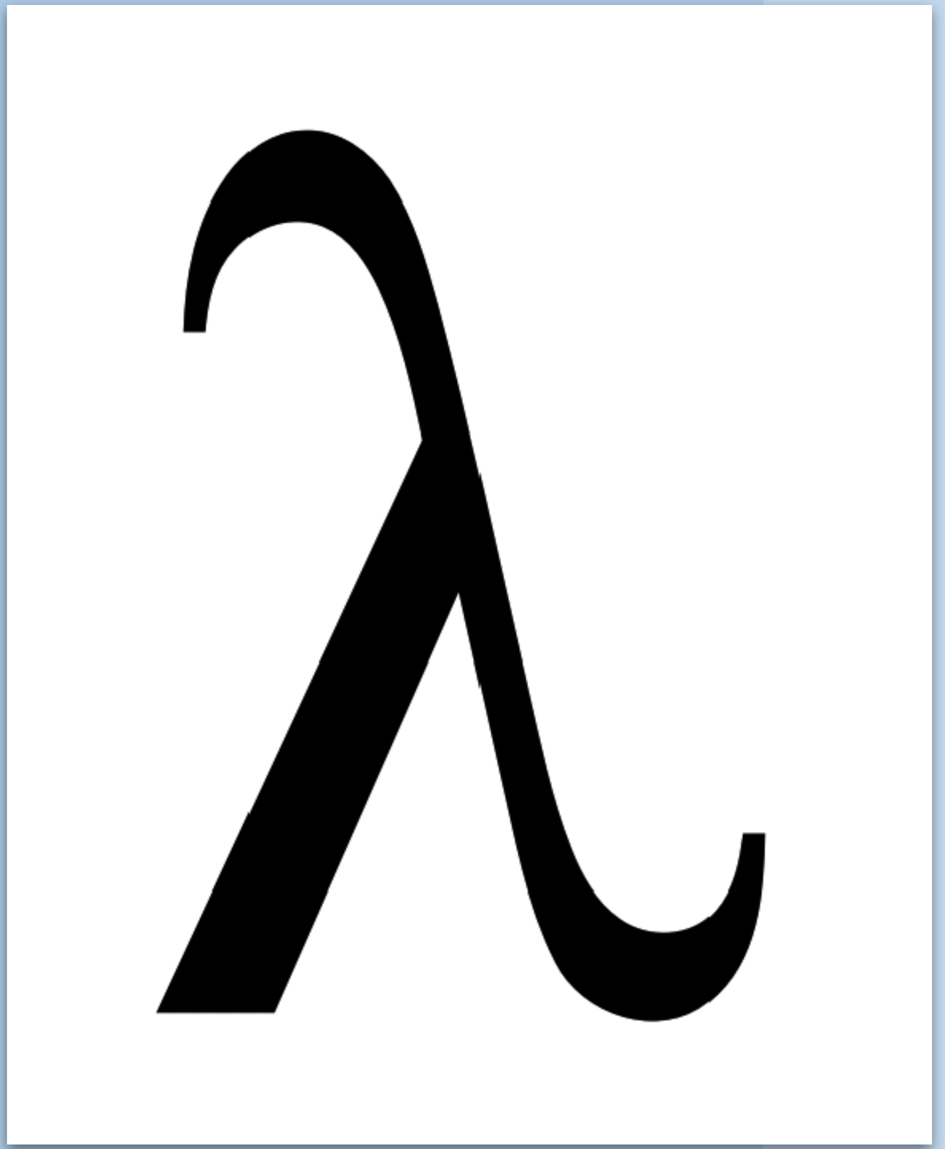
UNIVERSITÄT BERN

5. Introduction to the Lambda Calculus

Oscar Nierstrasz



Roadmap

> What is Computability? — Church's Thesis
> Lambda Calculus — operational semantics
> The Church-Rosser Property
> Modelling basic programming constructs

References

> Paul Hudak, *“Conception, Evolution, and Application of Functional Programming Languages,”* ACM Computing Surveys 21/3, Sept. 1989, pp 359-411.

> Kenneth C. Louden, *Programming Languages: Principles and Practice*, PWS Publishing (Boston), 1993.

> H.P. Barendregt, *The Lambda Calculus — Its Syntax and Semantics*, North-Holland, 1984, Revised edition.

