COMPSCI 3MI3 - Principles of Programming Languages

The Lambda Calculus

J. Carette

McMaster University

Fall 2023

Adapted from "Types and Programming Languages" by Benjamin C. Pierce and Nick Moore's material.

Computation my Friends! Computation!

In the 1960s, Peter Landin observed that complex programming languages can be understood by capturing their essential mechanisms as a small core calculus.

- The core language used by Landin was λ-**Calculus**
  - Developed in the 1920s by Alonzo Church.
  - Reduces *all* computation to **function definition** and **application**.

The strength of λ-Calculus comes from it's *simplicity* and its capacity for **formal reasoning**.

λ-Calculus Syntax

Untyped λ-Calculus is comprised of only 3 terms!

$$
\begin{aligned}
\langle t \rangle ::= &\ \langle x \rangle \\
\mid &\ \lambda \langle x \rangle \,.\, \langle t \rangle \\
\mid &\ \langle t \rangle\ \langle t \rangle
\end{aligned}
$$

These terms are:

- variables
- λ abstraction
- application.

Kinds of Syntax

- **Concrete Syntax**
  - The "surface syntax" used by programmers
- **Abstract Syntax**
  - Often a **tree**, sometimes a **Directed Acyclic Graph** (DAG)
  - The "internal representation" that's nicer for programs to compute with.

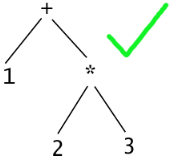
Concrete to Abstract:

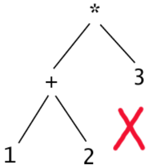
- Nice-to-have but redundant constructs removed (aka **desugaring**)
- Missing information is added (type inference and **elaboration**)

AST

Abstract syntax is an excellent way of visualizing a program's structure, especially in resolving operator precedence.

- For example, under BEDMAS, the expression $1 + 2 * 3$ would be the left diagram, not the right diagram:



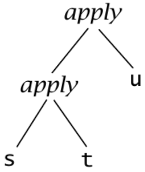


BEDMAS trees are evaluated leaf-first, however λ expressions may be evaluated using a number of different strategies.

ASTs of λ-Calculus

To reduce redundant parentheses in our concrete syntax for λ-Calculus:

- Application will be **left-associative**. That is, `s t u` is interpretted as:



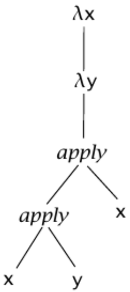
- i.e. `(s t) u`

Scope of λ Operator

The abstraction operator λ is taken to extend to the right as far as possible. For the following expression:

- λ`x.`λ`y.x y x`, aka λ`x.(`λ`y.(x y) x)`, aka

We would construct an AST:



Free vs Bound Variables

In predicate calculus, distinction between **free** and **bound** variables.

$$\exists x \mid x \neq y \tag{1}$$

- x is **bound** by the existential quantifier.
- y is not bound by a quantifier and is therefore **free**

$$(\lambda x.x\,y)\,x \tag{2}$$

- The first occurance of x is **bound**.
- Both y and the second occurrance of x are **free**.

Only One Evaluation Rule

These terms reduce by **substituting** the abstracted variable with the term applied to the function. In other words:

$$(\lambda x.t_1)\ t_2 \to [x \mapsto t_2]\ t_1 \tag{3}$$

- A λ expression which may be simplified is known as a **redex**, or *reducible expression*.
- Called**beta-reduction**, aka β-reduction.

