CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Lambda Calculus

Introduction to λ-calculus

• A notation for describing computations that focuses on **function** definition and application.

• Allows convenient definition and manipulation of functions as "first-class" **values**.

Example: a function that adds 1 to its argument

`(λx.(x + 1))`

• Invented by Church in the 1930's to study computability questions, but now primarily of interest in programming language community.

• In itself, serves as a basic functional language.

Practical FLs = λ-calculus + "syntactic sugar."

• Can also serve (in restricted or specialized form) as **intermediate language** in compilers, e.g.,

OO Language $\Longrightarrow$ λ-calculus + special constructs

(• Meta-notation for denotational semantics.)

• We will take a pragmatic, rather than theoretical, approach to λ-calculus. In particular, we will treat the calculus as a syntactic game, and won't worry about **models**.

Syntax

λ-calculus is an **expression** calculus, with the following **concrete syntax**:

`<exp>`	`:=`	`<constant>`	Built-in constants
	`|`	`<variable>`	Variable names
	`|`	`(<exp> <exp>)`	Function applications
	`|`	`(λ<variable>.<exp>)`	Function abstractions

A `<variable>` is ordinarily denoted by a lower-case letter, e.g., `x,y,z,f`.

We typically use upper-case letters (e.g., `M,N,P,Q`) as metavariables ranging over expressions (e.g., `(λx.M)`).

Pure λ-calculus has no constants. We will typically extend the λ-calculus with some set of "built-in" constants and functions (an **applied** λ-calculus):

`0,1,2,...`	Integer constants
`+-*/`	Integer operators
`true,false`	Boolean constants
`and,or,not`	Boolean operators
`if`	If-then-else operator
`pair`	Pair constructor
`fst, snd`	Pair selectors
`cons, nil`	List constructors
`hd, tl`	List selectors
`null`	List empty predicate

Examples:

```
(f a)
(+ 3 2)
λx.(+ x 1)
(λx.(+ x 1)) 17
λy.y
λy.pair (snd y) (fst y)
λf.λx.pair (f (fst x)) (f (snd x))
λx.if (null x) 0 (hd x)
```

We often cheat and use several **abbreviations** in our concrete syntax:

- Function application associates to the left.
- The body of a λ-abstraction extends as far to the right as possible.
- Drop parentheses where not needed.
- A sequence of consecutive abstractions can be written with a single λ.

Example:

S $\equiv$ `λxyz.(x z)(y z)`

could have been written with fewer parentheses as

`λxyz.x z(y z)`

Its full form is

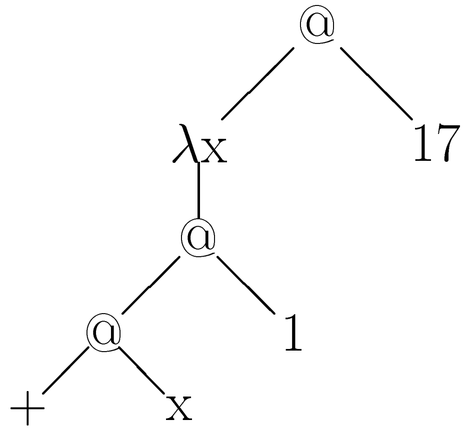
`(λx.(λy.(λz.((x z)(y z)))))`

Tree representation

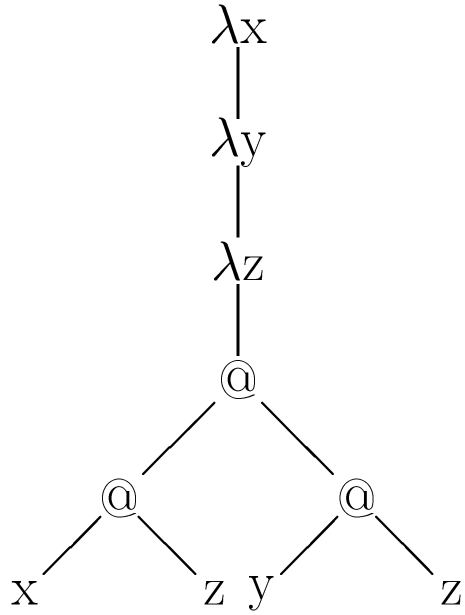
For expression of any size, it is best to write down the **abstract syntax tree** of the expression to avoid confusion. (This will prove more deeply useful later on.)

Examples:

`(λx.(+ x 1)) 17`



`(λx.(λy.(λz.((x z)(y z)))))`



More on Functions

λ-abstractions are fundamentally the same thing as `fun` or `fn` expressions in ML; e.g., λx.+ x 1 corresponds to

(CAML)	`fun x -> x + 1`
(SML)	`fn x => x + 1`

Thus we can view λ-calculus expressions as a subset of ML expressions. (Of course, we haven't said anything yet about how λ expressions are supposed to behave computationally.)

Notice that all λ-calculus functions are written in prefix notation and take exactly **one** argument. What do we do about functions that naturally require more than one, such as addition?

Recall that when we wrote `(+ 3 4)` this was really an abbreviation for `((+ 3) 4)`. That is, `+` is a function that takes a single integer argument `x` and returns a **new** function whose effect is to add `x` to *its* argument!

This trick is called **currying** after the logician Haskell Curry, who made extensive use of it.

Alternatively, we could define the built-in `+` function to take a single **pair** of integers as argument. This is the approach taken in ML. (But the built-in `pair` constructor must still somehow take *two* arguments.)

Reductions

We **evaluate** a λ-calculus expression by repeatedly selecting a **reducible (sub-)expression (redex)** and reducing it, according to a **reduction rule**.

Pure λ-calculus has the **β-rule**, which describes how to **apply** a λ-abstraction to an argument.

Informally: the result of applying a λ-abstraction to an argument is an instance of the body of the abstraction in which (free) occurrences of the formal parameter in the body are replaced with (copies of) the argument.

Each applied λ-calculus also has **δ-rules** describing how to reduce expressions involving the built-in functions and constants.

