CIS 500
Software Foundations
Fall 2006

September 25

The Lambda Calculus

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)
- Indeed, in the lambda-calculus, *all* computation happens by means of function abstraction and application.
- 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

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

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`.

λ`x. t` is written "`fun x` $\rightarrow$ `t`" in OCaml.

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

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

Abstractions over Functions

Consider the λ-abstraction

$$\texttt{g} \quad = \quad \lambda\texttt{f. f (f (succ 0))}$$

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:

$$\begin{array}{ll}
\multicolumn{2}{l}{\texttt{g plus3}} \\
= & (\lambda\texttt{f. f (f (succ 0))) (}\lambda\texttt{x. succ (succ (succ x)))} \\
\textit{i.e.} & (\lambda\texttt{x. succ (succ (succ x)))} \\
& \quad \texttt{((}\lambda\texttt{x. succ (succ (succ x))) (succ 0))} \\
\textit{i.e.} & (\lambda\texttt{x. succ (succ (succ x)))} \\
& \quad \texttt{(succ (succ (succ (succ 0))))} \\
\textit{i.e.} & \texttt{succ (succ (succ (succ (succ (succ (succ 0))))))}
\end{array}$$

Abstractions Returning Functions

Consider the following variant of `g`:

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

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)`.

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

$\mathtt{t}$	::=	*terms*
	$\mathtt{x}$	*variable*
	$\lambda\mathtt{x.t}$	*abstraction*
	$\mathtt{t\ t}$	*application*

*Term*inology:

- terms in the pure λ-calculus are often called λ-*terms*
- terms of the form $\lambda\mathtt{x.\ t}$ are called λ-*abstractions* or just *abstractions*

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., $\lambda x.\ \lambda y.\ x\ y$ *means* $\lambda x.\ (\lambda y.\ x\ y)$, *not* $\lambda x.\ (\lambda y.\ x)\ y$

Scope

The λ-abstraction term $\lambda\texttt{x.t}$ *binds* the variable $\texttt{x}$.
The *scope* of this binding is the *body* $\texttt{t}$.
Occurrences of $\texttt{x}$ inside $\texttt{t}$ are said to be *bound* by the abstraction.
Occurrences of $\texttt{x}$ that are *not* within the scope of an abstraction binding $\texttt{x}$ are said to be *free*.

$$\lambda\texttt{x. }\lambda\texttt{y. x y z}$$

Scope

The λ-abstraction term $\lambda\mathtt{x.t}$ *binds* the variable $\mathtt{x}$.
The *scope* of this binding is the *body* $\mathtt{t}$.
Occurrences of $\mathtt{x}$ inside $\mathtt{t}$ are said to be *bound* by the abstraction.
Occurrences of $\mathtt{x}$ that are *not* within the scope of an abstraction binding $\mathtt{x}$ are said to be *free*.

$$\lambda\mathtt{x.}\ \lambda\mathtt{y.}\ \mathtt{x\ y\ z}$$
$$\lambda\mathtt{x.}\ (\lambda\mathtt{y.}\ \mathtt{z\ y})\ \mathtt{y}$$

Values

$\mathtt{v}$ ::=		*values*
	$\lambda\mathtt{x.t}$	*abstraction value*

Operational Semantics

Computation rule:

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

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:

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

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:

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

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.

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

This idiom — a λ-abstraction that does nothing but immediately yield another abstraction — is very common in the λ-calculus.
In general, $\lambda\texttt{x.}\ \lambda\texttt{y.}\ \texttt{t}$ 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, $\lambda\texttt{x.}\ \lambda\texttt{y.}\ \texttt{t}$ 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., $\lambda x.\ \lambda y.\ x\ y$ *means* $\lambda x.\ (\lambda y.\ x\ y)$, *not* $\lambda x.\ (\lambda y.\ x)\ y$

