Lambda Calculus (λ Calculus)

September 19, 2016

Smallest Universal Programming Language

- A single transformation rule (variable substitution)
- A single function definition scheme
- $e ::= x|\lambda x.e|e0e1$

<expression>	:=	<name> | <function> | <application>
<function>	:=	λ <name>.<expression>
<application>	:=	<expression><expression>

As a programming language, sometimes a concrete implementation of lambda calculus also supports predefined constants such as '0' '1' and predefined functions such as '+' '*'; we add parenthesis for clarity

Examples

- $\lambda x.x$ (lambda abstraction: building new function)
- $(\lambda x.x)y$ (application)

What is λ Calculus and Why It Is Important?

1. A mathematical language; A formal computation model for functional programming; a theoretical foundation for the family of functional programming languages.
2. Study interactions between functional abstraction and function applications; study some mathematical properties of effectively computable functions
3. By Alonzo Church in the 1930s
4. In 1920s - 1930s, the mathematicians came up different systems for capturing the general idea of computation:
   - Turing machines – Turing
   - m-recursive functions – Gdel
   - rewrite systems – Post
   - the lambda calculus – Church
   - combinatory logic – Schnfinkel, Curry

   These systems are all computationally equivalent in the sense that each could encode and simulate the others.

The Mathematical Precursor to Scheme

Mathematical formalism to express computation using functions:

- Everything is a function. There are no other primitive types—no integers, strings, cons objects, Booleans ... If you want these things, you must encode them using functions.
- No state or side effects. It is purely functional. Thus we can think exclusively in terms of the substitution model.
- The order of evaluation is irrelevant.
- Only unary (one-argument) functions. No thunks or functions of more than argument.

Implementation in Scheme/DrRacket

- Syntax implemented in Scheme:

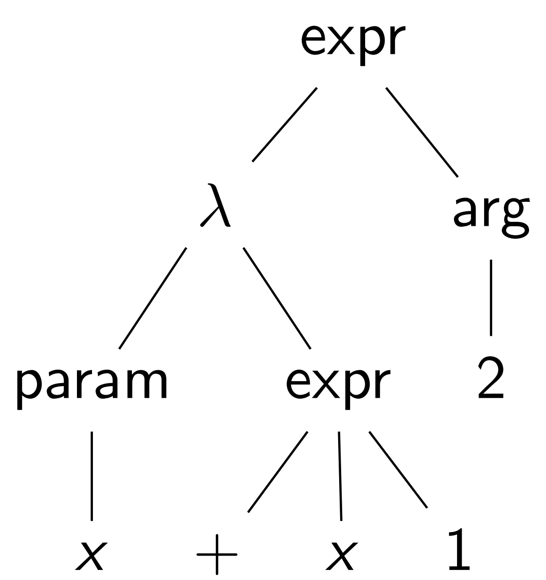
$$
\begin{array}{rcll}
e & \rightarrow & x & \text{Variable} \\
 & \mid & (_x\ \lambda\ (x)\ e\ )_x & \text{a lambda expression} \\
 & \mid & (e\ e) & \text{Application}
\end{array}
$$

$$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$$

Compute $.(2)$ where $.(x) = x + 1$

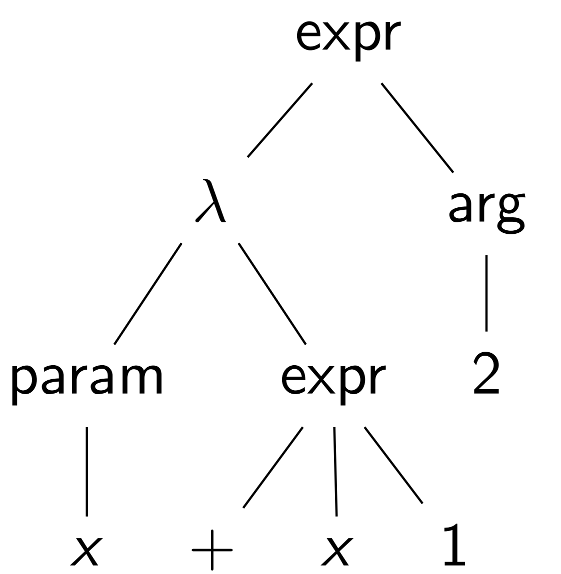
The AST View (Simplified)