Conception, Evolution, and Application of Functional Programming Languages

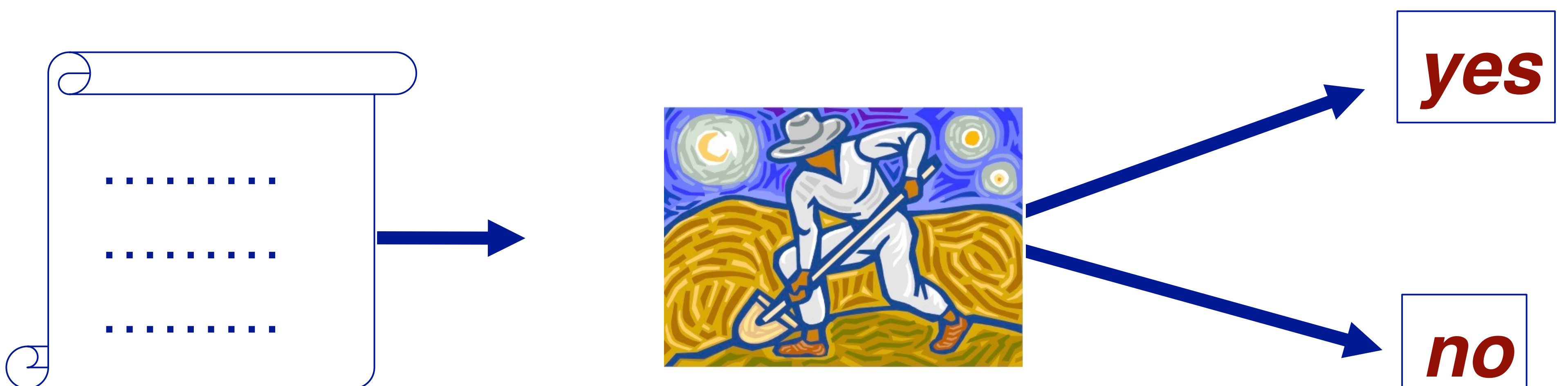
http://scgresources.unibe.ch/Literature/PL/Huda89a-p359-hudak.pdf

Roadmap

> **What is Computability? — Church's Thesis**
> Lambda Calculus — operational semantics
> The Church-Rosser Property
> Modelling basic programming constructs

What is Computable?

Computation is usually modelled as a *mapping from inputs to outputs*, carried out by a formal “machine,” or program, which processes its input in a *sequence of steps*.



input

Effectively computable function

output

An “effectively computable” function is one that can be computed in a *finite amount of time using finite resources*.

Church's Thesis

> *Effectively computable functions [from positive integers to positive integers] are just those definable in the lambda calculus.*

Or, equivalently:

> *It is not possible to build a machine that is more powerful than a Turing machine.*

Church's thesis cannot be proven because "effectively computable" is an *intuitive notion*, not a mathematical one. It can only be refuted by giving a counter-example — a machine that can solve a problem not computable by a Turing machine.

So far, all models of effectively computable functions have shown to be equivalent to Turing machines (or the lambda calculus).

Uncomputability

A problem that cannot be solved by any Turing machine in finite time (or any equivalent formalism) is called <u>uncomputable</u>.

Assuming Church's thesis is true, an uncomputable problem cannot be solved by any real computer.

The Halting Problem:

> *Given an arbitrary Turing machine and its input tape, will the machine eventually halt?*

The Halting Problem is *provably uncomputable* — which means that it cannot be solved in practice.

What is a Function? (I)

Extensional view:

A (total) <u>function</u> $f: A \rightarrow B$ is a subset of $A \times B$ (i.e., a *relation*) such that:

1. for each $a \in A$, there exists some $(a,b) \in f$ (i.e., $f(a)$ is *defined*), and

2. if $(a,b_1) \in f$ and $(a, b_2) \in f$, then $b_1 = b_2$ (i.e., $f(a)$ is *unique*)

The extensional view is the database view: a function is a particular *set* of mappings from arguments to values.

What is a Function? (II)

Intensional view:

A function $f: A \rightarrow B$ is an *abstraction* $\lambda x.e$, where x is a *variable name*, and e is an *expression*, such that when a value $a \in A$ is *substituted* for x in e, then this expression (i.e., f(a)) evaluates to some (unique) value $b \in B$.

The intensional view is the programmatic view: a function is a *specification* of how to transform the input argument to an output value.