The "Church Booleans"

$$
\begin{aligned}
\texttt{tru} &= \lambda\texttt{t.}\ \lambda\texttt{f.}\ \texttt{t} \\
\texttt{fls} &= \lambda\texttt{t.}\ \lambda\texttt{f.}\ \texttt{f}
\end{aligned}
$$

$$
\begin{array}{cll}
 & \texttt{tru v w} & \\
= & \underline{(\lambda\texttt{t.}\lambda\texttt{f.t})\ \texttt{v}}\ \texttt{w} & \text{by definition} \\
\longrightarrow & \underline{(\lambda\texttt{f. v})\ \texttt{w}} & \text{reducing the underlined redex} \\
\longrightarrow & \texttt{v} & \text{reducing the underlined redex}
\end{array}
$$

$$
\begin{array}{cll}
 & \texttt{fls v w} & \\
= & \underline{(\lambda\texttt{t.}\lambda\texttt{f.f})\ \texttt{v}}\ \texttt{w} & \text{by definition} \\
\longrightarrow & \underline{(\lambda\texttt{f. f})\ \texttt{w}} & \text{reducing the underlined redex} \\
\longrightarrow & \texttt{w} & \text{reducing the underlined redex}
\end{array}
$$

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

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

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

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.

Example

$$
\begin{array}{lll}
 & \texttt{fst (pair v w)} & \\
= & \texttt{fst (}\underline{\texttt{(}\lambda\texttt{f. }\lambda\texttt{s. }\lambda\texttt{b. b f s) v}}\texttt{ w)} & \text{by definition} \\
\longrightarrow & \texttt{fst (}\underline{\texttt{(}\lambda\texttt{s. }\lambda\texttt{b. b v s) w}}\texttt{)} & \text{reducing} \\
\longrightarrow & \texttt{fst (}\lambda\texttt{b. b v w)} & \text{reducing} \\
= & \underline{\texttt{(}\lambda\texttt{p. p tru) (}\lambda\texttt{b. b v w)}} & \text{by definition} \\
\longrightarrow & \underline{\texttt{(}\lambda\texttt{b. b v w) tru}} & \text{reducing} \\
\longrightarrow & \texttt{tru v w} & \text{reducing} \\
\longrightarrow^{*} & \texttt{v} & \text{as before.}
\end{array}
$$

Church numerals

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

$$
\begin{aligned}
\mathtt{c}_0 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{z} \\
\mathtt{c}_1 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s}\ \mathtt{z} \\
\mathtt{c}_2 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z}) \\
\mathtt{c}_3 &= \lambda\mathtt{s}.\ \lambda\mathtt{z}.\ \mathtt{s}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{z}))
\end{aligned}
$$

That is, each number n is represented by a term $\mathtt{c}_n$ that takes two arguments, $\mathtt{s}$ and $\mathtt{z}$ (for "successor" and "zero"), and applies $\mathtt{s}$, n times, to $\mathtt{z}$.

Functions on Church Numerals

Successor:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

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

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

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

Multiplication:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

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

Multiplication:

$$\texttt{times = } \lambda\texttt{m. } \lambda\texttt{n. m (plus n) c}_0$$

Functions on Church Numerals

Successor:

$$\texttt{scc} = \lambda\texttt{n}.\ \lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{s (n s z)}$$

Addition:

$$\texttt{plus} = \lambda\texttt{m}.\ \lambda\texttt{n}.\ \lambda\texttt{s}.\ \lambda\texttt{z}.\ \texttt{m s (n s z)}$$

Multiplication:

$$\texttt{times} = \lambda\texttt{m}.\ \lambda\texttt{n}.\ \texttt{m (plus n)}\ \texttt{c}_0$$

Zero test:

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

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

Multiplication:

times = λm. λn. m (plus n) c_0

Zero test:

```
iszro = λm. m (λx. fls) tru
```

Functions on Church Numerals

Successor:

```
scc = λn. λs. λz. s (n s z)
```

Addition:

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

Multiplication:

times = λm. λn. m (plus n) c_0

Zero test:

```
iszro = λm. m (λx. fls) tru
```

What about predecessor?

Predecessor

```
zz = pair c0 c0

ss = λp. pair (snd p) (scc (snd p))

prd = λm. fst (m ss zz)
```

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

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

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...