Examples:

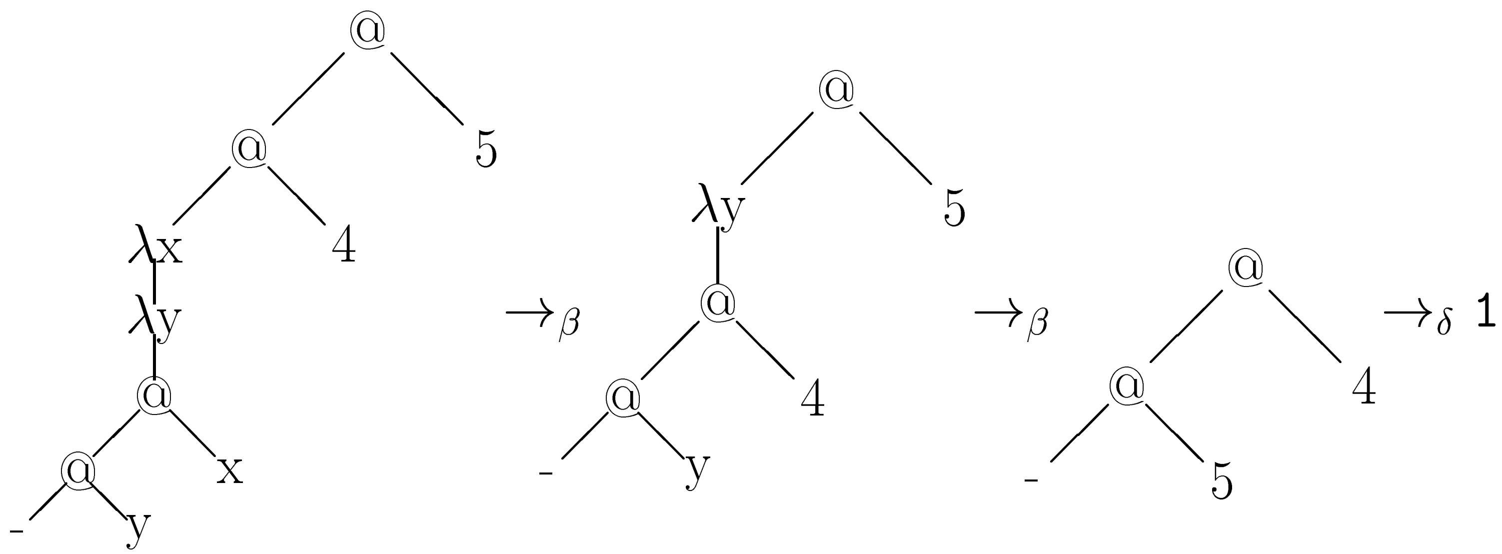
$\texttt{* 2 3} \rightarrow_\delta \texttt{6}$

$\texttt{(}\lambda\texttt{x.+ x 1) 4} \rightarrow_\beta \texttt{+ 4 1} \rightarrow_\delta \texttt{5}$

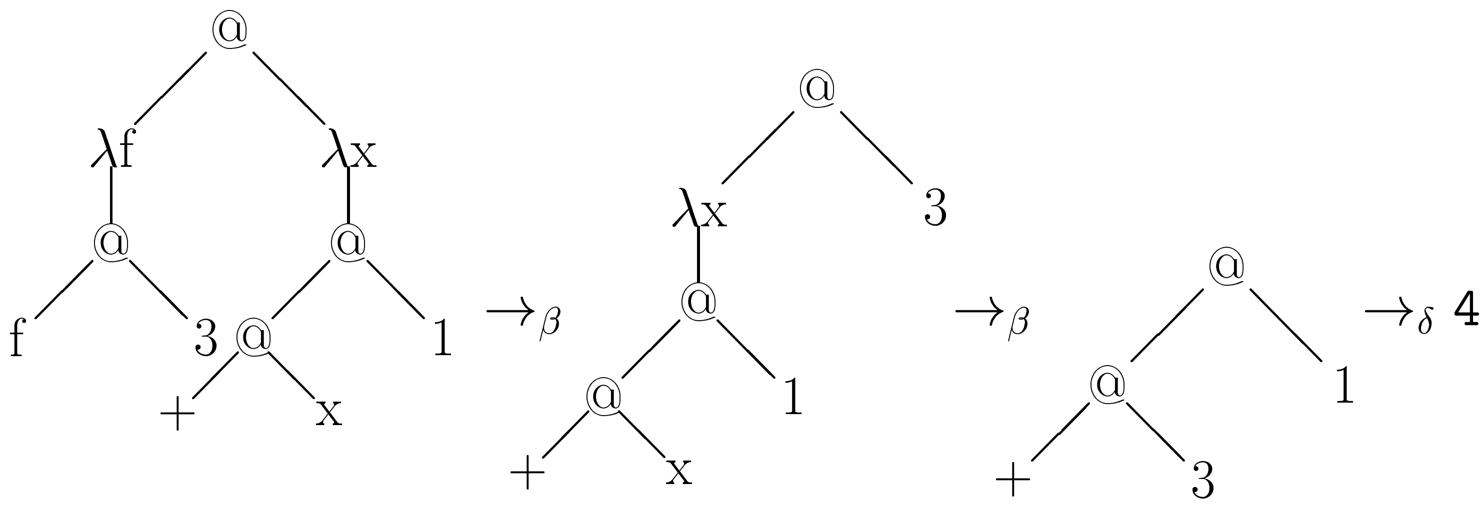
$\texttt{(}\lambda\texttt{x.+ x x) 5} \rightarrow_\beta \texttt{+ 5 5} \rightarrow_\delta \texttt{10}$

$\texttt{(}\lambda\texttt{x.3) 5} \rightarrow_\beta \texttt{3}$

$$
\begin{aligned}
\texttt{(}\lambda\texttt{x.(}\lambda\texttt{y.- y x)) 4 5} &\rightarrow_\beta \texttt{(}\lambda\texttt{ y.- y 4) 5}\\
&\rightarrow_\beta \texttt{- 5 4}\\
&\rightarrow_\delta \texttt{1}
\end{aligned}
$$



$$
\begin{aligned}
\texttt{(}\lambda\texttt{f.f 3)(}\lambda\texttt{x.+ x 1)} &\rightarrow_\beta \texttt{(}\lambda\texttt{x.+ x 1) 3}\\
&\rightarrow_\beta \texttt{+ 3 1}\\
&\rightarrow_\delta \texttt{4}
\end{aligned}
$$



Bound and Free Variables

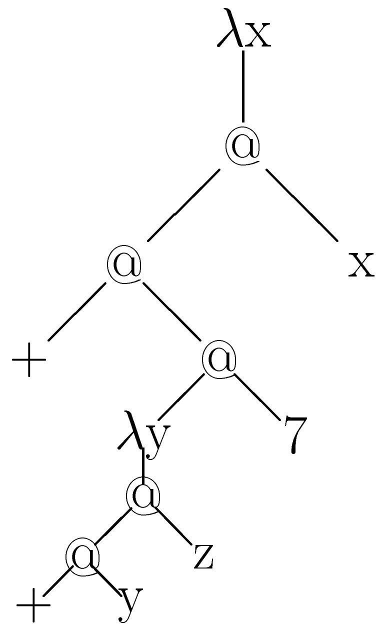
In an abstraction λ`x.M`, we say the λ **binds** the variable `x` in the **body** `M`, meaning that `x` is just a placeholder for a value that will be filled in when the abstraction is applied.