$$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$$

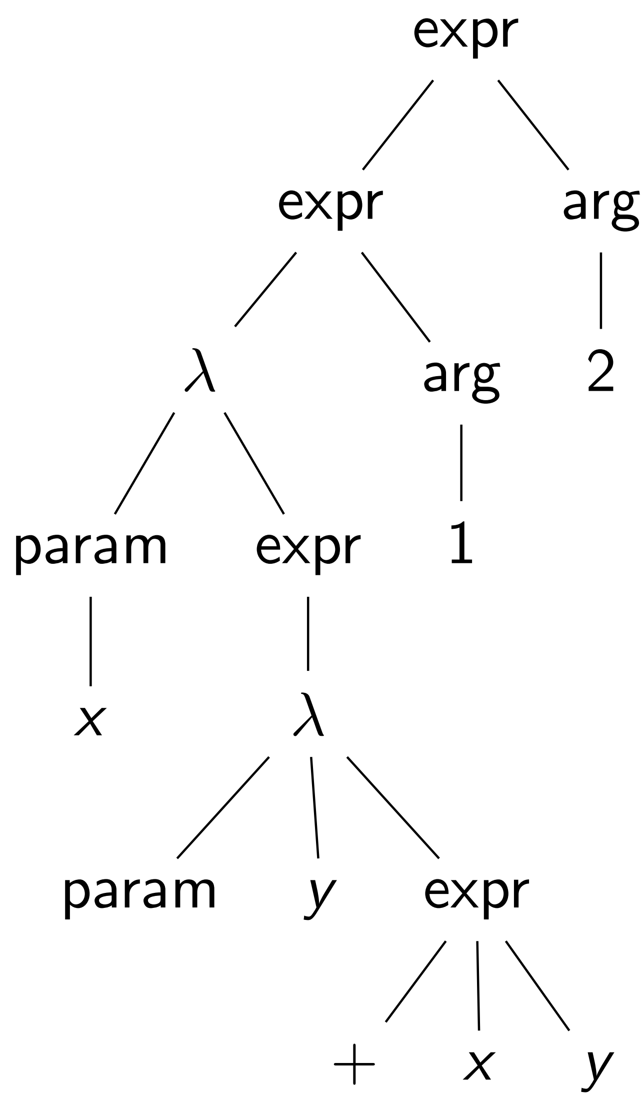


The AST View (Simplified)

$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$



$(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)\ 2)$



Another view: Imperative

λ-expression	Function Definition	Invocation

$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$	`.(x) {`
	`    x + 1`
	`}`
		`.(2) = 2 + 1`

Another view: Imperative

λ-expression	Function Definition	Invocation

$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$	`.(x) {`
	`    x + 1`
	`}`
		`.(2) = 2 + 1`

$(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)\ 2)$		

Another view: Imperative

<table>
<tr><th>λ-expression</th><th>Function Definition</th><th>Invocation</th></tr>
<tr><td>$((_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x\ 2)$</td><td><pre>.(x) {
    x + 1
}</pre></td><td><code>.(2) = 2 + 1</code></td></tr>
<tr><td>$(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)\ 2)$</td><td><pre>.(x) {
   ..(y) {
      x + y
   }
}</pre></td><td></td></tr>
</table>

Another view: Imperative

λ-expression	Function Definition	Invocation

$(({}_{x}\ \lambda\ (x)\ (+\ x\ 1)\ )_{x}\ 2)$	`.(x) {`
	`    x + 1`
	`}`
		`.(2) = 2 + 1`

$((({}_{x}\ \lambda\ (x)\ ({}_{y}\ \lambda\ (y)\ (+\ x\ y)\ )_{y}\ )_{x}\ 1)\ 2)$	`.(x) {`
	`  ..(y) {`
	`     x + y`
	`  }`
	`}`
		`.(1) = ..(y) {`
		`             1 + y`
		`         }`
		`..(2) = 1 + 2`

Another view: Imperative

λ-expression	Function Definition	Invocation

$(({}_{x}\ \lambda\ (x)\ (+\ x\ 1)\ )_{x}\ 2)$	`.(x) {`
	`    x + 1`
	`}`
		`.(2) = 2 + 1`

$((({}_{x}\ \lambda\ (x)\ ({}_{y}\ \lambda\ (y)\ (+\ x\ y)\ )_{y}\ )_{x}\ 1)\ 2)$	`.(x) {`
	`   ..(y) {`
	`      x + y`
	`   }`
	`}`
		`.(1) = ..(y) {`
		`            1 + y`
		`        }`
		`..(2) = 1 + 2`

What have we seen so far...

Anonymous function, functions as first-class elements, inner functions, formal parameters, actual arguments.

Bound and Free Variable: Imperative View

Bound Variable

A bound variable is one which appears in an expression after it has appeared in a λ.

λ-expression	Function Definition	Bound Variables
$(_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x$	`.(x) { x + 1 }`	x

Bound and Free Variable: Imperative View

Bound Variable

A bound variable is one which appears in an expression after it has appeared in a λ.

λ-expression	Function Definition	Bound Variables
$(_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x$	`.(x) { x + 1 }`	x

$(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x$

Bound and Free Variable: Imperative View

Bound Variable

A bound variable is one which appears in an expression after it has appeared in a λ.

λ-expression	Function Definition	Bound Variables

$(_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x$	`.(x) {`
	`    x + 1`
	`}`
		x

$(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x$	`.(x) {`
	`  ..(y) {`
	`      x + y`
	`  }`
	`}`
		x, y

Bound and Free Variable: Imperative View

Bound Variable

A bound variable is one which appears in an expression after it has appeared in a λ.

λ-expression	Function Definition	Bound Variables

$(_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x$	`.(x) {`
	`    x + 1`
	`}`
		x

$(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x$	`.(x) {`
	`  ..(y) {`
	`      x + y`
	`  }`
	`}`
		x, y

$(_y\ \lambda\ (y)\ (+\ x\ y)\ )_y$		

Bound and Free Variable: Imperative View

Bound Variable

A bound variable is one which appears in an expression after it has appeared in a λ.

