Concepts of programming languages

Lecture 5

Wouter Swierstra

Based on slides by Johan Jeuring, Andres Löh, Jan Rochel and Doaitse Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Announcements

- Submit your project proposal to me by email on Friday (anywhere on earth);
- The presentation schedule in now online
- The Scala & OCaml teams: meet me in the break to split topics more carefully.
- Thursday after lecture: exercise slot in this room.

Last time

- Study an example domain specific language (SVG).
- What different approaches are their to implementing domain specific languages?

This time

- What is the λ-calculus?
- How can we define its semantics?
- How can we define its type system?

The lambda calculus

Previously, we studied SVG – a realistic, but limited language.

The drawback is that our handling of variables could be distracted by many other language design choices.

For now, let's revisit how to treat variables, but focus on a smaller language: the *lambda calculus*.

The lambda calculus

The lambda calculus is the 'smallest programming language imaginable'.

It was originally introduced by Alonzo Church (1930's) as a foundation for mathematics – but surprisingly, it perfectly captures computation!

There are only three constructs:

$$
\begin{array}{lll}
e ::= x & & \text{(variables)} \\
\quad \mid & e\,e & \text{(application)} \\
\quad \mid & \lambda x\,.\,e & \text{(abstraction)}
\end{array}
$$

The lambda calculus – examples and conventions

Example terms:

$\lambda x \,.\, x\, x$
$\lambda x \,.\, (\lambda y \,.\, x\, z)\ (\lambda x \,.\, x\, a)$

- Note: application associates to the left: $a\, b\, c = (a\, b)\ c$
- Note: only unary functions and unary application – but we sometimes write $\lambda x\, y \,.\, e$ for $\lambda x \,.\, \lambda y \,.\, e$.
- Note: the function body of a lambda extends as far as possible to the right: $\lambda x \,.\, e\, f$ should be read as $\lambda x \,.\, (e\, f)$.

Bound and free variables

- An abstraction $\lambda x\,.\,e$ *binds* variable x in its body e.
- An occurrence of a variable that is not bound by a lambda is called *free*.

Examples:

- x occurs free in $\lambda y\,.\,y\,(\lambda z\,.\,x)$
- $(\lambda x\,.\,x\,z)\;y\;x$ has one bound and one free occurrence of x.

A term without free variables is called a *closed* term.

We consider two lambda terms that are equal, up to renaming of bound variables, *equivalent* (or more formally α-equivalent).

Equivalence – examples

For example, the following two terms are equivalent:

- $\lambda x\,.\,\lambda t\,.\,x\,y\,t$
- $\lambda z\,.\,\lambda u\,.\,z\,y\,u$

But these are both different from: $\lambda a\,.\,\lambda s\,.\,a\,z\,s$ – we can allowed to rename the *bound* variables.

λ-Calculus: β-Reduction

A term of the form $\lambda x\,.\,e$ is called an *abstraction* or *lambda binding*; e is called the abstraction's *body*.

The central rewrite rule of the λ-calculus is β-reduction:

$$(\lambda x\,.\,e)\ a \to_\beta\ e\,[x \mapsto a]$$

$[x \mapsto a]$:= substitution of all *free* occurrences of variable x by a

$$\begin{aligned} & (\lambda f\,.\,\lambda x\,.\,\lambda y\,.\,f\,y\,x)\ a\ b\ c \\ \to_\beta\ & (\lambda x\,.\,\lambda y\,.\,a\,y\,x)\ b\ c \\ \to_\beta\ & (\lambda y\,.\,a\,y\,b)\ c \\ \to_\beta\ & a\ c\ b \end{aligned}$$

An expression of the form $(\lambda x\,.\,e)\ (t)$ is called a β-redex.

β-reduction: properties

The definition of beta-reduction should be familiar from any other programming language: it allows you to fill in the arguments of a function call in the function's body.

In that sense, the lambda calculus really is a tiny programming language!

Beta reduction has several properties:

- arguments are consumed one by one
- functions are gradually destroyed when applied
- function definitions do not live in a separate space

λ-Calculus: Name Capturing and α-conversion

Consider the following example:

$$\begin{aligned} & (\lambda y\,.\,(\lambda b\,.\,y\,b))\;a \\ \rightarrow_\beta\; & (\lambda b\,.\,y\,b)\;[y \mapsto a] \\ =\; & \lambda b\,.\,a\,b \end{aligned}$$

And now we can reduce the following equivalent expression:

$$\begin{aligned} & (\lambda y\,.\,(\lambda a\,.\,y\,a))\;a \\ \rightarrow_\beta\; & (\lambda a\,.\,y\,a)\;[y \mapsto a] \\ =\; & \lambda a\,.\,a\,a \end{aligned}$$

What went wrong? Two equivalent expressions produced different results!

Problem: a is *captured* by the innermost lambda binding!

Capture avoiding substitution

The substitution $[x \mapsto y]$ must be a *capture-avoiding* substitution – that is, it should renames the abstraction variable if necessary:

$$\begin{array}{rl} & (\lambda y\,.\,(\lambda a\,.\,y\,a))\;a \\ \to_\beta & (\lambda a\,.\,y\,a)\;[y \mapsto a] \\ \to_\alpha & (\lambda a'\,.\,y\,a')\;[y \mapsto a] \\ = & \lambda a'\,.\,a\,a' \end{array}$$

Note that we introduce an explicit α-conversion step, renaming a to a'.

Capture avoiding substitution

We can define such a capture avoiding substitution as follows:

$$
\begin{array}{ll}
x\,[x \mapsto t] & = t \\
y\,[x \mapsto t] & = y \quad \text{when } x \not\equiv y \\
(t_1\ t_2)\,[x \mapsto t] & = (t_1\,[x \mapsto t])\,(t_2\,[x \mapsto t]) \\
(\lambda y\,.\,s)\,[x \mapsto t] & = \lambda y\,.\,s\,[x \mapsto t]
\end{array}
$$

provided $y \not\equiv x$ and y does not occur free in t.

Note that this last rule may require α-renaming the variable y in order to proceed with the substitution.

Transitive, reflexive closure

Beta-reduction allows us to define a single reduction step...

...but what if we're interested in evaluating a more complicated λ-term?

We can define the relation $t \to_\beta^* t'$ as follows:

- $t \to_\beta^* t$ for any term t
- if $t_1 \to_\beta t_2$ and $t_2 \to_\beta^* t_3$, then also $t_1 \to_\beta^* t_3$

If $t \to_\beta^* t'$, we can reach t' from t after zero or more β-reduction steps.

λ-Calculus: β-equivalence

When we can find a term v, such that $t \to_\beta^* v$ or $t' \to_\beta^* v$ we call t and t' β-equivalent.

In that case, we sometimes write $t =_\beta t'$.

This corresponds to saying that the two lambda terms, t and t', correspond to the same program.

This is undecidable in general.

For example $(\lambda y\,.\,a\,y)\;b =_\beta (\lambda x\,.\,x\,b)\;a$ because
$(\lambda y\,.\,a\,y)\;b \to_\beta a\,b \leftarrow_\beta (\lambda x\,.\,x\,b)\;a$

Lambda terms and functional programming

Despite all its simplicity, the lambda calculus really captures the heart of (functional) programming.

A function like:

$$flip\ f\ x\ y = f\ y\ x$$

Is easy to represent by the following lambda term:

$$flip = \lambda f\ .\ \lambda x\ .\ \lambda y\ .\ f\ y\ x$$

In fact, this is how GHC represents programs under the hood.