An **occurrence** of a variable in an expression is **bound** if there is an enclosing λ abstraction that binds it; otherwise it is **free**.

Note that these definitions are relative to a particular occurrence and a particular expression.

Example

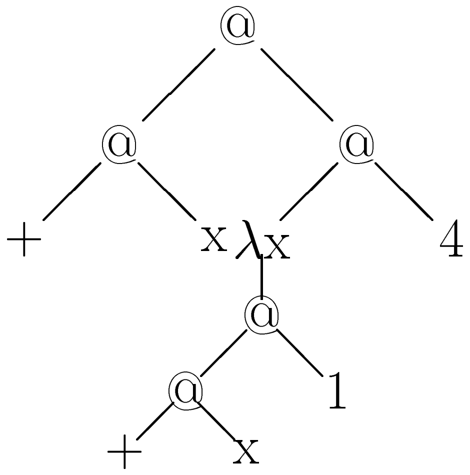
λ`x.+ ((`λ`y.+ y z) 7) x`



Here `x` and `y` occur bound in the overall expression, but `z` occurs free.

Example

`+ x ((λx.+ x 1) 4)`



Here the first occurrence of `x` is free but the second is bound.

If `M` is an expression we write:

$FV($`M`$)$	set of free variables in `M`
$BV($`M`$)$	set of bound variables in `M`

An expression `M` such that $FV(\mathtt{M}) = \emptyset$ is said to be **closed**.

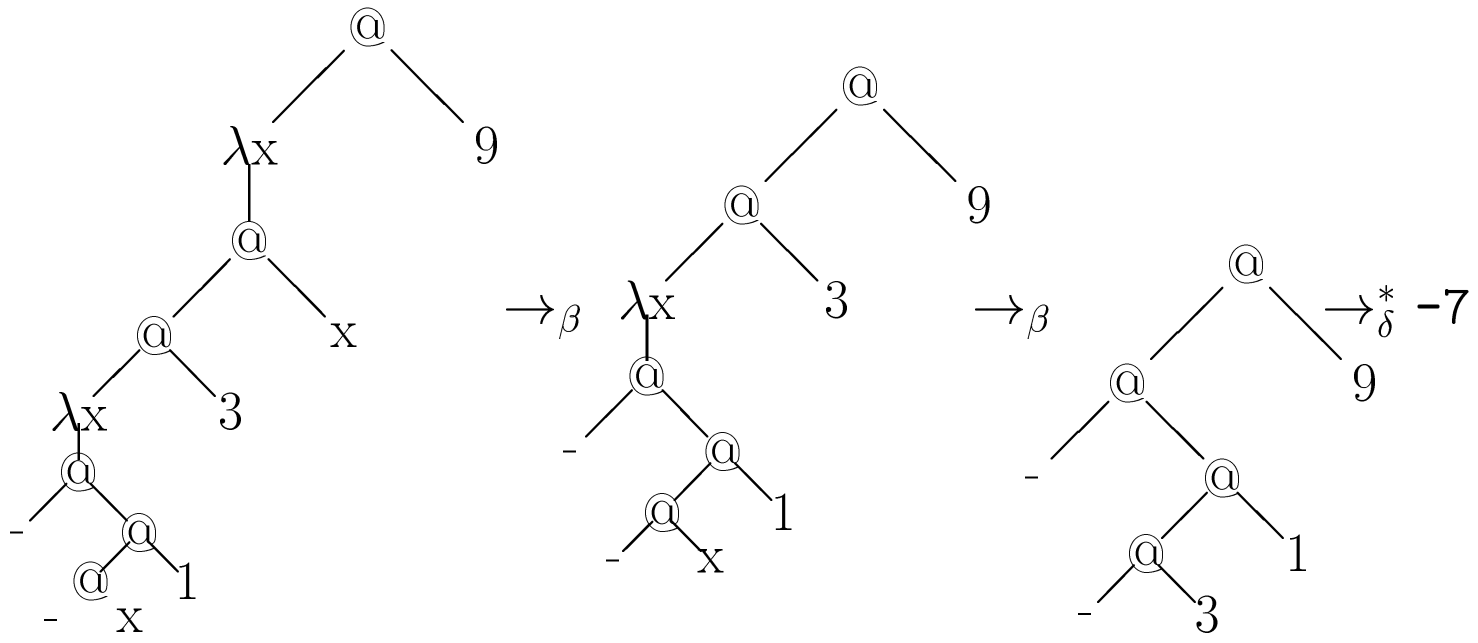
β-reduction, more carefully

If the same name appears in several λ-bindings, we want to use idea of nested scopes from block-structured languages.

I.e., when doing β-reduction, we should only substitute for **free** occurrences of the formal parameter variable.