λ-expression	Function Definition	Bound Variables

$(_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x$	`.(x) {`
	`    x + 1`
	`}`
		x

$(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x$	`.(x) {`
	`  ..(y) {`
	`      x + y`
	`  }`
	`}`
		x, y

$(_y\ \lambda\ (y)\ (+\ x\ y)\ )_y$	`..(y) {`
	`  x + y`
	`}`
		y

Free Variables

Any variable that is not bound is free.

Formal Semantics of the Language

- $((_x\ \lambda\ (x)\ e_1\ )_x\ e_2)$: Evaluate the expression e_1 by replacing every ("free") occurrences of x in e_1 by e_2. I.e., $e_1[x \mapsto e_2]$

  (β-reduction)

Formal Semantics of the Language

- $((_x\ \lambda\ (x)\ e_1\ )_x\ e_2)$: Evaluate the expression e_1 by replacing every ("free") occurrences of x in e_1 by e_2. I.e., $e_1[x \mapsto e_2]$

  (β-reduction)

$(\ (_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)$

Formal Semantics of the Language

- $((_x\ \lambda\ (x)\ e_1\ )_x\ e_2)$: Evaluate the expression e_1 by replacing every ("free") occurrences of x in e_1 by e_2. I.e., $e_1[x \mapsto e_2]$

  (β-reduction)

$(\ (_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)$

Formal Semantics of the Language

- $((_x\ \lambda\ (x)\ e_1\ )_x\ e_2)$: Evaluate the expression e_1 by replacing every ("free") occurrences of x in e_1 by e_2. I.e., $e_1[x \mapsto e_2]$

  (β-reduction)

$(\ (_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)$

$(_y\ \lambda\ (y)\ (+\ x\ y)\ )_y[x \mapsto 1]$

Formal Semantics of the Language

- $((_x\ \lambda\ (x)\ e_1\ )_x\ e_2)$: Evaluate the expression e_1 by replacing every ("free") occurrences of x in e_1 by e_2. I.e., $e_1[x \mapsto e_2]$

  (β-reduction)

$$(\ (_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ (+\ x\ y)\ )_y\ )_x\ 1)$$

$$(_y\ \lambda\ (y)\ (+\ x\ y)\ )_y[x \mapsto 1]$$

$$(_y\ \lambda\ (y)\ (+\ 1\ y)\ )_y$$

How about

$$((((_x \lambda (x) (_y \lambda (y) (_x \lambda (x) (+ x y))_x)_y)_x 1) 2) 3))$$

How about

$$(\; (\; (\; (_x \; \lambda \; (x) \; (_y \; \lambda \; (y) \; (_x \; \lambda \; (x) \; (+ \; x \; y) \;)_x \;)_y \;)_x \; 1) \; 2) \; 3) \;)$$

```
.(x) {
  ..(y) {
    ...(x) {
       x + y
    }
  }
}
```

How about

$$((((_x \lambda (x) (_y \lambda (y) (_x \lambda (x) (+ x y))_x)_y)_x 1) 2) 3))$$

```
.(x) {
  ..(y) {
    ...(x) {
       x + y
    }
  }
}
```

```
.(1) = ..(y) {
          ...(x) {
             x + y
          }
       }
```

Resolving Name Capture

α-Conversion

Rename variables

$$((((_x \lambda (x) (_y \lambda (y) (_x \lambda (x) (+ x\ y))_x)_y)_x 1) 2) 3))$$

Resolving Name Capture

α-Conversion

Rename variables

$$((((_x \; \lambda \; (x) \; (_y \; \lambda \; (y) \; (_x \; \lambda \; (x) \; (+ \; x \; y) \;)_x \;)_y \;)_x \; 1) \; 2) \; 3) \;)$$

$$((((_x \; \lambda \; (x) \; (_y \; \lambda \; (y) \; (_z \; \lambda \; (z) \; (+ \; z \; y) \;)_z \;)_y \;)_x \; 1) \; 2) \; 3) \;)$$

Currying

Pure lambda calculus pairs one variable with one λ

- Functions with many parameters
  $(_{x\ y}\ \lambda\ (x\ y)\ e\ )_{x\ y}$: two formal parameters x and y
- Semantically equivalent expression: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ e\ )_y\ )_x$

Concept introduced by Haskell Curry.