NB: uniqueness does not come for free. The latter view is closer to that of programming languages, since infinite relations can only be represented intensionally.

Roadmap

> What is Computability? — Church's Thesis
> **Lambda Calculus — operational semantics**
> The Church-Rosser Property
> Modelling basic programming constructs

What is the Lambda Calculus?

The Lambda Calculus was invented by Alonzo Church [1932] as a mathematical formalism for expressing computation by functions.

Syntax:

e	::=	x	*a variable*
	|	λ x . e	*an abstraction (function)*
	|	e_1 e_2	*a (function) application*

Examples:

λ x . x — is a function taking an argument x, and returning x

f x — is a function f applied to an argument x

NB: *same as f(x) !*

We have seen lambda abstractions before in Haskell with a very similar syntax:

```
\ x -> x+1
```

is the anonymous Haskell function that adds 1 to its argument x.

Function application in Haskell also has the same syntax as in the lambda calculus:

```
Prelude> (\x ->x+1) 2
3
```

Parsing Lambda Expressions

Lambda extends as far as possible to the right

λ f.x y	≡	λ f.(x y)

Application is left-associative

x y z	≡	(x y) z

Multiple lambdas may be suppressed

λ f g.x	≡	λ f . λ g.x

What is the Lambda Calculus? ...

(Operational) Semantics:

α *conversion (renaming):*	λ x . e	$\leftrightarrow$	λ y . [y/x] e	*where y is not free in e*
β *reduction (application):*	$(\lambda$ x . $e_1)$ e_2	$\rightarrow$	[e_2/x] e_1	*avoiding name capture*
η *reduction:*	λ x . e x	$\rightarrow$	e	*if x is not free in e*

The lambda calculus can be viewed as the simplest possible pure functional programming language.

The ***α conversion*** rule simply states that "*variable names don't matter*". If you define a function with an argument *x*, you can change the name of *x* to *y*, as long as you do it consistently (change every *x* to *y*) and avoid name clashes (there must not be another [free] y in the same scope).

The ***β reduction*** rule shows *how to evaluate function application*: just (syntactically) replace the formal parameter of the function body by the argument everywhere, taking care to avoid name clashes.

Finally, the ***η reduction*** rule can be seen as a *wrapper removal optimization*: if the body of a function just applies another function *f* to its argument, then it is just a wrapper for *f*. We can remove the wrapper and replace that whole function by *f*.

Note that the α rule only rewrites an expression but does not simplify it. That is why it is called a "conversion" and not a "reduction".

Beta Reduction

Beta reduction is the *computational engine* of the lambda calculus:

Define: $\mathrm{I} \equiv \lambda\, x\,.\,x$

Now consider:

$$\mathrm{I}\,\mathrm{I} = (\lambda\, x\,.\,x)\,(\lambda\, x\,.\,x) \rightarrow [\lambda\, x\,.\,x / x]\, x \qquad \beta \textit{ reduction}$$

$$= \lambda\, x\,.\,x \qquad \textit{substitution}$$

$$= \mathrm{I}$$

In the expression:

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

we replace the x in the body of the first lambda by its argument. The body is simply x, so we end up with $(\lambda\, x\, .\, x\ )$

Let's number each x to make clear what is happening:

$(\lambda\, x_1\, .\, x_2)\ (\lambda\, x_3\, .\, x_4\ )$

x_1 and x_3 are formal parameters, and x_2 and x_4 are the bodies of the two lambda expressions. We are applying the first expression $(\lambda\, x_1\, .\, x_2)$ as a function to its argument $(\lambda\, x_3\, .\, x_4\ )$

To do this, we replace the body of $(\lambda\, x_1\, .\, x_2)$, i.e., x_2, by the argument $(\lambda\, x_3\, .\, x_4\ )$. This is written as follows:

$[\ (\lambda\, x_3\, .\, x_4\ )\ /\ x_2\ ]\ x_2$

This leaves as the end result: $(\lambda\, x_3\, .\, x_4\ )$ (i.e., $(\lambda\, x\, .\, x\ )$).