Using All our Substitutions

$[x \mapsto t_2]\, t_1$ stands for "the term obtained by the replacement of all free occurances of x in t_1 by t_2. Examples:

$$(\lambda x.x)\, y \to y \tag{4}$$

$$(\lambda x.x\, (\lambda x.x))\, (u\, r) \to u\, r\, (\lambda x.x) \tag{5}$$

Our Test Expression

To examine strategies, we will use a running example expression:

$$(\lambda x.x)\,((\lambda x.x)\,(\lambda z.(\lambda x.x)\,z)) \tag{6}$$

- $\lambda x.x$ is effectively an **identity function**, so we write it as id.

$$id\,(id\,(\lambda z.id\,z)) \tag{7}$$

The above expression has three redexes:

$$\color{red}{id\,(id\,(\lambda z.id\,z))} \tag{8}$$

$$id\,(\color{red}{id\,(\lambda z.id\,z)}) \tag{9}$$

$$id\,(id\,(\lambda z.\color{red}{id\,z})) \tag{10}$$

The Worst Strategy Ever

Under **Full Beta-Reduction**, the redexes may be reduced in any order.

- not deterministic.

Normal Order

Normal order begins with the leftmost, outermost redex, and proceeds until there are no more redexes to evaluate.

$$
\begin{array}{ll}
 & id\ (id\ (\lambda z.id\ z)) \\
\rightarrow & id\ (\lambda z.id\ z) \\
\rightarrow & \lambda z.id\ z \\
\rightarrow & \lambda z.z \\
\not\rightarrow &
\end{array}
$$

Call By Name

The **call by name** strategy is more restrictive than normal order. You can't evaluate anything under a lambda.

$$\begin{array}{rl} & \color{red}{id\ (id\ (\lambda z.id\ z))} \\ \rightarrow & \color{red}{id\ (\lambda z.id\ z)} \\ \rightarrow & \lambda z.id\ z \\ \not\rightarrow & \end{array}$$

In this case, $\lambda z.id\ z$ is considered a **normal form**.

Haskell

Haskell uses **call by need**, which is an optimization of call by name.

- To avoid re-evaluation, expressions are kept as a graph that joins identical expressions,
- Further, once an expression is evaluated, the expression is replaced by its value in the AST.
- thus only need to be evaluated *once*.
- is a reduction relation on syntax **graphs**, rather than syntax **trees**.

Call By Value

Most languages use **call by value**, where only the outermost redexes are reduced, and a redex is only reduced when the right-hand-side has already been reduced to a value.

- Here, as elsewhere, a value is a term in normal form.

$$
\begin{array}{rl}
 & id\ (id\ (\lambda z.id\ z)) \\
\to & id\ (\lambda z.id\ z) \\
\to & \lambda z.id\ z \\
\not\to &
\end{array}
$$

Bool

Can we even do Booleans? (Want to reconstruct UAE).

$$\texttt{tru} = \lambda t.\lambda f.t \tag{11}$$

$$\texttt{fls} = \lambda t.\lambda f.f \tag{12}$$

Bool as 2-argument functions?!?

This will make more sense once we consider `if then else`:

$$\texttt{ifte} = \lambda c.\lambda th.\lambda el.\ c\ th\ el \tag{13}$$

$$\begin{array}{ll} & \underline{\text{With } c = \texttt{tru}} \\ & (\lambda c.\lambda th.\lambda el.\ c\ th\ el)\ \texttt{tru}\ u\ v \\ \to & (\lambda th.\lambda el.\ \texttt{tru}\ th\ el)\ u\ v \\ \to & (\lambda el.\ \texttt{tru}\ u\ el)\ v \\ \to & \texttt{tru}\ u\ v \\ \to & (\lambda t.\lambda f.t)\ u\ v \\ \to & (\lambda f.u)\ v \\ \to & u \\ \not\to & \end{array}$$

$$\begin{array}{ll} & \underline{\text{With } c = \texttt{fls}} \\ & (\lambda c.\lambda th.\lambda el.\ c\ th\ el)\ \texttt{fls}\ u\ v \\ \to & (\lambda th.\lambda el.\ \texttt{fls}\ th\ el)\ u\ v \\ \to & (\lambda el.\ \texttt{fls}\ u\ el)\ v \\ \to & \texttt{fls}\ u\ v \\ \to & (\lambda t.\lambda f.f)\ u\ v \\ \to & (\lambda f.f)\ v \\ \to & v \\ \not\to & \end{array}$$

