrightarrow^*$ f (Z f) v, which fix does not (quite) share.

fix is the (call-by-value) Y-combinator

Y Combinator