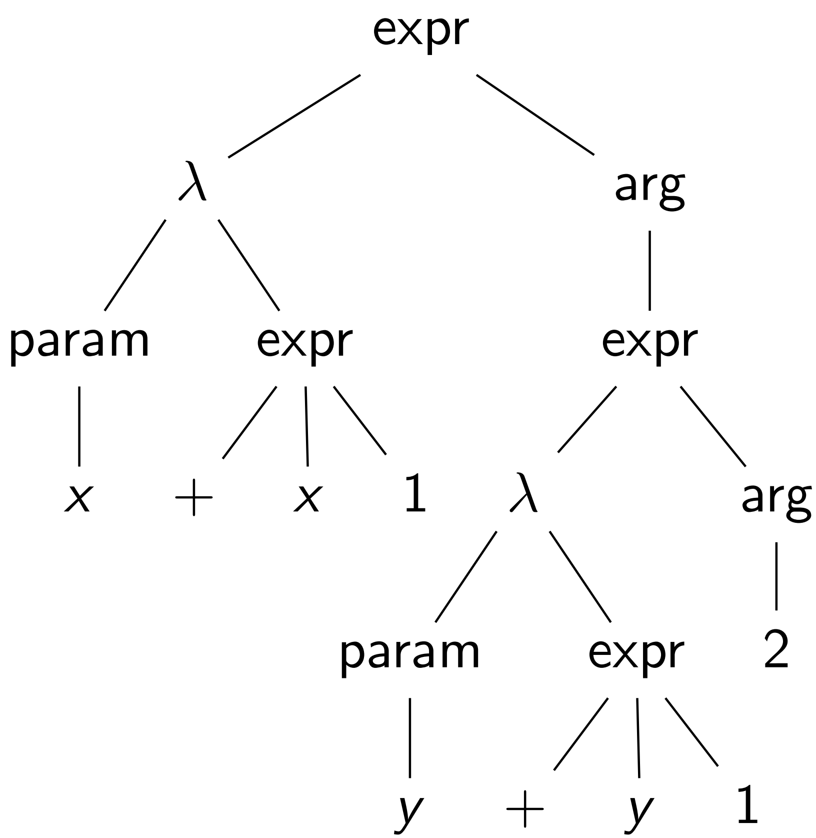
More on this in subsequent lectures.

Examples: AST View

$$(\ {\color{green}(_x} \ \lambda \ (x) \ (+ \ x \ 1) \ {\color{green})_x} (\ {\color{green}(_y} \ \lambda \ (y) \ (+ \ y \ 1) \ {\color{green})_y} \ 2 \ ) \ )$$