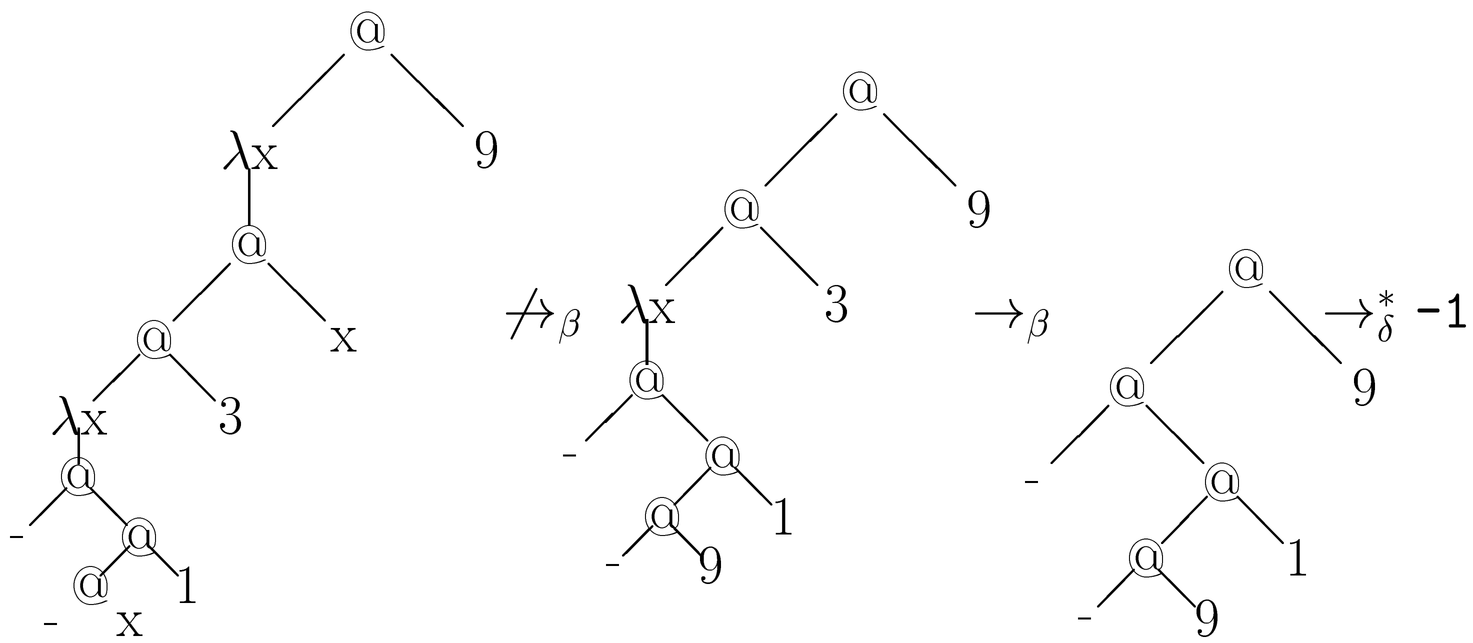
Example

$$
\begin{array}{ll}
 & \texttt{(}\lambda\texttt{x.(}\lambda\texttt{x.- (- x 1)) 3 x) 9} \\
\rightarrow_\beta & \texttt{(}\lambda\texttt{ x.- (- x 1)) 3 9} \\
\rightarrow_\beta & \texttt{- (- 3 1) 9} \\
\rightarrow_\delta^* & \texttt{-7}
\end{array}
$$



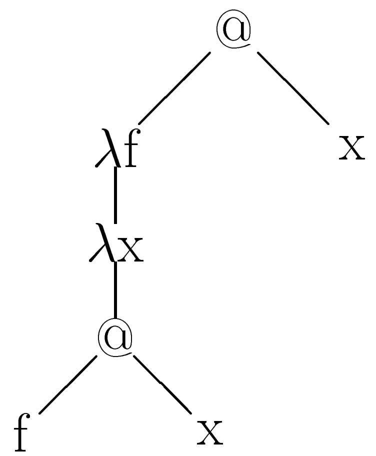
The following is **wrong!**

$$
\begin{array}{ll}
 & \texttt{(}\lambda\texttt{x.(}\lambda\texttt{x.- (- x 1)) 3 x) 9} \\
\not\rightarrow_\beta & \texttt{(}\lambda\texttt{x.- (- 9 1)) 3 9} \\
\rightarrow_\beta & \texttt{- (- 9 1) 9} \\
\rightarrow_\delta^* & \texttt{-1}
\end{array}
$$

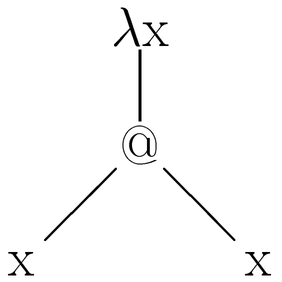


Variable Capture

Our revised definition of β-reduction still isn't adequate; here's another name-clash problem.



Naive substitution leads to



but this is obviously wrong: the right-hand `x`, originally free in the expression, has been "captured" by the λ`x`.

We must conclude that a β-reduction of `(`λ`x.M)N` is only valid if FV(`N`) do not clash with any formal parameters in `M`.

We will see several ways to cope with this problem.

α-Conversion and β-Conversion

We can avoid name-clash problems by **uniformly** changing the names of bound variables where necessary.

Intuitively, changing names is justifiable because they are only formal parameters.

Formally, we call such name changes **α-conversion**.

Example

$(\lambda\texttt{x.x + 1}) \leftrightarrow_\alpha (\lambda\texttt{y.y + 1})$

Of course, the new name must not appear free in the body of the abstraction.

$(\lambda\texttt{x.x + y}) \not\leftrightarrow_\alpha (\lambda\texttt{y.y + y})$

In general, we'd like to define two expressions as **interconvertible** if they "mean the same thing" (intuitively).

To do this, we must define two other forms of conversion relation.

β-conversion is the symmetric closure of β-reduction (the inverse operation is **β-abstraction**).

Example

$\texttt{+ 4 1} \leftrightarrow_\beta (\lambda\texttt{x.+ x 1) 4}$

Substitution

We can avoid all name-clash problems and give simple definitions of both α- and β-conversion via a careful definition of **capture-avoiding substitution**.

We write `[M/x] N` for the substitution of `M` for `x` in `N`. (Other similar notations are also in common use, e.g., `N [M/x]`.)

$$
\begin{array}{lcll}
\texttt{[M/x] c} & = & \texttt{c} & \\
\texttt{[M/x] x} & = & \texttt{M} & \\
\texttt{[M/x] y} & = & \texttt{y} & (\texttt{y} \neq \texttt{x}) \\
\texttt{[M/x] (Y Z)} & = & \texttt{([M/x] Y)([M/x] Z)} & \\
\texttt{[M/x] }\lambda\texttt{x.Y} & = & \lambda\texttt{x.Y} & \\
\texttt{[M/x] }\lambda\texttt{y.Z} & = & \lambda\texttt{y.[M/x]Z} & (\texttt{x} \notin FV(\texttt{Z}) \\
 & & & \texttt{or y} \notin FV(\texttt{M})) \\
\texttt{[M/x] }\lambda\texttt{y.Z} & = & \lambda\texttt{w.[M/x]([w/y]Z)} & (\texttt{w} \notin FV(\texttt{Z}) \\
 & & & \quad \cup FV(\texttt{M}))
\end{array}
$$