Recursion in the Lambda-Calculus

Iterated Application

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

$$\texttt{Y}_f \quad = \quad (\lambda\texttt{x. f (x x)})\ (\lambda\texttt{x. f (x x)})$$

Iterated Application

Suppose `f` is some λ-abstraction, and consider the following term:

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

Now the "pattern of divergence" becomes more interesting:

$$\begin{array}{c}
\mathtt{Y}_f \\
= \\
\underline{(\lambda\mathtt{x.}\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))} \\
\longrightarrow \\
\mathtt{f}\ (\underline{(\lambda\mathtt{x.}\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))}) \\
\longrightarrow \\
\mathtt{f}\ (\mathtt{f}\ (\underline{(\lambda\mathtt{x.}\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))})) \\
\longrightarrow \\
\mathtt{f}\ (\mathtt{f}\ (\mathtt{f}\ (\underline{(\lambda\mathtt{x.}\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\mathtt{x}\ \mathtt{x}))}))) \\
\longrightarrow \\
\ldots
\end{array}$$

$\texttt{Y}_f$ 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:

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

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

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

Another delayed variant

Suppose `f` is a function. Define

$$\mathtt{Z}_f \quad = \quad \lambda\mathtt{y.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y}$$

This term combines the "added `f`" from $\mathtt{Y}_f$ with the "delayed divergence" of `omegav`.

If we now apply $\mathtt{Z}_f$ to an argument $\mathtt{v}$, something interesting happens:

$$\begin{array}{c}
\mathtt{Z}_f\ \mathtt{v} \\
= \\
\underline{(\lambda\mathtt{y.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y})\ \mathtt{v}} \\
\longrightarrow \\
\underline{(\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))}\ \mathtt{v} \\
\longrightarrow \\
\mathtt{f}\ (\lambda\mathtt{y.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y})\ \mathtt{v} \\
= \\
\mathtt{f}\ \mathtt{Z}_f\ \mathtt{v}
\end{array}$$

Since $\mathtt{Z}_f$ and $\mathtt{v}$ are both values, the next computation step will be the reduction of $\mathtt{f}\ \mathtt{Z}_f$ — that is, before we "diverge," $\mathtt{f}$ 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.).

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

$$
\begin{array}{c}
\texttt{Z}_f\ \texttt{3} \\
\longrightarrow^* \\
\texttt{f}\ \texttt{Z}_f\ \texttt{3} \\
= \\
(\lambda\texttt{fct.}\ \lambda\texttt{n.}\ \texttt{...})\ \texttt{Z}_f\ \texttt{3} \\
\longrightarrow\ \ \longrightarrow \\
\texttt{if 3=0 then 1 else 3 * (}\texttt{Z}_f\texttt{ (pred 3))} \\
\longrightarrow^* \\
\texttt{3 * (}\texttt{Z}_f\texttt{ (pred 3)))} \\
\longrightarrow \\
\texttt{3 * (}\texttt{Z}_f\texttt{ 2)} \\
\longrightarrow^* \\
\texttt{3 * (f }\texttt{Z}_f\texttt{ 2)} \\
\cdots
\end{array}
$$

A Generic Z

If we define

$$\mathtt{Z} \quad = \quad \lambda\mathtt{f.}\ \mathtt{Z}_f$$

i.e.,

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

then we can obtain the behavior of $\mathtt{Z}_f$ for any $\mathtt{f}$ we like, simply by applying $\mathtt{Z}$ to $\mathtt{f}$.

$$\mathtt{Z}\ \mathtt{f} \quad \longrightarrow \quad \mathtt{Z}_f$$

For example:

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

Technical Note

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

$$\mathtt{Z} =$$
$$\lambda\mathtt{f.}\ \lambda\mathtt{y.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ \mathtt{y}$$

$$\mathtt{fix} =$$
$$\lambda\mathtt{f.}\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x.}\ \mathtt{f}\ (\lambda\mathtt{y.}\ \mathtt{x}\ \mathtt{x}\ \mathtt{y}))$$

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