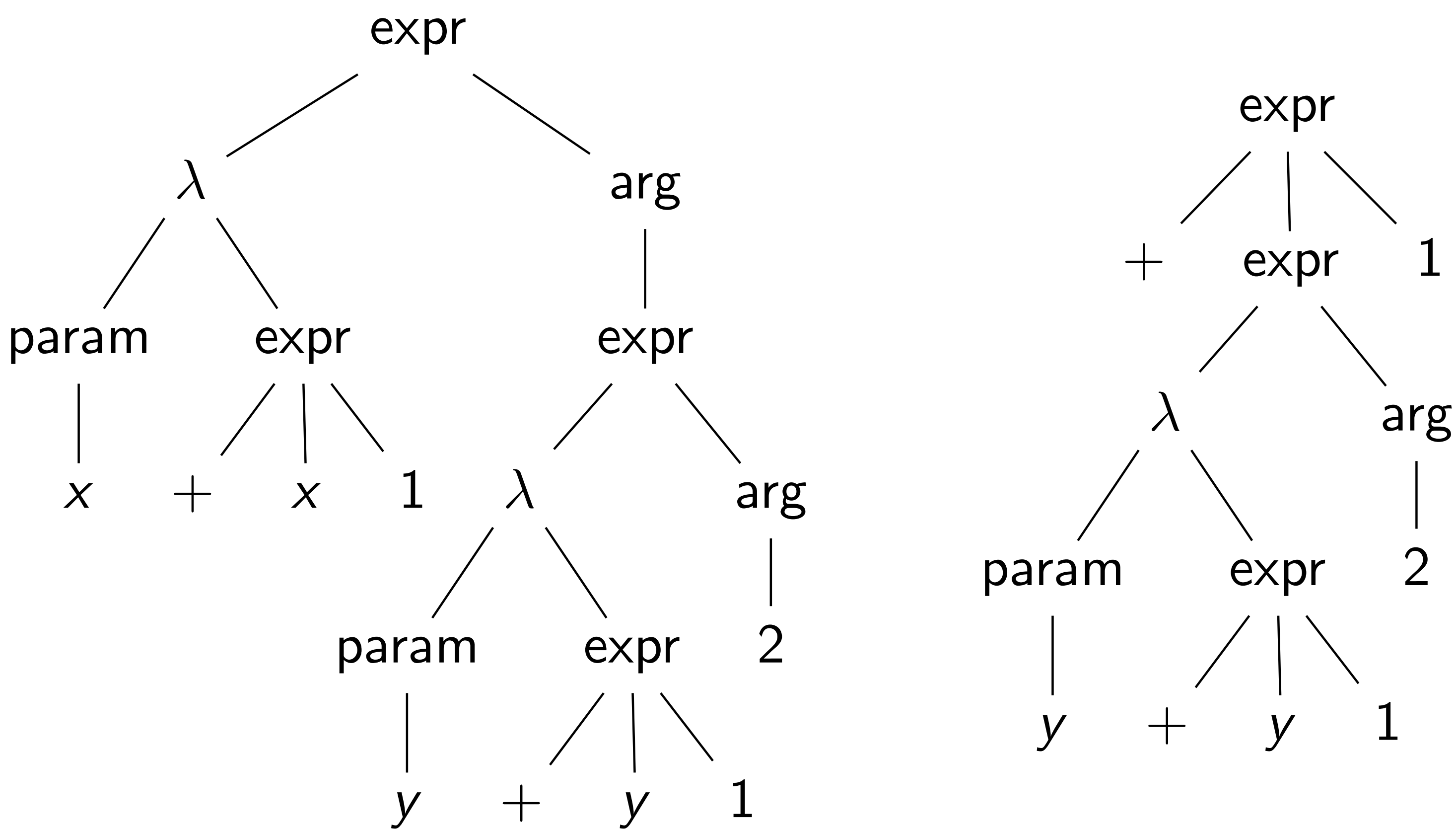
Examples: AST View

$$(\ (_x \ \lambda \ (x) \ (+ \ x \ 1) \ )_x (\ (_y \ \lambda \ (y) \ (+ \ y \ 1) \ )_y \ 2 \ ) \ )$$



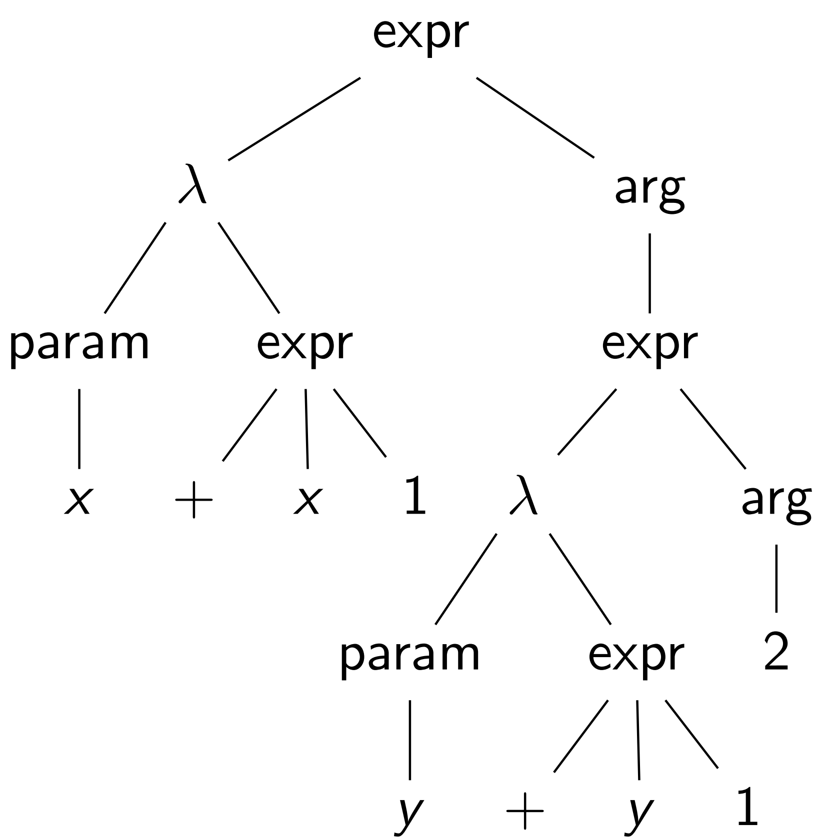
Examples: AST View

$$(\; (_x \; \lambda \, (x) \; (+ \; x \; 1) \;)_x (\; (_y \; \lambda \, (y) \; (+ \; y \; 1) \;)_y \; 2 \;) \;)$$

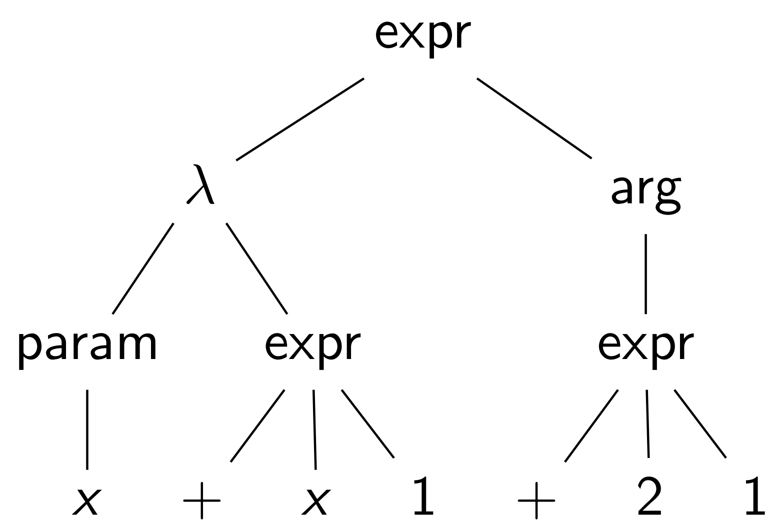


Examples: AST View

$$(\; (_x \; \lambda \; (x) \; (+ \; x \; 1) \;)_x (\; (_y \; \lambda \; (y) \; (+ \; y \; 1) \;)_y \; 2 \;) \;)$$



OR ...