Then we can define

$$
\begin{array}{lll}
\lambda\texttt{x.Z} & \leftrightarrow_{\alpha} & \lambda\texttt{y.[y/x]Z} \ (\texttt{y} \notin FV(\texttt{Z})) \\
(\lambda\texttt{x.M})\texttt{ N} & \leftrightarrow_{\beta} & \texttt{[N/x]M}
\end{array}
$$

de Bruijn Notation

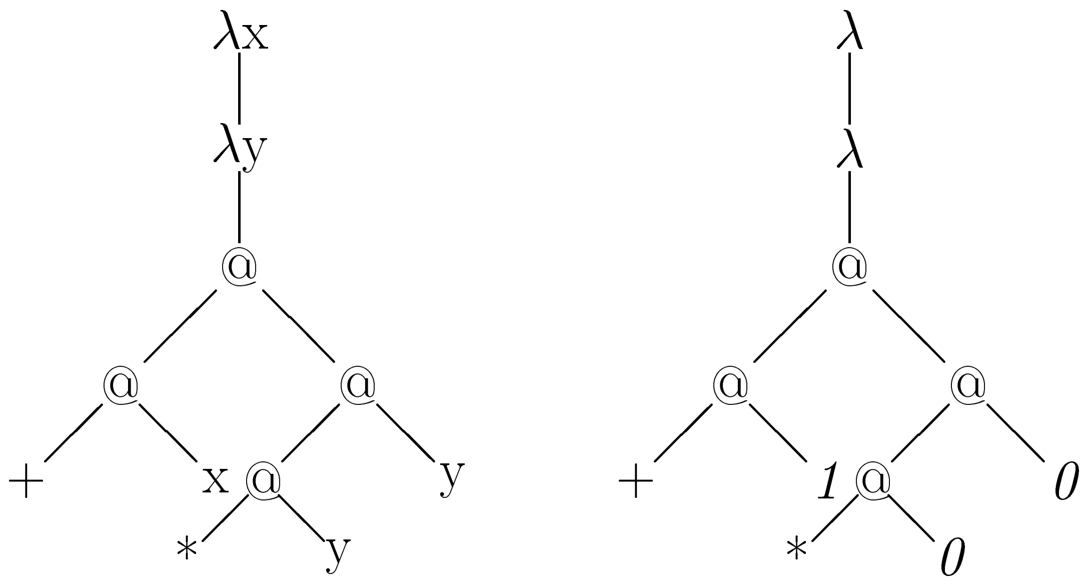
Another way to solve name clash problems is to do away with names altogether!

In the tree for a closed expression, every variable occurrence names a variable bound at some λ-binding on the path between the occurrence node and the root.

We can uniquely identify a variable by counting the number of λ-bindings that intervene between its mention and its defining binding.

Example

λ`x.`λ`y.+ x (* y y)` λ`.`λ`.+` *1* `(*` *0* *0*`)`



It is now fairly straightforward to define β-reduction in terms of this representation.

(Since there are no names, we don't need α-conversion!)

η-Reduction and η-Conversion

Consider these two expressions:

`λx.+ 1 x`

`(+ 1)`

They "mean the same thing" in the sense that they behave the same way when applied to any argument (i.e., they add `1` to it). This is the concept of **functional extensionality**.

To formalize this, we add an η-reduction rule:

$(\lambda \mathtt{x}.\mathtt{F}\ \mathtt{x}) \rightarrow_\eta \mathtt{F}$

provided $\mathtt{x} \notin FV(\mathtt{F})$.

η-reduction tends to simplify an expression, but its inverse, η-expansion can also be useful. Their combination is called **η-conversion**, written $\leftrightarrow_\eta$.

The η-rules cannot be deduced from the β-rules.

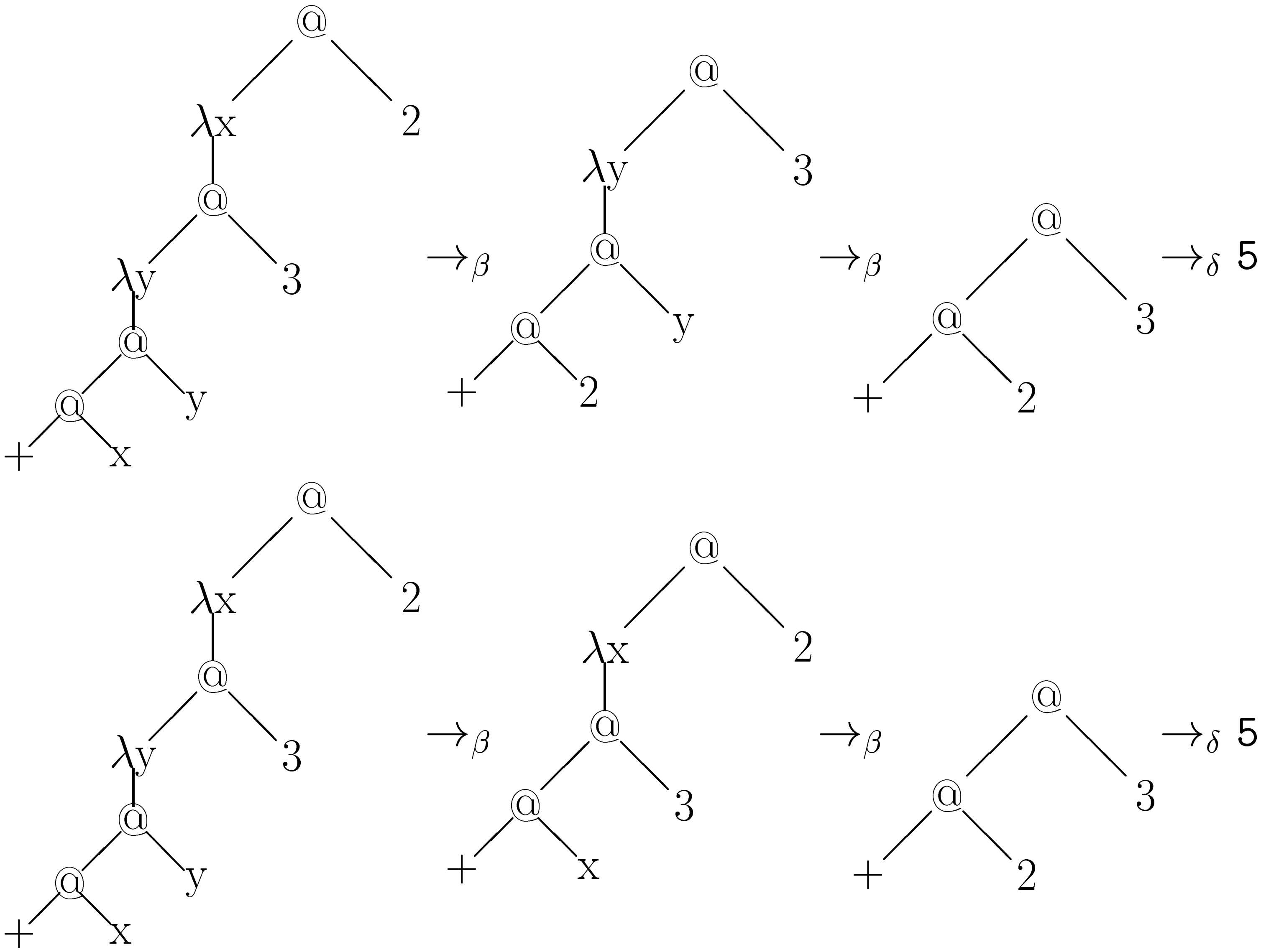
We will generally appeal to the η-rules to justify program transformations, rather than to perform computation.