Extended example

Given the following definitions:

$$
\begin{aligned}
\mathit{main} &= \mathit{print}\ (\mathit{flip}\ \mathit{map}\ [1\,..]\ \mathit{inc}) \\
\mathit{print}\ x &= \mathit{putStrLn}\ (\mathit{show}\ x) \\
\mathit{flip}\ f\ x\ y &= f\ y\ x \\
\mathit{inc}\ x &= x + 1 \\
\mathit{map}\ f &= \ldots
\end{aligned}
$$

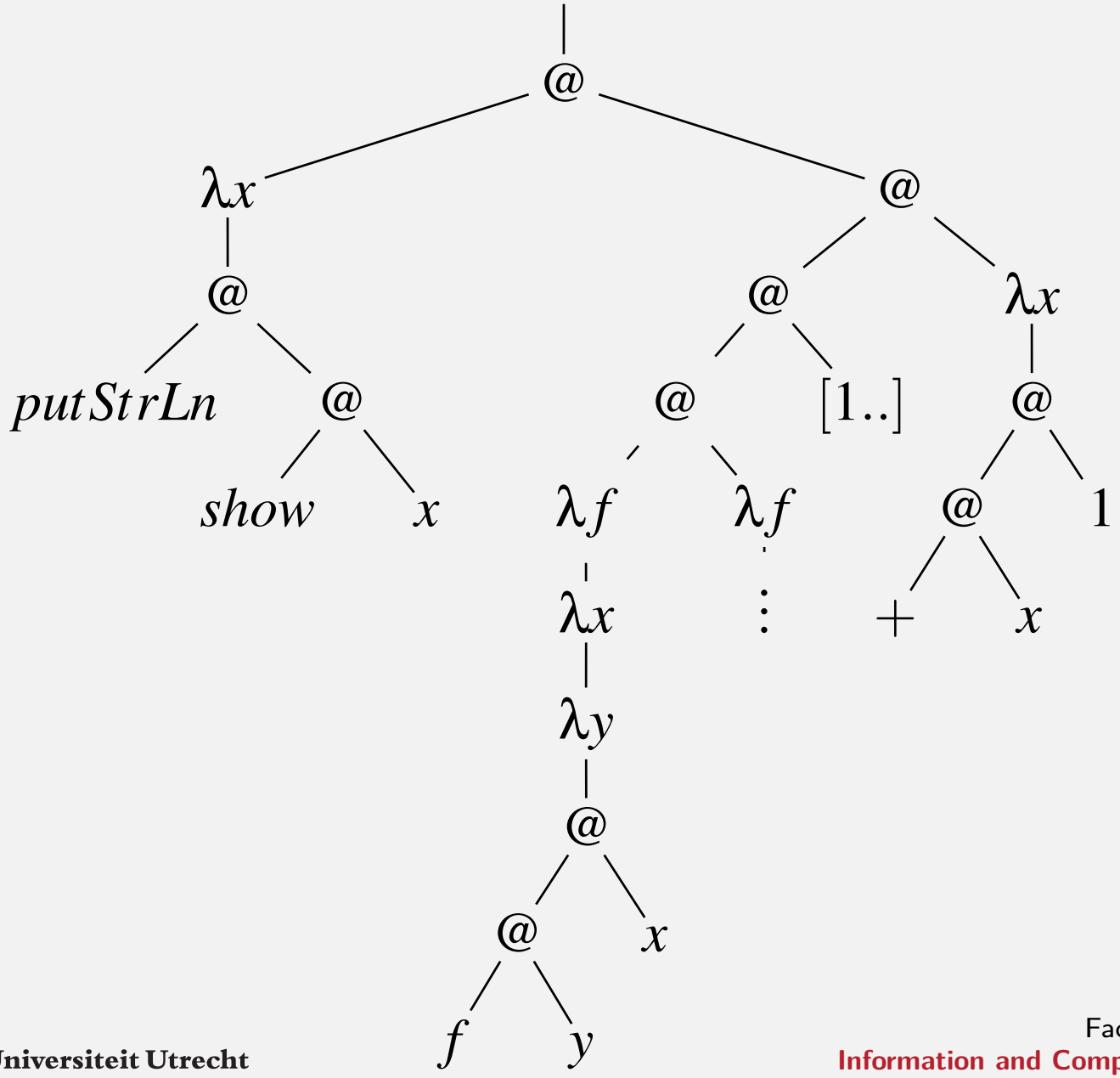
These can be desugared as:

$$
\begin{aligned}
\mathit{main} &= \mathit{print}\ (\mathit{flip}\ \mathit{map}\ [1\,..]\ \mathit{inc}) \\
\mathit{print} &= \lambda x\,.\ \mathit{putStrLn}\ (\mathit{show}\ x) \\
\mathit{flip} &= \lambda f\,.\ \lambda x\,.\ \lambda y\,.\ f\ y\ x \\
\mathit{inc} &= \lambda x\,.\ x + 1 \\
\mathit{map} &= \lambda f\,.\ \ldots
\end{aligned}
$$

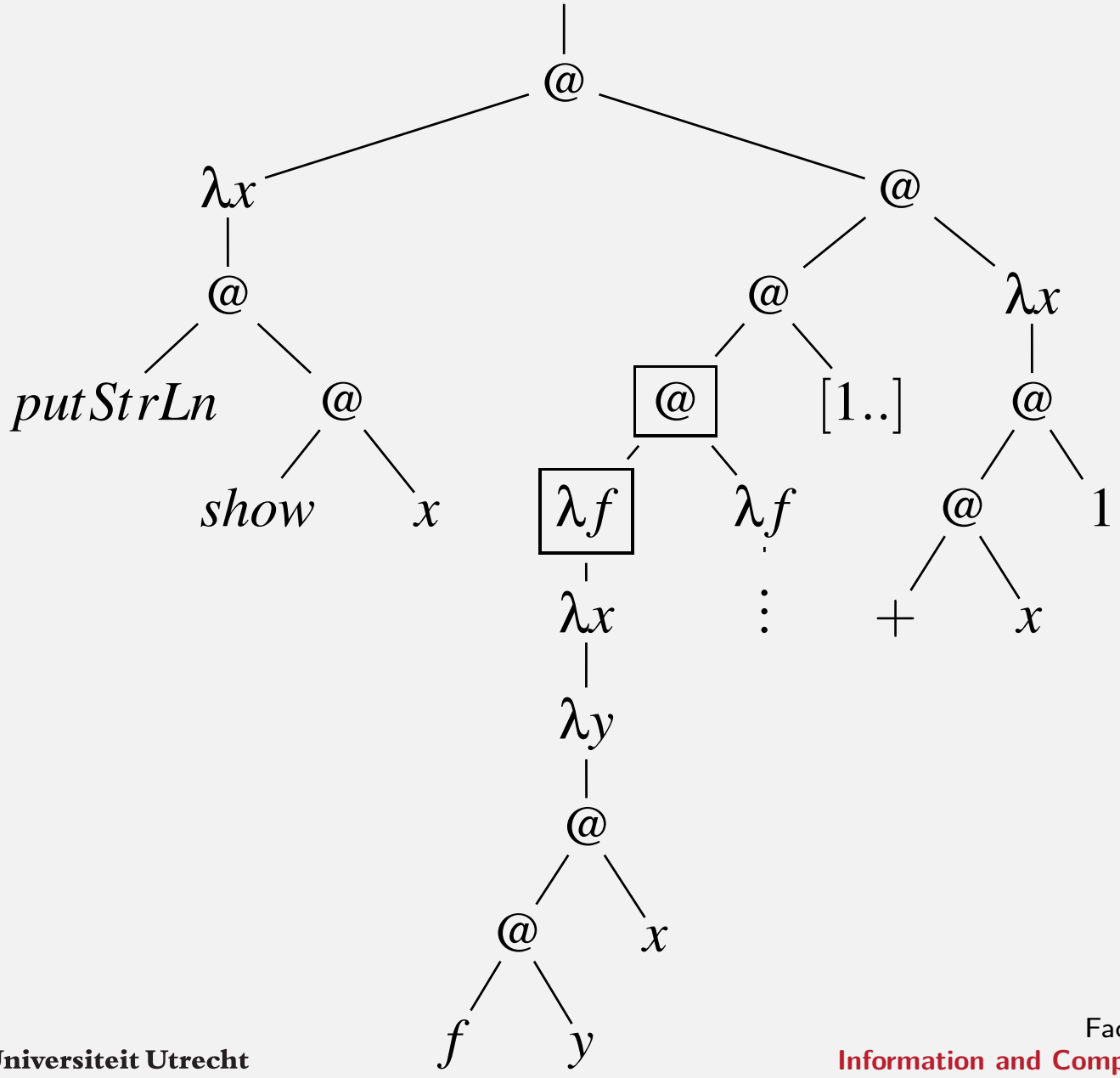
And after substituting function calls with their definition:

$$(\lambda x\,.\ \mathit{putStrLn}\ (\mathit{show}\ x))\ ((\lambda f\,.\ \lambda y\,.\ \lambda x\,.\ f\ y\ x)\ \ldots$$