Ordering in Evaluation

$(\; (_x \; \lambda \; (x) \; (+ \; x \; 1) \;)_x (\; (_y \; \lambda \; (y) \; (+ \; y \; 1) \;)_y \; 2 \;) \;)$

After β-reduction

Either $(+ \; (\; (_y \; \lambda \; (y) \; (+ \; y \; 1) \;)_y \; 2 \;) \; 1)$

Or $(\; (_x \; \lambda \; (x) \; (+ \; x \; 1) \;)_x (+ \; 2 \; 1) \;)$

Recap

- $(_x\ \lambda\ (x)\ e\ )_x$: a lambda expression representing definition of function
- $(\ (_x\ \lambda\ (x)\ e\ )_x\ p)$: a lambda expression representing application of a function.
  - Formal parameter: x
  - Actual argument: p
  - Computation: $e[x \mapsto p]$, replace free occurrences of x in e with p (β-reduction)
- Order of β-reduction does not impact the result if each β-reduction terminates

$$(\ (_x\ \lambda\ (x)\ (+\ x\ 1)\ )_x(\ (_y\ \lambda\ (y)\ (+\ y\ 1)\ )_y\ 2\ )\ )$$

Examples

What is the result of

$$(\;(_x\; \lambda\, (x)\; x\;)_x\; (_y\; \lambda\, (y)\; y\;)_y\;)$$

Examples

What is the result of

$(\ (_x\ \lambda\ (x)\ x\ )_x\ (_y\ \lambda\ (y)\ y\ )_y\ )$

- $(_x\ \lambda\ (x)\ x\ )_x$: identify function. The function applied to any entity returns the entity itself.

Adding and subtracting 0 from an arith. expression returns the expression.
Multiplying and dividing by 1

Examples

- $(_x\ \lambda\ (x)\ (x\ x)\ )_x$: self application function. The function when applied to an entity, applies the entity to itself.
  What is the result of $((_x\ \lambda\ (x)\ (x\ x)\ )_x\ 3)$?

  What is the result of $(\ (_x\ \lambda\ (x)\ (x\ x)\ )_x\ (_y\ \lambda\ (y)\ y\ )_y\ )$

Examples: Function Application

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$: Application of function f on x.

What about
$(\ ((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f\ (_y\ \lambda\ (y)\ y\ )_y)\ (_z\ \lambda\ (z)\ (z\ z)\ )_z\ )$?

Examples: Function Application

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$: Application of function f on x.

What about
$(\ ((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f\ (_y\ \lambda\ (y)\ y\ )_y)\ (_z\ \lambda\ (z)\ (z\ z)\ )_z\ )$?

let $g = (_y\ \lambda\ (y)\ y\ )_y$ and $v = (_z\ \lambda\ (z)\ (z\ z)\ )_z$. Then,

Examples: Function Application

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$: Application of function f on x.

What about
$(\ ((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f\ (_y\ \lambda\ (y)\ y\ )_y)\ (_z\ \lambda\ (z)\ (z\ z)\ )_z\ )$?

let $g = (_y\ \lambda\ (y)\ y\ )_y$ and $v = (_z\ \lambda\ (z)\ (z\ z)\ )_z$. Then,

result is $(g\ v) = (\ (_y\ \lambda\ (y)\ y\ )_y\ v) =$

Examples: Function Application

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$: Application of function f on x.

What about
$(\ ((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f\ (_y\ \lambda\ (y)\ y\ )_y)\ (_z\ \lambda\ (z)\ (z\ z)\ )_z\ )$?

let $g = (_y\ \lambda\ (y)\ y\ )_y$ and $v = (_z\ \lambda\ (z)\ (z\ z)\ )_z$. Then,

result is $(g\ v) = (\ (_y\ \lambda\ (y)\ y\ )_y\ v) =\ v =$

Examples: Function Application

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$: Application of function f on x.

What about
$(\ ((_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f\ (_y\ \lambda\ (y)\ y\ )_y)\ (_z\ \lambda\ (z)\ (z\ z)\ )_z\ )$?

let $g = (_y\ \lambda\ (y)\ y\ )_y$ and $v = (_z\ \lambda\ (z)\ (z\ z)\ )_z$. Then,

result is $(g\ v) = (\ (_y\ \lambda\ (y)\ y\ )_y\ v) = \ v = (_z\ \lambda\ (z)\ (z\ z)\ )_z$

Syntax Revisited

$$
\begin{array}{rcll}
e & \rightarrow & x & \text{Variable} \\
 & | & (_x\ \lambda\ (x)\ e\ )_x & \text{a lambda expression} \\
 & | & (e\ e) & \text{Application}
\end{array}
$$

What about the data? Boolean, Integers, ...

Natural Numbers (Church Numerals)

Encoding of numbers: 0, 1, 2, . . . , as functions such that their semantics follows the natural number semantics.

Intuition: The number n means how many times one can do certain operation.

Encoding Natural Numbers

zero $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ x\ )_x\ )_f$

one $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$

two $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f$

n $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ \ldots(f\ x)\ldots))\ )_x\ )_f$

Assume f is operation and x is the object on which the operation is done.

Encoding Natural Numbers

zero $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ x\ )_x\ )_f$

one $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$

two $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ (f\ x))\ )_x\ )_f$

n $(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ \ldots(f\ x)\ldots))\ )_x\ )_f$

Assume f is operation and x is the object on which the operation is done.