Reduction Order

To evaluate a λ-expression, we keep performing reductions as long as a redex exists. An expression containing no redexes is said to be in **normal form**.

But an expression may contain more than one redex, so evaluation may proceed by different routes.

Example: `(λx.(λy.+ x y) 3) 2` can be reduced in two ways:

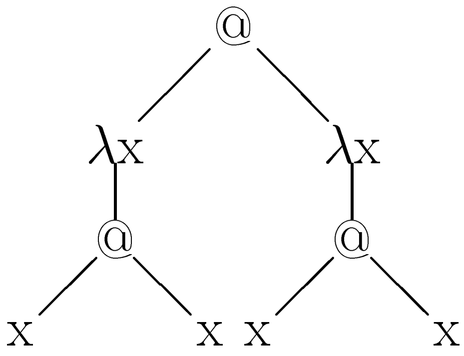


Fortunately, both reduction orders lead to the same result in this example. Will this always happen?

Non-terminating reductions

Not every expression has a normal form at all.

Consider `(D D)`, where $\texttt{D} \equiv \lambda\texttt{x.x x}$.



The evaluation of this expression never terminates because `(D D)` $\rightarrow_\beta$ `(D D)`! In other words, evaluation enters an infinite loop. Such a computation is said to **diverge**.

For some expressions, one reduction sequence may terminate while another does not.

Example: $(\lambda$ `x.3)(D D)`

If we perform the application of $(\lambda$ `x.3)` to `(D D)` first, we immediately get `3` and stop. But if we keep trying to evaluate the argument `(D D)` first, we keep getting `(D D)` again.

So choice of reduction order can affect whether we get **any** answer!

Church-Rosser Theorem

It turns out that all reduction sequences that terminate **will** give the same result, at least for the pure λ-calculus. Thus, if an expression has any normal form, that normal form is unique.

This happy result is a consequence of the **Church-Rosser Theorem**: If $\mathtt{E}_1 \leftrightarrow^* \mathtt{E}_2$ then $\exists$ $\mathtt{E}$, such that $\mathtt{E}_1 \rightarrow^* \mathtt{E}$ and $\mathtt{E}_2 \rightarrow^* \mathtt{E}$.

(Any system (not necessarily related to the λ-calculus) for which this theorem is true is said to be "Church-Rosser.")

Corollary Suppose now that $\mathtt{E}_0 \rightarrow^* \mathtt{E}_1$ and $\mathtt{E}_0 \rightarrow^* \mathtt{E}_2$ and that both $\mathtt{E}_1$ and $\mathtt{E}_2$ are in normal form. Then, since $\mathtt{E}_1 \leftrightarrow^* \mathtt{E}_2$, C-R says $\exists$ $\mathtt{E}$ such that $\mathtt{E}_1 \rightarrow^* \mathtt{E}$ and $\mathtt{E}_2 \rightarrow^* \mathtt{E}$. But since $\mathtt{E}_1$ and $\mathtt{E}_2$ are already both in normal form, this can only mean that $\mathtt{E}_1 = \mathtt{E} = \mathtt{E}_2$.

(If we are using names, the normal form is unique "up to α-conversion".)

The Church-Rosser theorem for λ-calculus is surprisingly difficult to prove.

(It isn't necessarily true for the extensions of the λ-calculus that we will use, but this is not a problem in practice.)

Specifying Reduction Order

No reduction sequence can give us a wrong answer, but some sequences might fail to terminate even when a normal form exists.

To define reduction orders, we need to characterize **redex positions** in an expression.

An **outermost** redex of an expression is one that is not contained within any other redex (i.e., within either the argument or the function body).

An **innermost** redex of an expression is one that contains no other redex.

The **leftmost outermost** redex is the outermost redex whose λ (for a β-redex) or function constant (for a δ-redex) is furthest to the left (in the string or tree representation). **Leftmost innermost** is defined similarly.

Two important reduction **strategies** are:

• **applicative order**, which says to always reduce the leftmost innermost redex first;

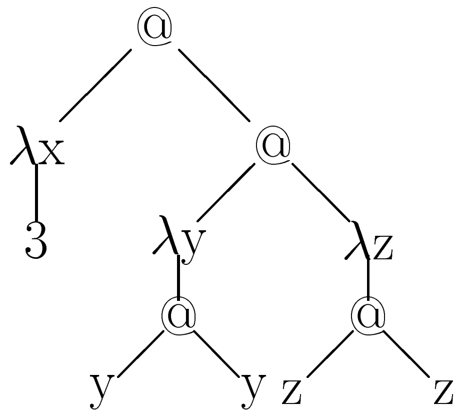
• **normal order**, which says to always reduce the leftmost outermost redex first.

Normalization Theorem

The **normal order** reduction strategy always leads to a normal form, if one exists.

Applicative order, which corresponds to the idea of evaluating a function's arguments **before** invoking the function, does **not** have this property.

Example revisited: `(`λ`x.3)(D D)`



Here the λx redex is the (only) outermost one and the λy redex is the (only) innermost one. Normal-order reduction says to do the λx reduction first, so `(D D)` is never evaluated. Applicative order reduction says to evaluate `(D D)` repeatedly.

Intuitively, normal order evaluation wins over applicative order here because the function λ`x.3` doesn't use its argument, so there's no point in trying to evaluate it (fruitlessly).