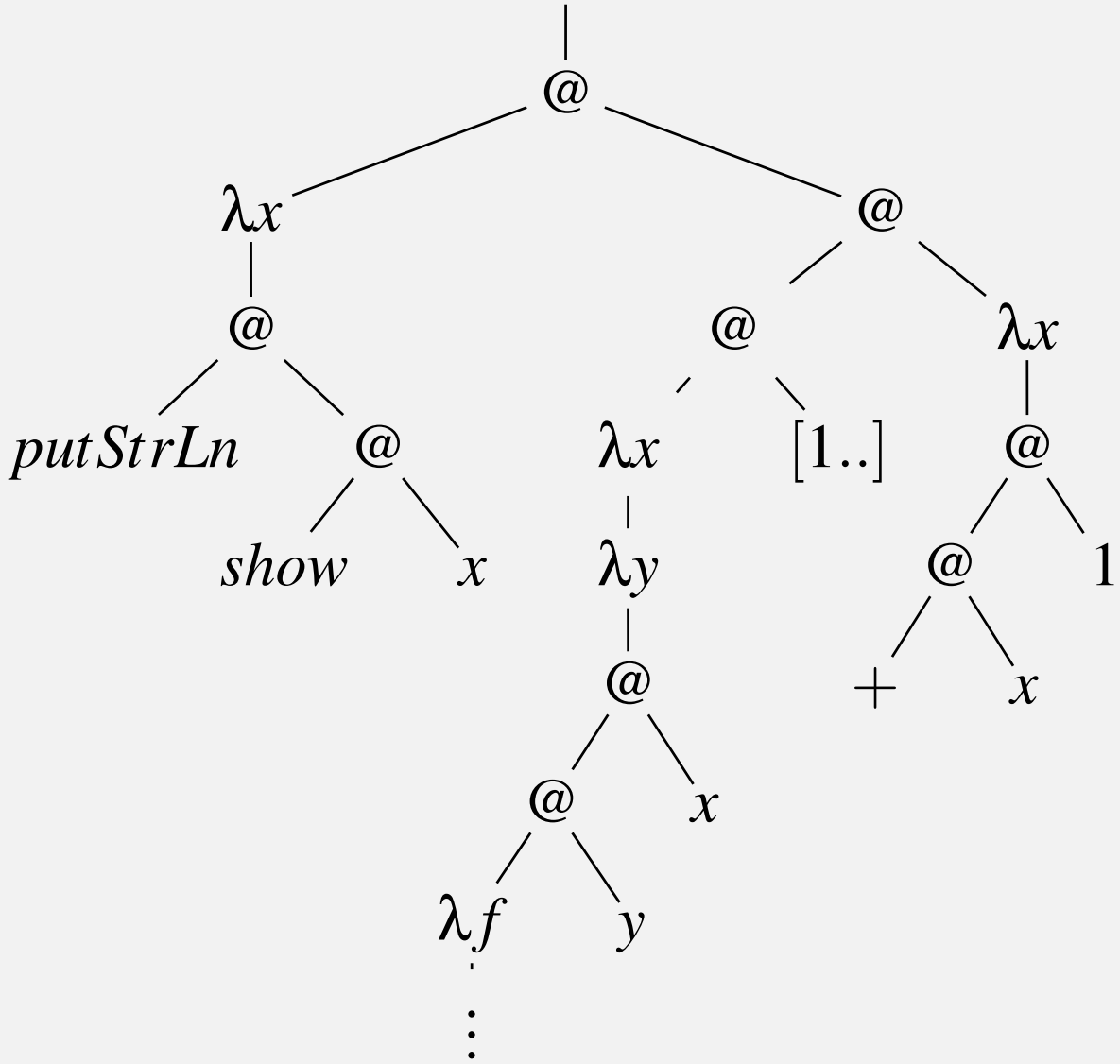
Example as a Syntax-Tree



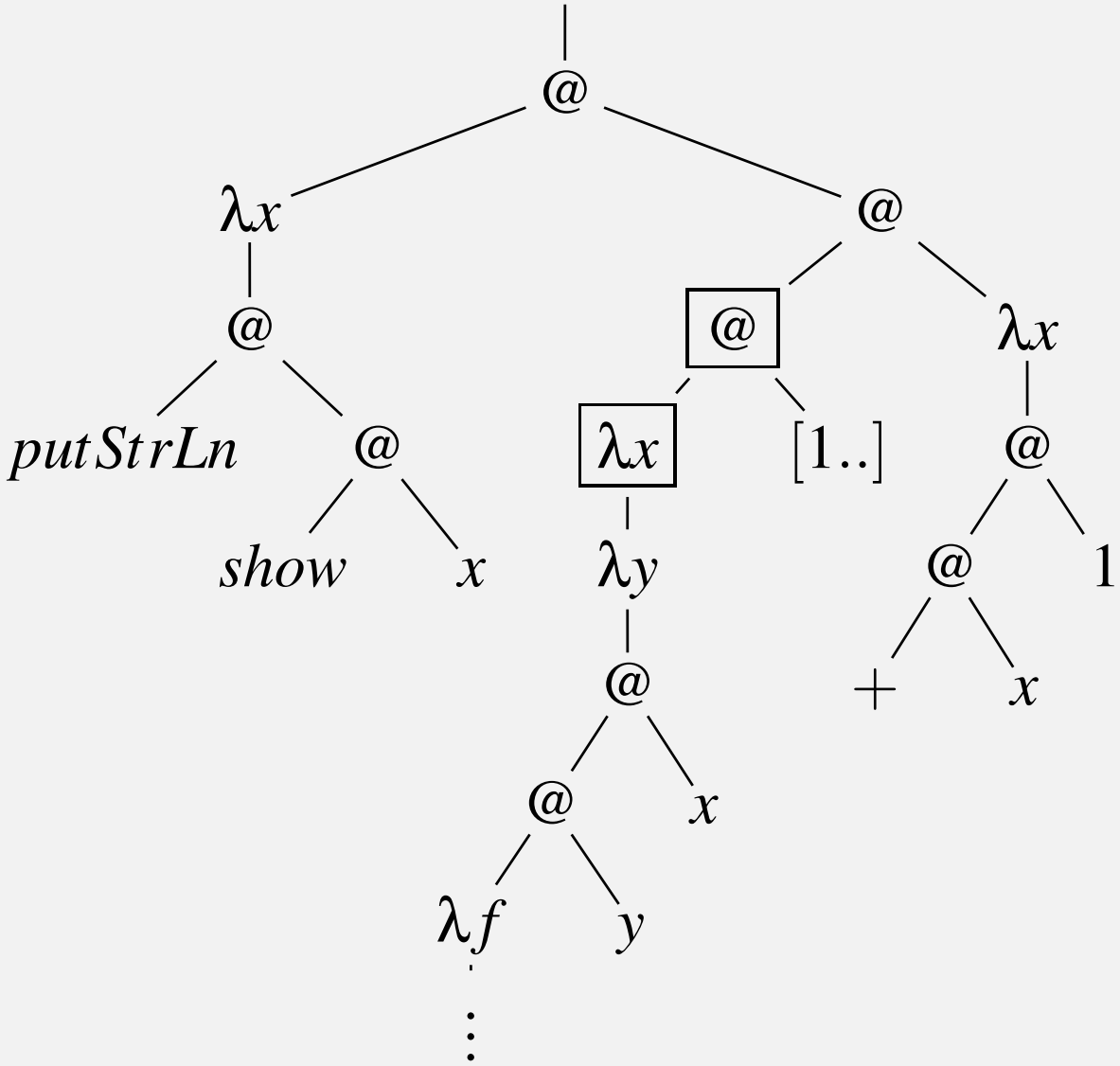
Example as a Syntax-Tree



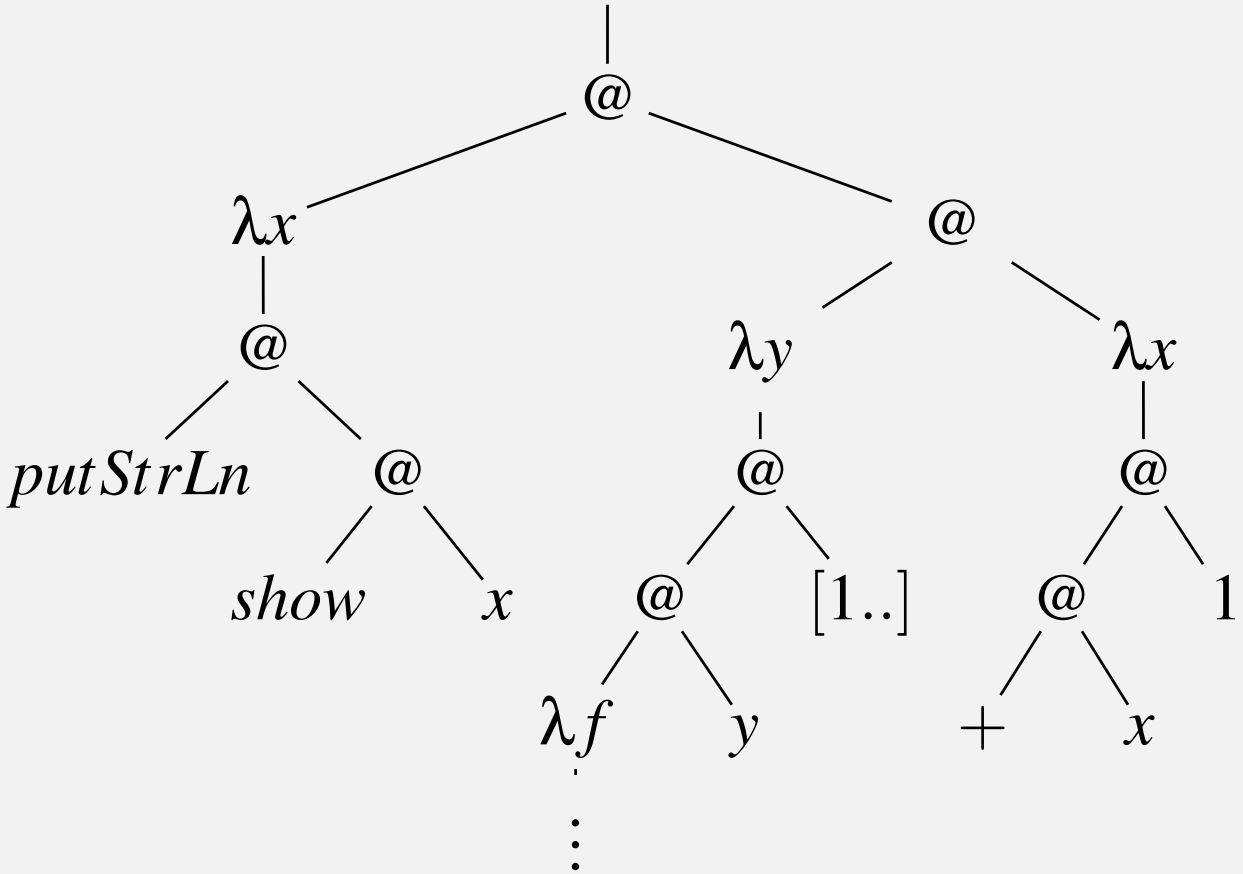
Example as a Syntax-Tree



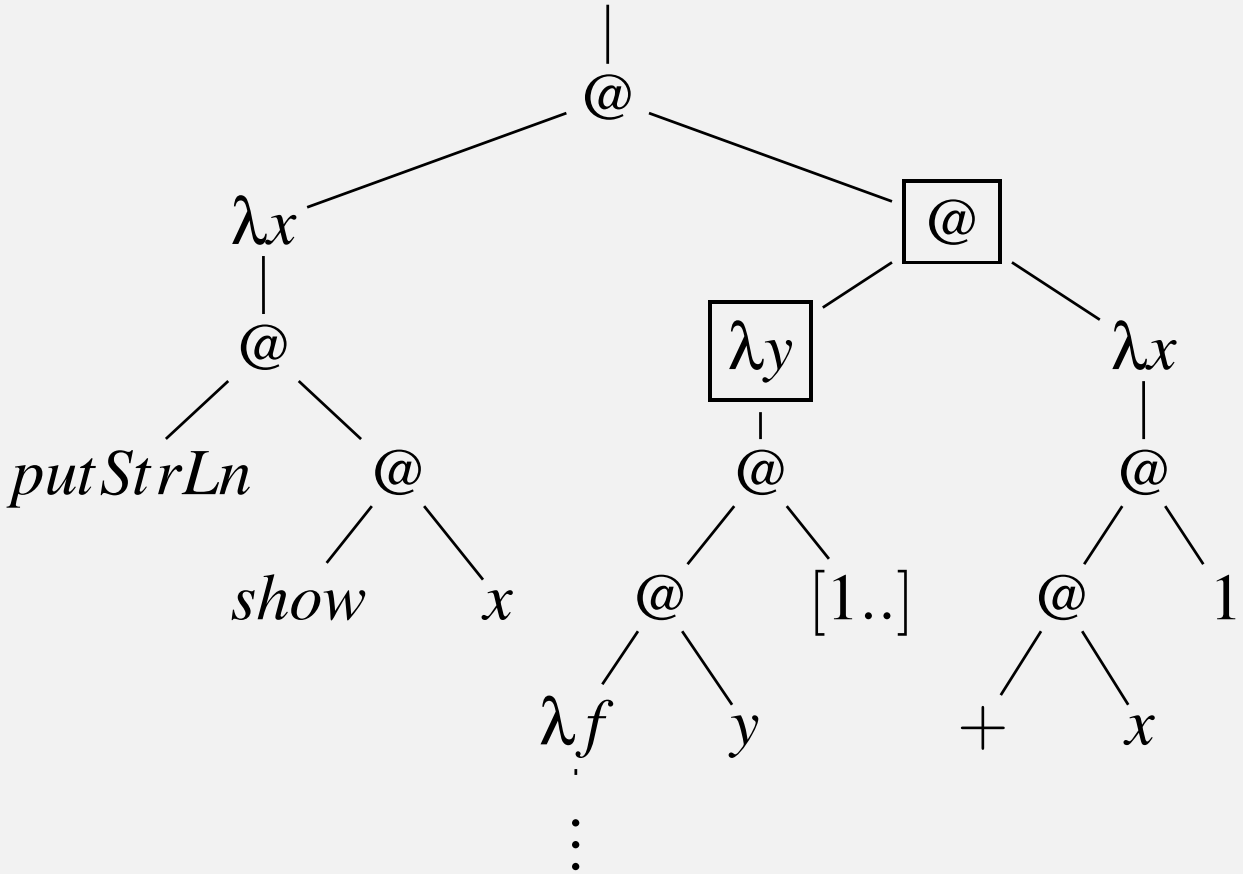
Example as a Syntax-Tree



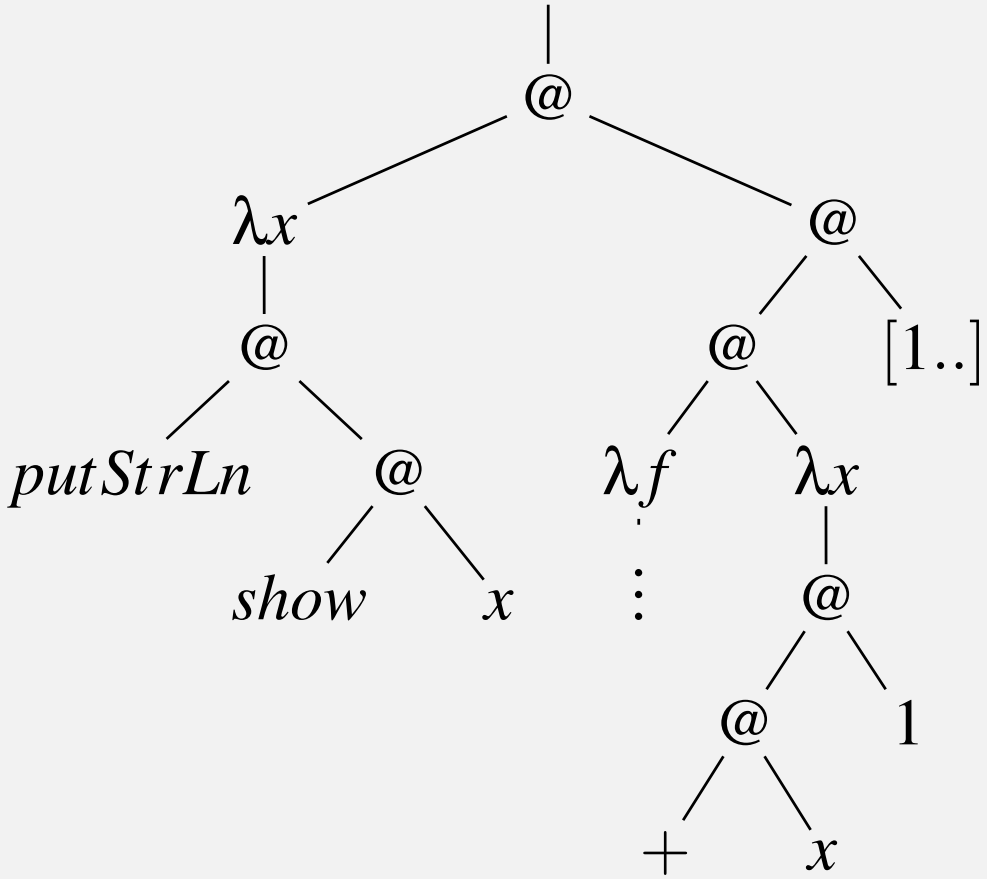
Example as a Syntax-Tree