Lambda expressions in Haskell

We can implement many lambda expressions directly in Haskell:

```
Prelude> let i = \x -> x
Prelude> i 5
5
Prelude> i i 5
5
```

How is `i i 5` *parsed?*

Lambdas are anonymous functions

A lambda abstraction is just an *anonymous function.*

Consider the Haskell function:

```
compose f g x = f(g(x))
```

The *value* of `compose` is the anonymous lambda abstraction:

λ f g x . f (g x)

NB: This is the same as:

λ f . λ g . λ x . f (g x)

```
Prelude> let compose = \f g x -> f(g x)
Prelude> compose (\x->x+1) (\x->x*2) 5
11
```

Free and Bound Variables

The variable x is <u>bound</u> by λ in the expression: λ x.e

A variable that is not bound, is <u>free</u> :

$$\begin{aligned} fv(x) &= \{ x \} \\ fv(e_1\ e_2) &= fv(e_1) \cup fv(e_2) \\ fv(\lambda\ x\ .\ e) &= fv(e) - \{ x \} \end{aligned}$$

An expression with no free variables is <u>closed</u>.

(AKA a <u>combinator</u>.) Otherwise it is <u>open</u>.

For example, y is *bound* and x is *free* in the (open) expression:

λ y . x y

You can also think of *bound* variables as being *defined*. The expression

λ x.e

defines the variable x within the body e, just like:

```
int plus(int x, int y) { ... }
```

defines the variables `x` and `y` within the body of the Java method `plus`.

A variable that is not defined in some outer scope by some lambda is "*free*", or simply *undefined*.

Closed expressions have no "undefined" variables. In statically typed programming languages, all procedures and programs are normally closed.

A Few Examples

1. (λx.x) y
2. (λx.f x)
3. x y
4. (λx.x) (λx.x)
5. (λx.x y) z
6. (λx y.x) t f
7. (λx y z.z x y) a b (λx y.x)
8. (λf g.f g) (λx.x) (λx.x) z
9. (λx y.x y) y
10. (λx y.x y) (λx.x) (λx.x)
11. (λx y.x y) ((λx.x) (λx.x))

Which variables are free?
Which are bound?

“Hello World” in the Lambda Calculus

```
hello world
```

✎ *Is this expression open? Closed?*

Roadmap

> What is Computability? — Church's Thesis
> Lambda Calculus — operational semantics
> **The Church-Rosser Property**
> Modelling basic programming constructs

Why macro expansion is wrong

Syntactic substitution will not work:

(λ x y . x y) y	$\rightarrow$	[y / x] (λ y . x y)	β *reduction*
	$\neq$	(λ y . y y)	*incorrect substitution!*

Since y is *already bound* in (λ y . x y), we cannot directly substitute y for x.

Substitution

We must define substitution carefully to avoid *name capture:*

$$
\begin{array}{rcll}
[e/x]\ x & = & e & \\
[e/x]\ y & = & y & \textit{if } x \neq y \\
[e/x]\ (e_1\ e_2) & = & ([e/x]\ e_1)\ ([e/x]\ e_2) & \\
[e/x]\ (\lambda\ x\ .\ e_1) & = & (\lambda\ x\ .\ e_1) & \\
[e/x]\ (\lambda\ y\ .\ e_1) & = & (\lambda\ y\ .\ [e/x]\ e_1) & \textit{if } x \neq y \textit{ and } y \notin fv(e) \\
[e/x]\ (\lambda\ y\ .\ e_1) & = & (\lambda\ z.\ [e/x]\ [z/y]\ e_1) & \textit{if } x \neq y \textit{ and} \\
 & & & z \notin fv(e) \cup fv(e_1)
\end{array}
$$

Consider:

$$(\lambda\ \mathbf{x}\ .\ ((\lambda\ y\ .\ \mathbf{x})\ (\lambda\ x\ .\ x))\ \mathbf{x}\ )\ y \rightarrow [y\ /\ x]\ ((\lambda\ y\ .\ \mathbf{x})\ (\lambda\ x\ .\ x))\ \mathbf{x}$$

$$= \quad ((\lambda\ \mathbf{z}\ .\ y\ )\ (\lambda\ x\ .\ x))\ y$$

Of these six cases, only the last one is tricky.

If the expression e (i.e., the argument to our function (λ y . e_1)) contains a variable name y that conflicts with the formal parameter y of our function, then we must first rename y to a fresh name z in that function. After renaming y to z, there is no longer any conflict with the name y in our argument e, and we can proceed safely with the substitution.

Alpha Conversion

Alpha conversions allow us to rename bound variables.