A function that uses its argument is said to be **strict** in that argument; e.g., λ`x.3` is not strict in `x`, but λ`x.x` is.

Eager or Lazy?

The **applicative order** strategy corresponds to **call-by-value**, as used in most imperative languages. In FL's this is usually called **eager evaluation**.

The **normal order** strategy corresponds to **call-by-name** (as found in Algol-60).

Although the normal order strategy appears better (it computes something useful more often) it is significantly less efficient to implement. Intuitively, this is because if the argument is a complicated expression, call-by-name must transmit the entire expression including the values of its free variables, whereas call-by-value needs only to transmit the value of the expression, which may be much simpler.

There's another obvious problem with call-by-name: if an argument **is** needed, it may be needed at several points in the body, and hence get reduced several times. Practical implementations avoid this problem by **sharing** the result of evaluating the argument at all points where it is used. This combination of call-by-name with sharing is called **call-by-need** or (loosely) **lazy evaluation**. (To model sharing, we need to change our tree representation of λ-expressions to a **graph**.)

Well-known lazy functional languages include **Miranda** and **Haskell**; eager functional languages include the (pure subsets of) **Scheme** and **ML**.

Weak Head Normal Form

In real programming languages, we don't expect the body of a function to be evaluated (even in part) before the function has been called, i.e., before the formal parameters have been instantiated. (If it happens, we tend to think of it as an optimization, as with a loop invariant.)

But either of our strategies, as defined so far, may do such reductions. An applicative order example:

`(λx.+ (+ 2 3) x) 8`

The leftmost innermost redex is `(+ 2 3)`, so this will be δ-reduced to `5` **before** the top-level reduction occurs.

To avoid this behavior, most functional languages implementations "don't evaluate under a λ." Technically, we stop evaluating when we reach **weak head normal form (WHNF)**.

A consequence of this implementation approach is that we never have name clash problems, since we never reduce an expression containing free variables (assuming we started with a closed expression).

Recursion

Suppose we wish to define a recursive function, such as the factorial function:

`FAC =` $($$\lambda$`n.if (= n 0) 1 (* n (FAC (- n 1))))`

This definition is faulty: it relies on a ability to **name** a λ-expression (`FAC`) and refer to that name from within the expression itself.

The problem is that λ-abstractions are **anonymous**. (We've named λ-expressions before, but only as an abbreviation mechanism which could be easily "unfolded.") The only names a λ-abstraction can refer to are those of its arguments (or the arguments to enclosing abstractions).

So the trick is to rewrite the recursive function so that it takes **itself** as an extra argument! We do this by β-abstraction:

$\mathtt{FAC} \leftrightarrow \lambda\mathtt{n.(\ldots FAC\ldots)}$

becomes

$\mathtt{FAC} \leftrightarrow \lambda\mathtt{f.(}\lambda\mathtt{n.(\ldots f\ldots))\ FAC}$

We can rewrite this as:

$\mathtt{FAC} \leftrightarrow \mathtt{H\ FAC}$ where $\mathtt{H} \equiv \lambda\mathtt{f.(}\lambda\mathtt{n.(\ldots f\ldots))}$

Note that `H` is a perfectly ordinary λ-abstraction that does **not** involve recursion.

Fixed Points

The factorial function `FAC` is a solution to the equation

`FAC` $\leftrightarrow$ `H FAC`

i.e., it is a function `FAC` such that the result of applying `H` to `FAC` is just `FAC`. We say that `FAC` is a **fixed point** of the function `H`.

It is easy to find fixed points for some functions. For example the function λ`x.* x x` has both `0` and `1` as fixed points. In general, a function may have 0, 1, some, or infinitely many fixed points.

It's not so easy to see what a fixed point of `H` should look like, but for a moment let's just **assume** the existence of a function **Y** that takes **any** function as argument and returns a fixed point of that function as result. That is, for any function **F**,

`F (Y F)` $\leftrightarrow$ `Y F`

Then, in particular, we have `H (Y H)` $\leftrightarrow$ `Y H`, so

`FAC` $\equiv$ `Y H`

is a solution to our equation for for the factorial function that doesn't involve recursion!

Example

With our definitions

$\texttt{FAC} \equiv \texttt{Y H}$

$\texttt{H} \equiv \lambda\texttt{f.}\lambda\texttt{n.if (= n 0) 1 (* n (f (- n 1)))}$

we can now compute, e.g., factorial(1):

$$
\begin{array}{ll}
 & \texttt{FAC 1} \\
= & \texttt{Y H 1} \\
\leftrightarrow & \texttt{H (Y H) 1} \\
= & (\lambda\texttt{f.}\lambda\texttt{n.if (= n 0) 1 (* n (f (- n 1))))} \\
 & \texttt{(Y H) 1} \\
\rightarrow_\beta & (\lambda\texttt{n.if (= n 0) 1 (* n (Y H (- n 1)))) 1} \\
\rightarrow_\beta & \texttt{if (= 1 0) 1 (* 1 (Y H (- 1 1)))} \\
\rightarrow_\delta^* & \texttt{* 1 (Y H 0)} \\
\leftrightarrow & \texttt{* 1 (H (Y H) 0)} \\
= & \texttt{* 1 ((}\lambda\texttt{f.}\lambda\texttt{n.if (= n 0) 1 (* n (f (- n 1))))} \\
 & \qquad\texttt{(Y H) 0)} \\
\rightarrow_\beta & \texttt{* 1 ((}\lambda\texttt{n.if (= n 0) 1 (* n (Y H (- n 1))))} \\
 & \qquad\texttt{0)} \\
\rightarrow_\beta & \texttt{* 1 (if (= 0 0) 1 (* n (Y H (- 0 1))))} \\
\rightarrow_\delta & \texttt{* 1 1} \\
\rightarrow_\delta & \texttt{1}
\end{array}
$$

Defining Y

We have seen that recursion can be expressed wholly in terms of a **fixed-point combinator** Y. But how do we define Y?

We could build it into our applied λ-calculus as a special function with a suitable δ-rule. In fact, this is more or less what we will do in practice.