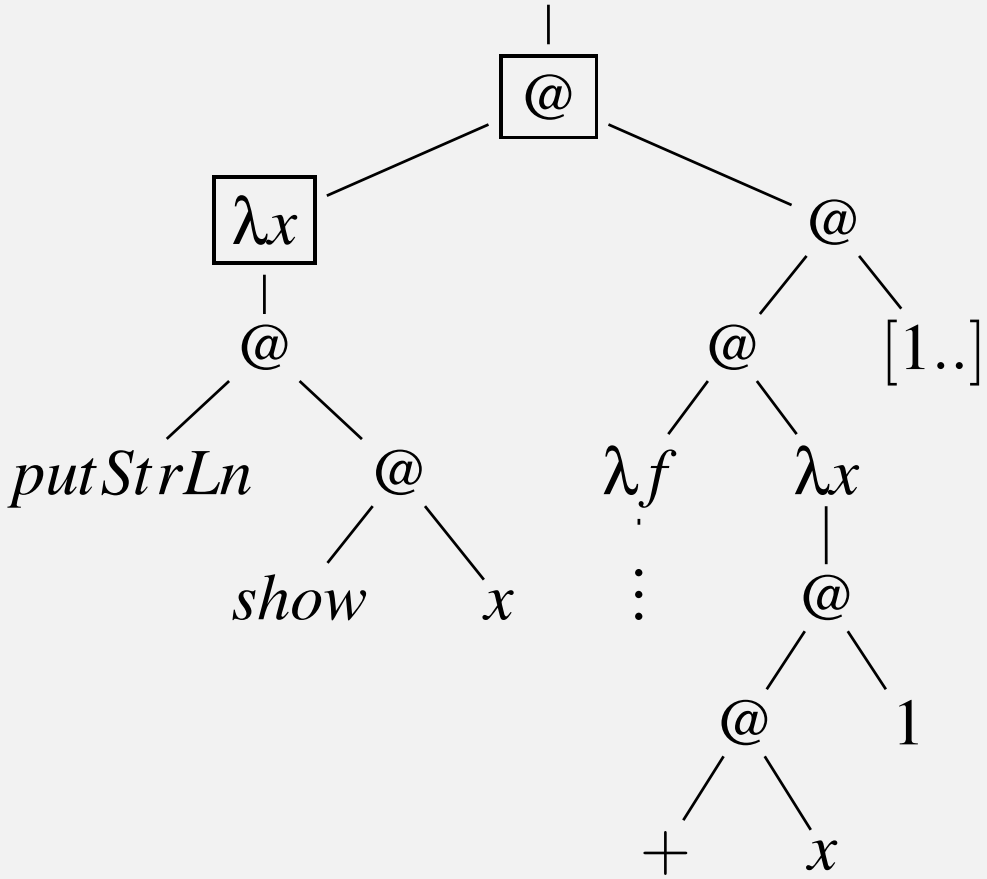
Example as a Syntax-Tree



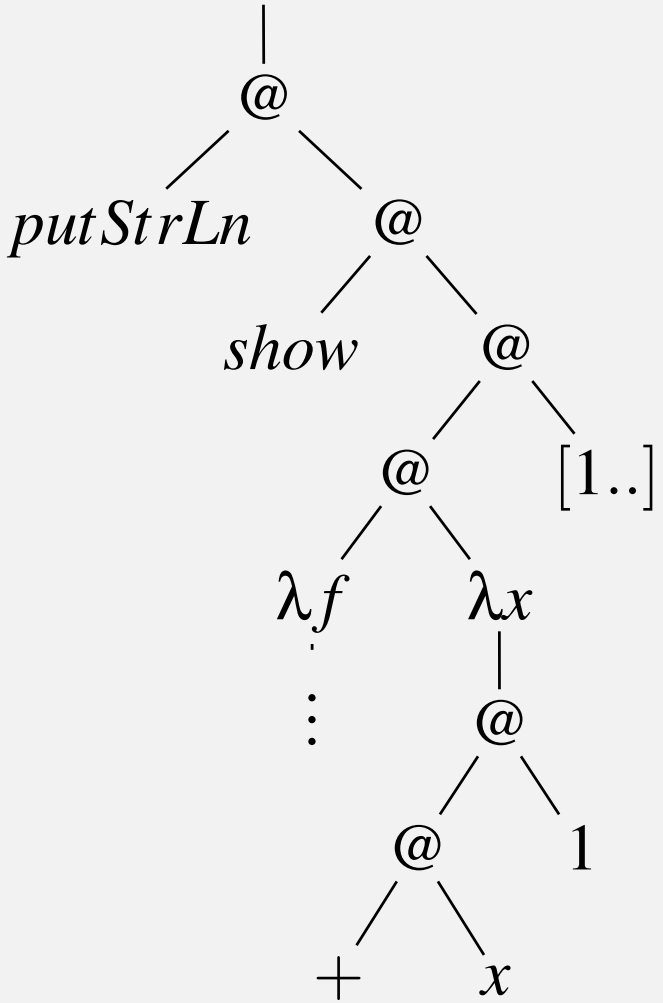
Example as a Syntax-Tree



Example as a Syntax-Tree



Example as a Syntax-Tree

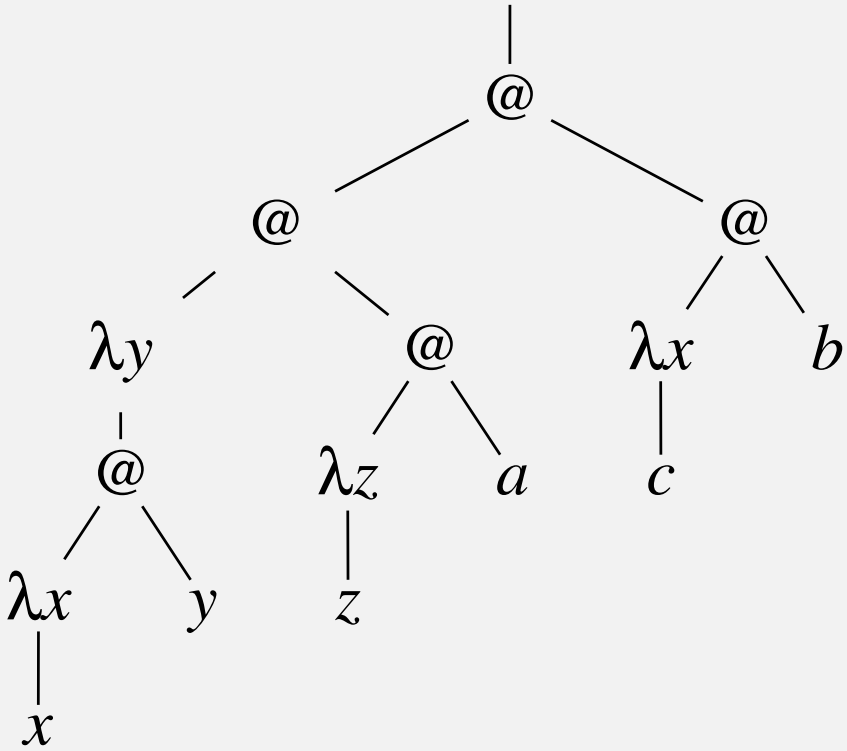


Evaluation order

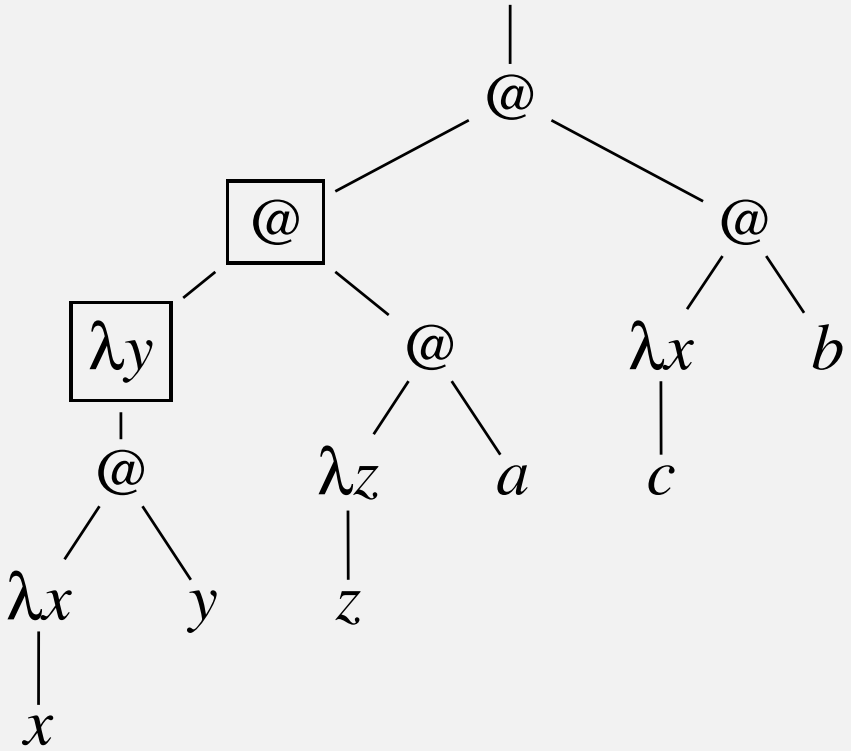
As we saw previously, we can choose different *evaluation orders*:

- **strict** languages evaluate arguments to a value, before beta reducing;
- **non-strict** languages evaluate leftmost-outermost beta redex on the spine; never reduce under lambdas. This reduces a term to *weak head normal form* – we will have a lambda or stuck application at the top level, but there may still be beta redexes.