A bound name x in the lambda abstraction (λ x.e) may be substituted by any other name y, as long as there are *no free occurrences of y in e:*

Consider:

(λ x y . x y) y	$\rightarrow$ (λ x z . x z) y	α *conversion*
	$\rightarrow$ [y / x] (λ z . x z)	β *reduction*
	$\rightarrow$ (λ z . y z)	
	= y	η *reduction*

Eta Reduction

Eta reductions allow one to remove “wrappers.”

Suppose that f is *closed* (i.e., there are no free variables in f).

Then:

$(\lambda\ x\ .\ f\ x\ )\ y \quad \rightarrow \quad f\ y$ $\qquad\qquad \beta$ *reduction*

So, $(\lambda\ x\ .\ f\ x\ )$ is just a wrapper around f, and behaves the same as f !

Eta reduction says, *whenever x does not occur free in f*, we can rewrite $(\lambda\ x\ .\ f\ x\ )$ as f.

αβη

	(λ x y . x y) (λ x . x y) (λ a b . a b)	*NB: left assoc.*
→	(λ x z . x z) (λ x . x y) (λ a b . a b)	α *conversion*
→	(λ z . (λ x . x y) z) (λ a b . a b)	β *reduction*
→	(λ x . x y) (λ a b . a b)	β *reduction*
→	(λ a b . a b) y	β *reduction*
→	(λ b . y b)	β *reduction*
→	y	η *reduction*

Normal Forms

A lambda expression is in <u>normal form</u> *if it can no longer be reduced by beta or eta reduction rules.*

Not all lambda expressions have normal forms!

$\Omega = (\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)$

$\rightarrow$	$[\ (\lambda\ x\ .\ x\ x)\ /\ x\ ]\ (\ x\ x\ )$	
$=$	$(\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)$	β *reduction*
$\rightarrow$	$(\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)$	β *reduction*
$\rightarrow$	$(\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)$	β *reduction*
$\rightarrow$	...	

Reduction of a lambda expression to a normal form is analogous to a *Turing machine halting* or a *program terminating.*

A Few Examples

1. $(\lambda x.x)\ y$
2. $(\lambda x.f\ x)$
3. $x\ y$
4. $(\lambda x.x)\ (\lambda x.x)$
5. $(\lambda x.x\ y)\ z$
6. $(\lambda x\ y.x)\ t\ f$
7. $(\lambda x\ y\ z.z\ x\ y)\ a\ b\ (\lambda x\ y.x)$
8. $(\lambda f\ g.f\ g)\ (\lambda x.x)\ (\lambda x.x)\ z$
9. $(\lambda x\ y.x\ y)\ y$
10. $(\lambda x\ y.x\ y)\ (\lambda x.x)\ (\lambda x.x)$
11. $(\lambda x\ y.x\ y)\ ((\lambda x.x)\ (\lambda x.x))$

Are these in normal form?
Can they be reduced?
If so, how?

Evaluation Order

Most programming languages are strict, that is, *all expressions passed to a function call are evaluated before control is passed to the function.*

Most modern functional languages, on the other hand, use lazy evaluation, that is, *expressions are only evaluated when they are needed.*

Consider:

```
sqr n = n * n
```

Applicative-order reduction:

```
sqr (2+5) ➪ sqr 7 ➪ 7*7 ➪ 49
```

Normal-order reduction:

```
sqr (2+5) ➪ (2+5) * (2+5) ➪ 7 * (2+5) ➪ 7 * 7 ➪ 49
```

The Church-Rosser Property

"If an expression can be evaluated at all, it can be evaluated by consistently using normal-order evaluation. If an expression can be evaluated in several different orders (mixing normal-order and applicative order reduction), then all of these evaluation orders yield the same result."

So, evaluation order "does not matter" in the lambda calculus.

Roadmap

> What is Computability? — Church's Thesis
> Lambda Calculus — operational semantics
> The Church-Rosser Property
> **Modelling basic programming constructs**

Non-termination

However, applicative order reduction may not terminate, even if a normal form exists!

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

Applicative order reduction	***Normal order reduction***
$\rightarrow (\lambda\ x\ .\ y)\ (\ (\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)\ )$	$\rightarrow y$
$\rightarrow (\lambda\ x\ .\ y)\ (\ (\lambda\ x\ .\ x\ x)\ (\lambda\ x\ .\ x\ x)\ )$	
…	

Compare to the Haskell expression:

```
(\x -> \y -> x) 1 (5/0) ⇨ 1
```

Currying

Since a lambda abstraction only binds a single variable, functions with multiple parameters must be modelled as Curried higher-order functions.