Amazingly, it is also possible to define Y as an ordinary expression in the pure λ-calculus! Here is one definition:

$$\mathtt{Y} \equiv \lambda\mathtt{h.}(\lambda\mathtt{x.h\ (x\ x)})(\lambda\mathtt{x.h\ (x\ x)})$$

Let's check it out:

$$\begin{array}{ll} & \mathtt{Y\ H} \\ = & (\lambda\mathtt{.h.}(\lambda\mathtt{x.h\ (x\ x)})(\lambda\mathtt{x.h\ (x\ x)}))\ \mathtt{H} \\ \leftrightarrow_\beta & (\lambda\mathtt{x.H\ (x\ x)})(\lambda\mathtt{x.H\ (x\ x)}) \\ \leftrightarrow_\beta & \mathtt{H}\ ((\lambda\mathtt{x.\ \ H\ (x\ x)})(\lambda\mathtt{x.\ \ H\ (x\ x)})) \\ \leftrightarrow_\beta & \mathtt{H}\ ((\lambda\mathtt{.h.}(\lambda\mathtt{x.h\ (x\ x)})(\lambda\mathtt{x.h\ (x\ x)}))\ \mathtt{H}) \\ = & \mathtt{H\ (Y\ H)} \end{array}$$

Note the similarity between this definition of Y and the looping expression (D D). In fact, this version of Y doesn't work with call-by-value evaluation because it goes into an infinite loop!

Fortunately, there are many ways to write Y; here's one that **does** work with call-by-value:

$$\mathtt{Y} \equiv \lambda\mathtt{f.}(\lambda\mathtt{x.f}(\lambda\mathtt{y.x\ x\ y}))(\lambda\mathtt{x.f\ }(\lambda\mathtt{y.x\ x\ y}))$$

Pure Fun

In principle, we can do without built-in constants and δ-rules altogether by **encoding** useful types and operators directly in the pure λ-calculus. In general, this means using λ-abstractions to represent values.

For example, to model **booleans** we need a way of representing `true` and `false`, functions correspondig to `and`, `or`, etc., and an equivalent of `if` that selects one of two expressions based on the value of the boolean. Here's how:

$$
\begin{array}{lcl}
\overline{\texttt{true}} & \equiv & \lambda\texttt{x}.\lambda\texttt{y}.\texttt{x} \\
\overline{\texttt{false}} & \equiv & \lambda\texttt{x}.\lambda\texttt{y}.\texttt{y} \\
\overline{\texttt{if}} & \equiv & \lambda\texttt{f}.\lambda\texttt{x}.\lambda\texttt{y}.\texttt{f x y} \\
\overline{\texttt{and}} & \equiv & \lambda\texttt{x}.\lambda\texttt{y}.\texttt{x y }\overline{\texttt{false}} \\
\ldots & &
\end{array}
$$

Example

$$
\begin{array}{ll}
 & \overline{\texttt{if}}\ \overline{\texttt{true}}\ \texttt{0 1} \\
= & (\lambda\texttt{f}.\lambda\texttt{x}.\lambda\texttt{y}.\texttt{f x y})\ \overline{\texttt{true}}\ \texttt{0 1} \\
\to_\beta & (\lambda\texttt{x}.\lambda\texttt{y}.\overline{\texttt{true}}\ \texttt{x y})\ \texttt{0 1} \\
\to_\beta & (\lambda\texttt{y}.\overline{\texttt{true}}\ \texttt{0 y})\ \texttt{1} \\
\to_\beta & \overline{\texttt{true}}\ \texttt{0 1} \\
= & (\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x})\ \texttt{0 1} \\
\to_\beta & (\lambda\texttt{y}.\texttt{0})\ \texttt{1} \\
\to_\beta & \texttt{0}
\end{array}
$$

Church Numerals

$$
\begin{array}{lll}
\overline{\mathtt{0}} & \equiv & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{x} \\
\overline{\mathtt{1}} & \equiv & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f\ x} \\
\overline{\mathtt{2}} & \equiv & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f(f\ x)} \\
\ldots & & \\
\overline{n} & \equiv & \lambda\mathtt{f}.\lambda\mathtt{x}.\overbrace{\mathtt{f}(\mathtt{f}\ldots(\mathtt{f}}^{n\text{ times}}\ \mathtt{x})\ldots) \\
\overline{\mathtt{succ}} & \equiv & \lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f(n\ f\ x)} \\
\overline{\mathtt{iszero}} & \equiv & \lambda\mathtt{n}.\mathtt{n}(\lambda\mathtt{x}.\overline{\mathtt{false}})\ \overline{\mathtt{true}}\ \ldots
\end{array}
$$

Example

$$
\begin{array}{ll}
 & \overline{\mathtt{succ}}\ \overline{\mathtt{2}} \\
= & (\lambda\mathtt{n}.\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f(n\ f\ x)})\ \overline{\mathtt{2}} \\
\rightarrow_\beta & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}(\overline{\mathtt{2}}\ \mathtt{f\ x}) \\
= & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}((\lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f(f\ x)})\ \mathtt{f\ x}) \\
\rightarrow_\beta & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f}((\lambda\mathtt{x}.\mathtt{f(f\ x)})\ \mathtt{x}) \\
\rightarrow_\beta & \lambda\mathtt{f}.\lambda\mathtt{x}.\mathtt{f(f(f\ x))} \\
= & \overline{\mathtt{3}}
\end{array}
$$

Data Constructors

$$\begin{array}{lcl}\overline{\texttt{pair}} & \equiv & \lambda\texttt{x}.\lambda\texttt{y}.\lambda\texttt{f}.\texttt{f x y} \\ \overline{\texttt{fst}} & \equiv & \lambda\texttt{p}.\texttt{p}(\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x}) \\ \overline{\texttt{snd}} & \equiv & \lambda\texttt{p}.\texttt{p}(\lambda\texttt{x}.\lambda\texttt{y}.\texttt{y})\end{array}$$

Example

$$\overline{\texttt{pair}}\ \texttt{1 2 = }(\lambda\texttt{x}.\lambda\texttt{y}.\lambda\texttt{f}.\texttt{f x y})\ \texttt{1 2} \rightarrow^*_\beta \lambda\texttt{f}.\texttt{f 1 2}$$

so

$$\begin{array}{ll} & \overline{\texttt{fst}}\ (\overline{\texttt{pair}}\ \texttt{1 2}) \\ = & (\lambda\texttt{p}.\texttt{p}(\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x}))(\lambda\texttt{f}.\texttt{f 1 2}) \\ \rightarrow_\beta & (\lambda\texttt{f}.\texttt{f 1 2})(\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x}) \\ \rightarrow_\beta & (\lambda\texttt{x}.\lambda\texttt{y}.\texttt{x})\ \texttt{1 2} \\ \rightarrow^*_\beta & \texttt{1}\end{array}$$

Note that pairs and booleans are encoded in essentially the same way!