Boolean Operators

Extending the λ-Calculus vs UAE:

- UAE: add additional terms and evaluation rules.
  - Makes recursion and induction longer
- λ-Calculus: define terms *in* the language
  - `tru` and `fls` are not terms, but **labels** for λ expressions *that were already valid terms!*

Conservative Extension

Consider two theories, T_1 and T_2. We say that T_2 is a **conservative extension** of T_1 if:

- Every theorem of T_1 is a theorem of T_2
- Any theorem of T_2 in the language of T_1 is already a theorem of T_1.

i.e. Booleans are a conservative extension of the λ-Calculus Why useful? All

properties of the λ-Calculus remain true of conservative extensions.

Boolean And I

More operations.

$$\texttt{and} = \lambda b.\lambda c.\ b\ c\ fls \tag{14}$$

With input $\texttt{tru}\ \texttt{tru}$

$$\begin{array}{ll}
 & (\lambda b.\lambda c.\ b\ c\ \texttt{fls})\ \texttt{tru}\ \texttt{tru} \\
\to & (\lambda c.\ \texttt{tru}\ c\ \texttt{fls})\ \texttt{tru} \\
\to & \texttt{tru}\ \texttt{tru}\ \texttt{fls} \\
\to & (\lambda t.\lambda f.t)\ \texttt{tru}\ \texttt{fls} \\
\to & (\lambda f.\texttt{tru})\ \texttt{fls} \\
\to & \texttt{tru} \\
\not\to &
\end{array}$$

With input $\texttt{tru}\ \texttt{fls}$

$$\begin{array}{ll}
 & (\lambda b.\lambda c.\ b\ c\ \texttt{fls})\ \texttt{tru}\ \texttt{fls} \\
\to & (\lambda c.\ \texttt{tru}\ c\ \texttt{fls})\ \texttt{fls} \\
\to & \texttt{tru}\ \texttt{fls}\ \texttt{fls} \\
\to & (\lambda t.\lambda f.t)\ \texttt{fls}\ \texttt{fls} \\
\to & (\lambda f.\texttt{fls})\ \texttt{fls} \\
\to & \texttt{fls} \\
\not\to &
\end{array}$$

Boolean And II

<u>With input `fls tru`</u>

$$
\begin{array}{ll}
 & (\lambda b.\lambda c.\ b\ c\ \texttt{fls})\ \texttt{fls}\ \texttt{tru} \\
\to & (\lambda c.\ \texttt{fls}\ c\ \texttt{fls})\ \texttt{tru} \\
\to & \texttt{fls}\ \texttt{tru}\ \texttt{fls} \\
\to & (\lambda t.\lambda f.f)\ \texttt{tru}\ \texttt{fls} \\
\to & (\lambda f.f)\ \texttt{fls} \\
\to & \texttt{fls} \\
\not\to &
\end{array}
$$

<u>With input `fls fls`</u>

$$
\begin{array}{ll}
 & (\lambda b.\lambda c.\ b\ c\ \texttt{fls})\ \texttt{fls}\ \texttt{fls} \\
\to & (\lambda c.\ \texttt{fls}\ c\ \texttt{fls})\ \texttt{fls} \\
\to & \texttt{fls}\ \texttt{fls}\ \texttt{fls} \\
\to & (\lambda t.\lambda f.f)\ \texttt{fls}\ \texttt{fls} \\
\to & (\lambda f.f)\ \texttt{fls} \\
\to & \texttt{fls} \\
\not\to &
\end{array}
$$

Pairs

$$\texttt{pair} = \lambda f.\lambda s.\lambda b.\, b\, f\, s \tag{15}$$

$$\texttt{fst} = \lambda p.\, p\, \texttt{tru} \tag{16}$$

$$\texttt{snd} = \lambda p.\, p\, \texttt{fls} \tag{17}$$

- b is used to select between f and s
- `fst` and `snd` merely apply `tru` and `fls` respectively.
- Since `tru` selects the first argument, it also selects the first term in the pair.
- Likewise for `fls`

Let's code it in Haskell!