As we have seen, to improve readability, multiple lambdas are suppressed, so:

$$\lambda\, x\, y\, .\, x \quad = \quad \lambda\, x\, .\, \lambda\, y\, .\, x$$

$$\lambda\, b\, x\, y\, .\, b\, x\, y \quad = \quad \lambda\, b\, .\, \lambda\, x\, .\, \lambda\, y\, .\, (\, b\, x\,)\, y$$

Don't forget that functions written this way are still Curried, so arguments can be bound one at a time!

In Haskell:

```
Prelude> let f = (\ x y -> x) 1
Prelude> f 2
1
```

Representing Booleans

Many programming concepts can be directly expressed in the lambda calculus. Let us define:

$$\begin{aligned}
\text{True} &\equiv \lambda\, x\, y\, .\, x \\
\text{False} &\equiv \lambda\, x\, y\, .\, y \\
\text{not} &\equiv \lambda\, b\, .\, b\ \text{False}\ \text{True} \\
\text{if } b \text{ then } x \text{ else } y &\equiv \lambda\, b\, x\, y\, .\, b\, x\, y
\end{aligned}$$

then:

$$\begin{aligned}
\text{not True} &= (\lambda\, b\, .\, b\ \text{False}\ \text{True}\,)\,(\lambda\, x\, y\, .\, x\,) \\
&\rightarrow (\lambda\, x\, y\, .\, x\,)\ \text{False}\ \text{True} \\
&\rightarrow \text{False} \\
\text{if True then } x \text{ else } y &= (\lambda\, b\, x\, y\, .\, b\, x\, y\,)\,(\lambda\, x\, y\, .\, x)\, x\, y \\
&\rightarrow (\lambda\, x\, y\, .\, x)\, x\, y \\
&\rightarrow x
\end{aligned}$$

This is the “standard encoding” of Booleans as lambdas (other encodings are possible).

A Boolean makes a choice between two values, a “true” one and a “false” one. `True` returns the first argument and `False` returns the second.

Negation just reverses the logic, by passing `False` and `True` as arguments to the boolean: `not True` will return `False` and `not False` will return `True`.

Representing Tuples

Although tuples are not supported by the lambda calculus, they can easily be modelled as higher-order functions that “wrap” pairs of values.

n-tuples can be modelled by composing pairs ...

Define:	pair	≡	(λ x y z . z x y)
	first	≡	(λ p . p True)
	second	≡	(λ p . p False)
then:	(1, 2)	=	pair 1 2
		→	(λ z . z 1 2)
	first (pair 1 2)	→	(pair 1 2) True
		→	True 1 2
		→	1

The function *pair* takes three arguments. The first two arguments are the x and y values of the pair. Since *pair* is a Curried function, passing in x and y returns a function (i.e., a pair) that will take a third argument, z. The body of the pair will pass x and y to z, which can then bind x and y and do what it likes with them.

As examples, consider the functions first and second. Each takes a pair p as argument and and passes it a boolean as the final argument z. These booleans respectively return x or y, i.e., the first or second value in the pair.

How would you define a lambda expression sum *that takes a pair p as argument and returns the sum of the x and y values it contains?*

Tuples as functions

In Haskell:

```haskell
t      = \x -> \y -> x
f      = \x -> \y -> y
pair   = \x -> \y -> \z -> z x y
first  = \p -> p t
second = \p -> p f
```

```
Prelude> first (pair 1 2)
1
Prelude> first (second (pair 1 (pair 2 3)))
2
```

What you should know!

- ✎ *Is it possible to write a Pascal compiler that will generate code just for programs that terminate?*
- ✎ *What are the alpha, beta and eta conversion rules?*
- ✎ *What is name capture? How does the lambda calculus avoid it?*
- ✎ *What is a normal form? How does one reach it?*
- ✎ *What are normal and applicative order evaluation?*
- ✎ *Why is normal order evaluation called lazy?*
- ✎ *How can Booleans and tuples be represented in the lambda calculus?*

Can you answer these questions?

- ✎ *How can name capture occur in a programming language?*
- ✎ *What happens if you try to program Ω in Haskell? Why?*
- ✎ *What do you get when you try to evaluate (pred 0)? What does this mean?*
- ✎ *How would you model numbers in the lambda calculus? Fractions?*