E.g.: f is adding '1' to the list

E.g.: x is an empty list

Then,
meaning of zero is empty list ( )
meaning of one is (1)
meaning of two is (11)
meaning of three is (111)

Encoding Natural Numbers

A natural number is represented by the number of application of some function on some entity.

A natural number function takes two arguments (function and entity on which the function is to be applied).

Example

What is the semantics of

- $((two\ g)\ z)$: two applications of g on z.

$$((({}_{f}\ \lambda\ (f)\ ({}_{x}\ \lambda\ (x)\ (f\ (f\ x))\ )_{x}\ )_{f}\ g)\ z) = (g\ (g\ z))$$

- $((n\ g)\ z)$: n applications of g on z, where n is a natural number.
- $({}_{z}\ \lambda\ (z)\ ((n\ g)\ z)\ )_{z}$: n applications of g on the formal parameter z, where n is a natural number. This result is a function (z if the formal parameter of the function).

Encoding Natural Number

- successor function: succinct representation of any number
- addition
- multiplication
- subtraction

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

n: the number whose successor we want to compute.
$((n\ f)\ x)$: n applications of the function f on x, i.e., $(f^n x)$. Therefore, $(f\ ((n\ f)\ x))$ is $(f\ (f^n\ x))$, which is representation of $n+1$.

Successor Function

succ: $(_n\ \lambda\,(n)\ (_f\ \lambda\,(f)\ (_x\ \lambda\,(x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$(succ\ zero)$

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$(succ\ zero)$
$= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero)$

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$$
\begin{aligned}
& (succ\ zero) \\
= & ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero) \\
= & \qquad\qquad (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f
\end{aligned}
$$

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$$
\begin{aligned}
&(succ\ zero)\\
&= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero)\\
&= \qquad\qquad (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f
\end{aligned}
$$

$$(zero\ f) = ((_g\ \lambda\ (g)\ (_y\ \lambda\ (y)\ y\ )_y\ )_g\ f) = (_y\ \lambda\ (y)\ y\ )_y$$

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$$
\begin{aligned}
&(succ\ zero)\\
&= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero)\\
&= \qquad\qquad (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f
\end{aligned}
$$

$(zero\ f) = ((_g\ \lambda\ (g)\ (_y\ \lambda\ (y)\ y\ )_y\ )_g\ f) = (_y\ \lambda\ (y)\ y\ )_y$

Therefore,
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f$

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$$\begin{aligned}
&(succ\ zero)\\
&= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero)\\
&= \qquad\qquad (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f
\end{aligned}$$

$$(zero\ f) = ((_g\ \lambda\ (g)\ (_y\ \lambda\ (y)\ y\ )_y\ )_g\ f) = (_y\ \lambda\ (y)\ y\ )_y$$

Therefore,

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f =$

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((_y\ \lambda\ (y)\ y\ )_y\ x))\ )_x\ )_f$

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$$
\begin{aligned}
&(succ\ zero)\\
&= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero)\\
&= \qquad (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f
\end{aligned}
$$

$$(zero\ f) = ((_g\ \lambda\ (g)\ (_y\ \lambda\ (y)\ y\ )_y\ )_g\ f) = (_y\ \lambda\ (y)\ y\ )_y$$

Therefore,

$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f\ =$
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((_y\ \lambda\ (y)\ y\ )_y\ x))\ )_x\ )_f\ =$
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f$

Successor Function

succ: $(_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n$

$(succ\ zero)$
$= ((_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((n\ f)\ x))\ )_x\ )_f\ )_n\ zero)$
$= \qquad (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f$

$(zero\ f) = ((_g\ \lambda\ (g)\ (_y\ \lambda\ (y)\ y\ )_y\ )_g\ f) = (_y\ \lambda\ (y)\ y\ )_y$

Therefore,
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((zero\ f)\ x))\ )_x\ )_f =$
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ ((_y\ \lambda\ (y)\ y\ )_y\ x))\ )_x\ )_f =$
$(_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ (f\ x)\ )_x\ )_f =$
one

How about?

$(\mathit{succ}\ (\mathit{succ}\ \mathit{zero}))$

More with successors

$$(_m\ \lambda\ (m)\ (_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ ((((m\ succ)\ n)\ f)\ x)\ )_x\ )_f\ )_n\ )_m$$

More with successors

$(_m\ \lambda\ (m)\ (_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ ((((m\ succ)\ n)\ f)\ x)\ )_x\ )_f\ )_n\ )_m$

Apply *succ* m times to create a function that is applied n. E.g., $m = 2$ and $n = 3$ results in *succ* of *succ* of 3, which is 5.

More with successors

add: $(_m\ \lambda\ (m)\ (_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ ((((m\ succ)\ n)\ f)\ x)\ )_x\ )_f\ )_n\ )_m$

Apply *succ* *m* times to create a function that is applied *n*. E.g., $m = 2$ and $n = 3$ results in *succ* of *succ* of 3, which is 5.