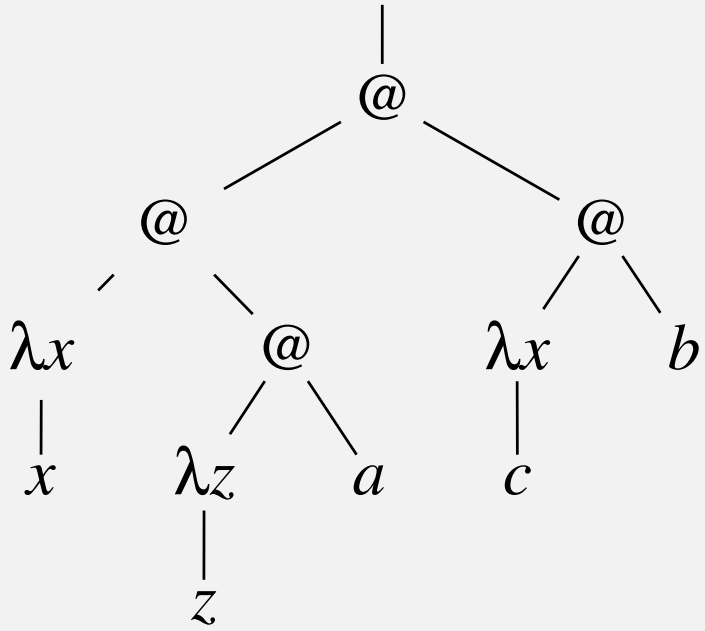
Example: Non-strict Evaluation



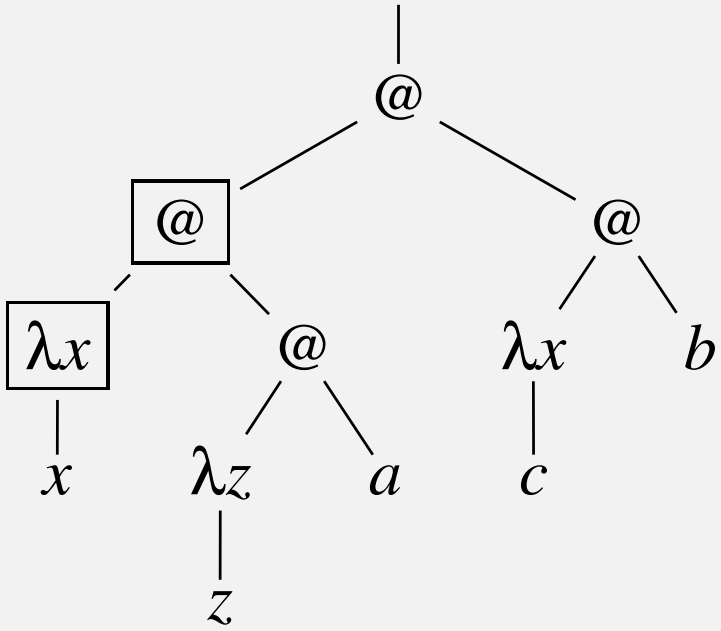
Example: Non-strict Evaluation



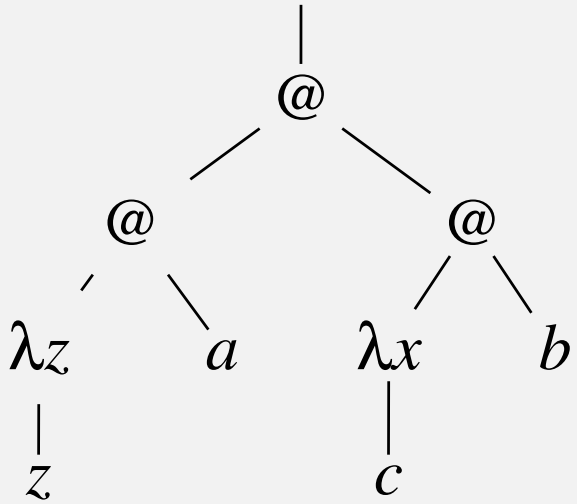
Example: Non-strict Evaluation



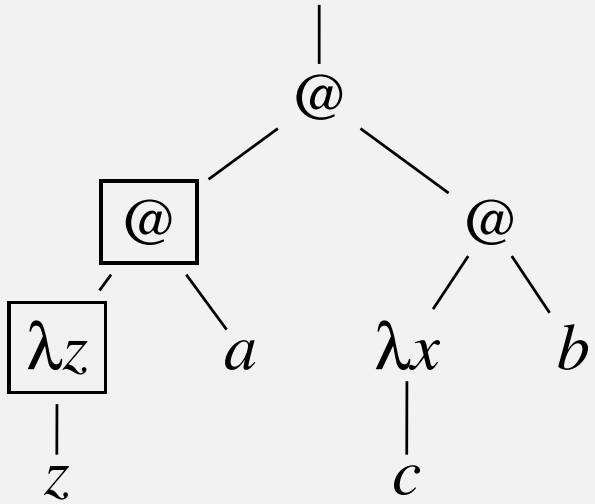
Example: Non-strict Evaluation



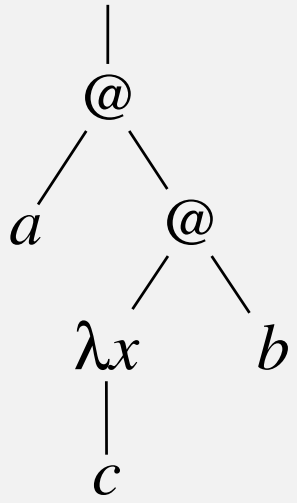
Example: Non-strict Evaluation



Example: Non-strict Evaluation



Example: Non-strict Evaluation



Term is a WHNF but not a normal form.

Semantics

To complete our definition of the lambda calculus, we need to specify its semantics.

To do so, we'll define a handful of *rules* capturing how to perform a single evaluation step.

These rules should – of course – include β-reduction – but they also need to fix the order of evaluation.

Operational Semantics

We will define an *operational semantics* for the lambda calculus.

This corresponds to defining a relation between two terms, $t \to t'$, when we can perform a single reduction step from t to t'.

Note that the *values* of our language are variables and lambda abstractions. We will use the variable v (rather than t) to refer to values.

Evaluation rules

$$\frac{t_1 \to t_2}{t_1\ t \to t_2\ t}\ \text{Left}$$

$$\frac{t_1 \to t_2}{v\ t_1 \to v\ t_2}\ \text{Right}$$

$$\frac{}{(\lambda x\ .\ t)\ v \to t\ [x \mapsto v]}\ \text{Beta}$$

Question: Is this strict or lazy?

Evaluation rules

$$\frac{t_1 \to t_2}{t_1\ t \to t_2\ t}\ \text{Left}$$

$$\frac{t_1 \to t_2}{v\ t_1 \to v\ t_2}\ \text{Right}$$

$$\frac{}{(\lambda x\ .\ t)\ v \to t\ [x \mapsto v]}\ \text{Beta}$$

Question: Is this strict or lazy?

The beta rule will only trigger when the argument is a value; hence these rules describe a *strict* evaluation order.

Evaluation rules

$$\frac{t_1 \to t_2}{t_1\ t \to t_2\ t}\ \text{Left}$$

$$\frac{t_1 \to t_2}{v\ t_1 \to v\ t_2}\ \text{Right}$$

$$\frac{}{(\lambda x\,.\,t)\ v \to t\,[x \mapsto v]}\ \text{Beta}$$

Question: Why do these rules fix the evaluation order?

Evaluation rules

$$\frac{t_1 \to t_2}{t_1\ t \to t_2\ t}\ \text{Left}$$

$$\frac{t_1 \to t_2}{v\ t_1 \to v\ t_2}\ \text{Right}$$

$$\frac{}{(\lambda x\ .\ t)\ v \to t\,[x \mapsto v]}\ \text{Beta}$$

Question: Why do these rules fix the evaluation order?

The beta rule will only trigger when the argument is a value; the second rule will only trigger when the left-hand side of an application is a value.

What can you do with lambda terms?

The *untyped* lambda calculus is surprisingly expressive.

In fact you can encode:

- booleans and if-then-else;
- natural numbers and iteration;
- arbitrary recursion;
- ...

all using untyped lambda terms...

In fact, it's a Turing complete language.

But in general, we're (more) interested in *typed* programming languages.

Terms and Types

Terms

$$
\begin{array}{lll}
e ::= x & & \text{variables} \\
\quad \mid & e\, e & \text{application} \\
\quad \mid & \lambda(x : \tau)\ .\ e & \text{lambda abstraction}
\end{array}
$$

The only new thing is that lambda abstractions are annotated with a type.

Terms and Types

Terms

$$
\begin{array}{lll}
e ::= x & & \text{variables} \\
\quad \mid & e\, e & \text{application} \\
\quad \mid & \lambda(x : \tau)\ .\ e & \text{lambda abstraction}