Church Numerals

Natural numbers are quite similar to Peano arithmetic:

$$c_0 = \lambda s.\lambda z.\ z \tag{18}$$

$$c_1 = \lambda s.\lambda z.\ s\ z \tag{19}$$

$$c_2 = \lambda s.\lambda z.\ s\ (s\ z) \tag{20}$$

$$c_3 = \lambda s.\lambda z.\ s\ (s\ (s\ z)) \tag{21}$$

$$\vdots$$

Church numerals take two arguments, a successor s and a zero term z **representation**.

Clash?

You might have noticed that c_0 has the same definition as `fls`.

- This is sometimes called a **pun** in computer science.
- The same thing occurs in lower level languages, where the interpretation of a sequence of bits is context dependant.
- In C, the bit arrangement 0x00000000 corresponds to:
  - Zero (Integer)
  - False (Boolean)
  - "\0\0\0\0" (Character Array)

This is not a *good thing*.

Succ-ess!

Adding one:

$$\mathtt{succ} = \lambda n.\lambda s.\lambda z.\ s\,(n\ s\ z) \tag{22}$$

Successor of Two

$$\begin{array}{ll}
 & \mathtt{succ}\ \ c_2 \\
\to & (\lambda n.\lambda s.\lambda z.\ s\,(n\ s\ z))\ c_2 \\
\to & \lambda s.\lambda z.\ s\,(c_2\ s\ z) \\
\to & \lambda s.\lambda z.\ s\,((\lambda s.\lambda z.\ s\,(s\ z))\ s\ z) \\
\to & \lambda s.\lambda z.\ s\,((\lambda z.\ s\,(s\ z))\ z) \\
\to & \lambda s.\lambda z.\ s\,(s\,(s\ z)) \\
\to & c_3 \\
\not\to &
\end{array}$$

Adding

$$\texttt{plus} = \lambda m.\lambda n.\lambda s.\lambda z.\ m\, s (n\, s\, z) \tag{23}$$

$$\begin{array}{ll}
 & \texttt{plus}\ c_2\ c_2 \\
\to & (\lambda m.\lambda n.\lambda s.\lambda z.\ m\, s\, (n\, s\, z)) c_2 c_2 \\
\to & (\lambda n.\lambda s.\lambda z.\ c_2\ s\, (n\, s\, z)) c_2 \\
\to & \lambda s.\lambda z.\ c_2\ s\, (c_2\ s\, z) \\
\to & \lambda s.\lambda z.\ (\lambda s.\lambda z.\ s\, (s\, z))\ s\, ((\lambda s.\lambda z.\ s\, (s\, z))\ s\, z) \\
\to & \lambda s.\lambda z.\ (\lambda z.\ s\, (s\, z))\ ((\lambda s.\lambda z.\ s\, (s\, z))\ s\, z) \\
\to & \lambda s.\lambda z.\ (s\, (s\, ((\lambda s.\lambda z.\ s\, (s\, z))\ s\, z))) \\
\to & \lambda s.\lambda z.\ (s\, (s\, ((\lambda z.\ s\, (s\, z))\ z))) \\
\to & \lambda s.\lambda z.\ (s\, (s\, (s\, (s\, z)))) \\
\to & c_4 \\
\not\to &
\end{array}$$

Times Have Changed

Finally, let's define a multiplication operator.