More with successors

add: $(_m\ \lambda\ (m)\ (_n\ \lambda\ (n)\ (_f\ \lambda\ (f)\ (_x\ \lambda\ (x)\ ((((m\ succ)\ n)\ f)\ x)\ )_x\ )_f\ )_n\ )_m$

Apply *succ* m times to create a function that is applied n. E.g., $m = 2$ and $n = 3$ results in *succ* of *succ* of 3, which is 5.

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ \ t)\ \ e)\ )_e\ )_t\ )_c$

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ \ t)\ \ e)\ )_e\ )_t\ )_c$

$(((ite\ \ true)\ \ s_1)\ \ s_2)$

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c$

$$(((ite\ true)\ s_1)\ s_2) =$$
$$((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2)$$

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ \ t)\ e)\ )_e\ )_t\ )_c$

$$(((ite\ \ true)\ s_1)\ s_2) =$$
$$((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ \ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) =$$
$$(((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ \ t)\ e)\ )_e\ )_t\ s_1)\ s_2)$$

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ \ t)\ e)\ )_e\ )_t\ )_c$

$$
\begin{aligned}
&(((ite\ \ true)\ s_1)\ s_2) = \\
&((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ \ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) = \\
&\qquad (((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ \ t)\ e)\ )_e\ )_t\ s_1)\ s_2) = \\
&\qquad\qquad ((_e\ \lambda\ (e)\ ((true\ s_1)\ e)\ )_e\ s_2)
\end{aligned}
$$

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ \ t)\ e)\ )_e\ )_t\ )_c$

$$
\begin{aligned}
&(((ite\ \ true)\ s_1)\ s_2) = \\
&((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ \ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) = \\
&\qquad (((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ \ t)\ e)\ )_e\ )_t\ s_1)\ s_2) = \\
&\qquad\qquad ((_e\ \lambda\ (e)\ ((true\ \ s_1)\ e)\ )_e\ s_2) = \\
&\qquad\qquad\qquad ((true\ \ s_1)\ s_2)
\end{aligned}
$$

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c$

$$
\begin{aligned}
&(((ite\ true)\ s_1)\ s_2) = \\
&((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) = \\
&\qquad (((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ t)\ e)\ )_e\ )_t\ s_1)\ s_2) = \\
&\qquad\qquad ((_e\ \lambda\ (e)\ ((true\ s_1)\ e)\ )_e\ s_2) = \\
&\qquad\qquad\qquad ((true\ s_1)\ s_2) = \\
&(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x\ s_1)\ s_2)
\end{aligned}
$$

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c$

$$
\begin{aligned}
&(((ite\ true)\ s_1)\ s_2) = \\
&((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) = \\
&\qquad (((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ t)\ e)\ )_e\ )_t\ s_1)\ s_2) = \\
&\qquad\qquad ((_e\ \lambda\ (e)\ ((true\ s_1)\ e)\ )_e\ s_2) = \\
&\qquad\qquad\qquad ((true\ s_1)\ s_2) = \\
&(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x\ s_1)\ s_2) = \\
&\qquad ((_y\ \lambda\ (y)\ s_1\ )_y\ s_2)
\end{aligned}
$$

Booleans

true: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x$ Select the first argument

false: $(_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ y\ )_y\ )_x$ Select the second argument

ite: $(_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c$

$$
\begin{aligned}
&(((ite\ true)\ s_1)\ s_2) = \\
&((((_c\ \lambda\ (c)\ (_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((c\ t)\ e)\ )_e\ )_t\ )_c\ true)\ s_1)\ s_2) = \\
&\qquad (((_t\ \lambda\ (t)\ (_e\ \lambda\ (e)\ ((true\ t)\ e)\ )_e\ )_t\ s_1)\ s_2) = \\
&\qquad\qquad ((_e\ \lambda\ (e)\ ((true\ s_1)\ e)\ )_e\ s_2) = \\
&\qquad\qquad\qquad ((true\ s_1)\ s_2) = \\
&(((_x\ \lambda\ (x)\ (_y\ \lambda\ (y)\ x\ )_y\ )_x\ s_1)\ s_2) = \\
&\qquad ((_y\ \lambda\ (y)\ s_1\ )_y\ s_2) = s_1
\end{aligned}
$$

Boolean Operators

not a: if a then false else true. $(((ite\ a)\ false)\ true)$

a b : if a then b else false. $(((ite\ a)\ b)\ false)$

What is the adequate set of operators for boolean logic?

Recursion

- λ-calculus does not allow recursive definition, i.e., a definition of a function must not include the name of the function.
- Relies on fixed point characterization of recursions to realize the results of recursions.

Do we program in lambda calculus

No

The objective to learn about Lambda Calculus:

- Better understanding of functional computation and functional programming
- Design of new languages/new features to existing languages

Review and Further Reading

- Concepts: Lambda abstraction and function application, high order functions
- bound and free variables
- currying
- $\beta-$reduction and $\alpha-$conversion
- church encoding

Further reading: pass by name, pass by value, lazy evaluation

- Programming Languages: Lambda Calculus - 1
  `https://www.youtube.com/watch?v=v1IlyzxP6Sg`
- Programming Languages: Lambda Calculus - 2
  `https://www.youtube.com/watch?v=Mg1pxUKeWCk`