\end{array}
$$

The only new thing is that lambda abstractions are annotated with a type.

Types

$$
\begin{array}{lll}
\tau ::= \sigma & & \text{constant type} \\
\quad \mid & \tau \to \tau & \text{function type}
\end{array}
$$

We assume some collection of constant types σ, which could include, for instance, *bool*, *int*, *float*, ...

Typing rules

To define a type system for the lambda calculus, we will need to define an inductive relation between types and terms:

$$t : \tau$$

Expressing the fact that the term t has the type τ.

But unfortunately, this isn't quite enough...

Types and free variables

Question: How do we assign a type to a term with free variables?

$$\lambda(x : \mathit{Nat}) \;.\; \mathit{plus}\ x\ \mathit{one}$$

Types and free variables

Question: How do we assign a type to a term with free variables?

$$\lambda(x : Nat)\ .\ plus\ x\ one$$

Answer

We cannot unless we know the types of the free variables.

Environments

We therefore do not assign types to terms, but types to terms in a certain *environment* (also called a (type) *context*).

Environments

$$\begin{array}{lll} \Gamma ::= \varepsilon & & \text{empty environment} \\ \quad \mid & \Gamma, x : \tau & \text{binding} \end{array}$$

Later bindings for a variable always shadow earlier bindings.

The typing relation

A statement of the form

$$\Gamma \vdash e : \tau$$

can be read as follows:

"In environment Γ, term e has type τ."

The typing relation

A statement of the form

$$\Gamma \vdash e : \tau$$

can be read as follows:

"In environment Γ, term e has type τ."

Note that $\Gamma \vdash e : \tau$ is formally a ternary *relation* between an environment, a term and a type.

The $\vdash$ (called turnstile) and the colon are just notation for making the relation look nice but carry no meaning. We could have chosen the notation $T(\Gamma, e, \tau)$ for the relation as well, but $\Gamma \vdash e : \tau$ is commonly used.

Type rules

We are now free to choose the *inference rules* describing our typing relation:

Type rules

We are now free to choose the *inference rules* describing our typing relation:

Variables

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

Type rules – contd.

Application

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2}$$

Variables scope over the entire inference rule. Multiple occurences of the same variable must be instantiated with the same expressions – in this rule, both occurrences of τ_1 must be equal.

Type rules – contd.

Abstraction

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda(x : \tau_1) \,.\, e : \tau_1 \to \tau_2}$$

It pays off that we introduced the environments. The body of a lambda abstraction (viewed in isolation) can contain free occurrences of the bound variable.

Example derivation

$$\dfrac{\dfrac{}{x : Nat \in x : Nat}}{x : Nat \vdash x : Nat}$$
$$\varepsilon \vdash \lambda x : Nat \,.\, x : Nat \to Nat$$

Example derivation

$$\dfrac{\dfrac{}{x : Nat \in x : Nat}}{\dfrac{}{x : Nat \vdash x : Nat}}{\varepsilon \vdash \lambda x : Nat \,.\, x : Nat \to Nat}$$

Multiple applications of type rules can be stacked, leading (in general) to *proof trees* or *derivation trees*.