$$times = \lambda m.\lambda n.\ m\,(\texttt{plus}\ n)\ c_0 \tag{24}$$

$$\underline{3 \times 2 =?}$$

$$\begin{array}{ll} & \texttt{times}\ c_3\ c_2 \\ \to & (\lambda m.\lambda n.\ m\,(\texttt{plus}\ n)\ c_0)\ c_3\ c_2 \\ \to & (\lambda n.\ c_3\,(\texttt{plus}\ n)\ c_0)\ c_2 \\ \to & (\lambda s.\lambda z.\ s\,(s\,(s\ z)))\,(\texttt{plus}\ c_2)\ c_0 \\ \to & (\texttt{plus}\ c_2)\,((\texttt{plus}\ c_2)\,((\texttt{plus}\ c_2)\ c_0)) \end{array}$$

Sub-Derivation

Technically this is cheating, since we don't have a rule for this type of substitution in the semantic, and it violates our evaluation strategy.

$$
\begin{array}{ll}
 & \texttt{plus}\; c_2 \\
\rightarrow & (\lambda m.\lambda n.\lambda s.\lambda z.\; m\, s\, (n\, s\, z))\, (\lambda s.\lambda z.\; s\, (s\, z)) \\
\rightarrow & (\lambda n.\lambda s.\lambda z.\; (\lambda s.\lambda z.\; s\, (s\, z))\, s\, (n\, s\, z)) \\
\rightarrow & (\lambda n.\lambda s.\lambda z.\; (\lambda z.\; s\, (s\, z))\, (n\, s\, z)) \\
\rightarrow & (\lambda n.\lambda s.\lambda z.\; (s\, (s\, (n\, s\, z))))
\end{array}
$$

(It saves a lot of time though)

$$
\begin{array}{ll}
 & (\mathtt{plus}\ c_2)\ ((\mathtt{plus}\ c_2)\ ((\mathtt{plus}\ c_2)\ c_0)) \\
\rightsquigarrow & (\lambda n. \lambda s. \lambda z.\ (s\ (s\ (n\ s\ z))))\ ((\mathtt{plus}\ c_2)\ ((\mathtt{plus}\ c_2)\ c_0)) \\
\to & \lambda s. \lambda z.\ (s\ (s\ (((\mathtt{plus}\ c_2)\ ((\mathtt{plus}\ c_2)\ c_0))\ s\ z))) \\
\rightsquigarrow & \lambda s. \lambda z.\ (s\ (s\ (((\lambda n. \lambda s. \lambda z.\ (s\ (s\ (n\ s\ z))))\ ((\mathtt{plus}\ c_2)\ c_0))\ s\ z))) \\
\to & \lambda s. \lambda z.\ (s\ (s\ ((\lambda z.\ (s\ (s\ (((\mathtt{plus}\ c_2)\ c_0)\ s\ z))))\ z))) \\
\to & \lambda s. \lambda z.\ (s\ (s\ (s\ (s\ (((\mathtt{plus}\ c_2)\ c_0)\ s\ z))))) \\
\rightsquigarrow & \lambda s. \lambda z.\ (s\ (s\ (s\ (s\ (((\lambda n. \lambda s. \lambda z.\ (s\ (s\ (n\ s\ z))))\ c_0)\ s\ z))))) \\
\to & \lambda s. \lambda z.\ (s\ (s\ (s\ (s\ ((\lambda s. \lambda z.\ (s\ (s\ (c_0\ s\ z))))\ s\ z))))) \\
\to & \lambda s. \lambda z.\ (s\ (s\ (s\ (s\ ((\lambda z.\ (s\ (s\ (c_0\ s\ z))))\ z))))) \\
\to & \lambda s. \lambda z.\ (s\ (s\ (s\ (s\ (s\ (s\ (c_0\ s\ z))))))) \\
\to & \lambda s. \lambda z.\ (s\ (s\ (s\ (s\ (s\ (s\ ((\lambda s. \lambda z.\ z)\ s\ z))))))) \\
\to & \lambda s. \lambda z.\ (s\ (s\ (s\ (s\ (s\ (s\ ((\lambda z.\ z)\ z))))))) \\
\to & \lambda s. \lambda z.\ (s\ (s\ (s\ (s\ (s\ (s\ z)))))) \\
\not\to &
\end{array}
$$