Examples – contd.

Let Γ abbreviate $\mathit{neg} : \mathit{Nat} \to \mathit{Nat}, \mathit{one} : \mathit{Nat}$.

$$\dfrac{\dfrac{\mathit{neg} : \mathit{Nat} \to \mathit{Nat} \in \Gamma}{\Gamma \vdash \mathit{neg} : \mathit{Nat} \to \mathit{Nat}} \quad \dfrac{\mathit{one} : \mathit{Nat} \in \Gamma}{\Gamma \vdash \mathit{one} : \mathit{Nat}}}{\Gamma \vdash \mathit{neg}\ \mathit{one} : \mathit{Nat}}$$

Examples – contd.

Let Γ abbreviate $\mathit{plus} : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat}, \mathit{one} : \mathit{Nat}$.

Exercise

Try to complete the following proof tree:

$$\frac{\dots}{\Gamma \vdash \lambda(x : \mathit{Nat}) \;.\; \mathit{plus}\; x\; \mathit{one} : \mathit{Nat} \to \mathit{Nat}}$$

Adding types

The simply typed lambda calculus itself knows only type variables and function types.

Adding types

The simply typed lambda calculus itself knows only type variables and function types.

We can simulate constants of certain types by placing them in the environment.

Adding types

The simply typed lambda calculus itself knows only type variables and function types.

We can simulate constants of certain types by placing them in the environment.

We can also actually extend the calculus with new types. Then we can also add reduction rules.

Booleans

For simplicity, let us play the addition of a new type through with Booleans.

Booleans

For simplicity, let us play the addition of a new type through with Booleans.

- the *datatype Bool*

Booleans

For simplicity, let us play the addition of a new type through with Booleans.

- the *datatype* *Bool*
- the *constructors* *True* and *False*

Booleans

For simplicity, let us play the addition of a new type through with Booleans.

- the *datatype* *Bool*
- the *constructors* *True* and *False*
- the *eliminator* **if-then-else**

Booleans

For simplicity, let us play the addition of a new type through with Booleans.

- the *datatype* *Bool*
- the *constructors* *True* and *False*
- the *eliminator* **if-then-else**
- reduction rules

Booleans

For simplicity, let us play the addition of a new type through with Booleans.

- the *datatype* *Bool*
- the *constructors* *True* and *False*
- the *eliminator* **if-then-else**
- reduction rules

Now, we also add type rules.

Boolean syntax

Terms

$e ::=$	...	(as before)
$\mid$	*True*	constructor
$\mid$	*False*	constructor
$\mid$	**if** e **then** e **else** e	eliminator

Boolean syntax

Terms

$e ::=$	...	(as before)
$\mid$	*True*	constructor
$\mid$	*False*	constructor
$\mid$	**if** e **then** e **else** e	eliminator

Types

$\tau ::=$	...	(as before)
$\mid$	*Bool*	type of Booleans

Boolean reduction rules

$$\frac{}{\textbf{if } \mathit{True} \textbf{ then } t_1 \textbf{ else } t_2 \rightarrow t_1}\ \mathsf{IfT}$$

$$\frac{}{\textbf{if } \mathit{False} \textbf{ then } t_1 \textbf{ else } t_2 \rightarrow t_2}\ \mathsf{IfF}$$

Boolean reduction rules

$$\frac{}{\textbf{if } \mathit{True} \textbf{ then } t_1 \textbf{ else } t_2 \rightarrow t_1}\ \mathsf{IfT}$$

$$\frac{}{\textbf{if } \mathit{False} \textbf{ then } t_1 \textbf{ else } t_2 \rightarrow t_2}\ \mathsf{IfF}$$

$$\frac{t \rightarrow t'}{\textbf{if } t \textbf{ then } t_1 \textbf{ else } t_2 \rightarrow \textbf{if } t' \textbf{ then } t_1 \textbf{ else } t_2}\ \mathsf{If}$$

Boolean type rules

$$\frac{}{\Gamma \vdash \mathit{True} : \mathit{Bool}} \qquad \frac{}{\Gamma \vdash \mathit{False} : \mathit{Bool}}$$

Boolean type rules

$$\frac{}{\Gamma \vdash \mathit{True} : \mathit{Bool}} \qquad \frac{}{\Gamma \vdash \mathit{False} : \mathit{Bool}}$$

$$\frac{\Gamma \vdash e_1 : \mathit{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \tau}$$

Other types

For other types (such as natural numbers or tuples or lists), we can also extend the simply typed lambda calculus.

We can also add additional constructs such as **let** or **case** to come closer to a full programming language.

Other types

For other types (such as natural numbers or tuples or lists), we can also extend the simply typed lambda calculus.

We can also add additional constructs such as **let** or **case** to come closer to a full programming language.

However, there are still a few fundamental shortcomings ...

Lack of polymorphism

In the simply typed lambda lambda calculus, there are no polymorphic functions.

We cannot type a term like

$$\begin{array}{l}\textbf{let } \mathit{id} = \lambda(x : ?)\ .\ x \\ \textbf{in } \ \textbf{if } \mathit{id}\ \mathit{True}\ \textbf{then } \mathit{id}\ 0\ \textbf{else } 1\end{array}$$

because there is no valid type we can choose for $?$.

Neither *Bool* nor *Nat* nor any other single type works.

We need multiple variants of the identity function for different types.

Introducing polymorphism

Idea

Allow to abstract from types.

Introducing polymorphism

Idea

Allow to abstract from types.

$$\begin{array}{l}\textbf{let } id = \lambda\langle a\rangle\ (x : a)\ .\ x \\ \textbf{in }\ \textbf{if } id\ \langle Bool\rangle\ True\ \textbf{then } id\ \langle Nat\rangle\ 0\ \textbf{else } 1\end{array}$$

We use angle brackets to syntactically distinguish type abstraction and application from term abstraction and application – similar to the way *generics* work in languages like C#.

Polymorphic types

A type abstraction is reflected in the type:

$$\lambda\langle a\rangle\ (x : a)\ .\ x : \mathit{forall}\ a.a \to a$$

Polymorphic types

A type abstraction is reflected in the type:

$$\lambda\langle a\rangle\ (x : a)\ .\ x : \mathit{forall}\ a.a \to a$$

Note that this is quite different from Haskell:

- In Haskell, type abstraction, type application and polymorphism is implicit.

Polymorphic types

A type abstraction is reflected in the type:

$$\lambda\langle a\rangle\ (x : a)\ .\ x : \mathit{forall}\ a.a \to a$$

Note that this is quite different from Haskell:

- In Haskell, type abstraction, type application and polymorphism is implicit.
- In (standard) Haskell, there is a difference between **let**-bound variables (can be polymorphic) and lambda-bound variables (always monomorphic).

What is still missing?

Parameterized types

Even if we have polymorphic types, but there is no general mechanism to abstract from type constructors such as lists.

Inference

All our lambdas carry type information: the language is very verbose. We need to look at how to infer types.

What is still missing?

Data types

The ability for users to define their own data types, together with the associated constructors and elimination principle.

Effects

We haven't said anything about assignments, exceptions, concurrency, interacting with file system, etc.

What is still missing?

General recursion

The simply typed lambda calculus can only express terminating computations – what about arbitrary recursion?

Subtyping and objects

Many programming language allow programmers to organize their code into objects/classes. These classes may be related through *subtyping* (or inheritance) – how can we define these notions?

What is still missing?

On the one hand, this is an impressive list of missing features...

Adding these features requires careful thought.

But no 'fundamental changes' to our definitions or principles.

Recap

- What is the λ-calculus?
- How can we define its semantics?
- How can we define its type system?

Next time

- Embedding and implementing the lambda calculus
- Deep vs shallow embeddings revisited