Announcements

- Prelim #2 issues:
  - Problem 5 grading guide out shortly
  - Problem 3 (hashing) issues
    - Will be on final!
- Friday RDZ office hours are 11-12 not 1:30-2:30
Introduction to the $\lambda$-Calculus

- Today we will take a look at the theoretical roots of functional programming in the form of a mathematical precursor to OCaml, the $\lambda$-(lambda-) calculus.

- The $\lambda$-calculus was invented by Alonzo Church in the 1930s to study the interaction of functional abstraction and function application from an abstract, purely mathematical point of view.

- It was one of many related systems that were proposed in the late 1920s and 1930s.
  - This was an exciting time for theoretical computer science, even though computers had not yet been invented.
  - There were many great mathematicians such as Alonzo Church, Kurt Gödel, Alan Turing, and Emil Post who were trying to understand the notion of computation. They certainly understood intuitively what an algorithm was, but they were lacking a formal mathematical definition.

- They came up with a number of different systems for capturing the general idea of computation:
  - Turing machines—Alan Turing
  - $\mu$-recursive functions—Kurt Gödel
  - rewrite systems—Emil Post
  - the $\lambda$-calculus—Alonzo Church
  - combinatory logic—Haskell Curry

- These systems all looked very different, but what was amazing was that they all turned out to be computationally equivalent in the sense that each could encode and simulate the others.
• Church thought that this was too striking to be mere coincidence, and declared that they had found the correct definition of the notion of *computability*: it was the common spirit embodied in a different way by each one of these formalisms.
  o This declaration became known as Church's thesis.
• Nowadays we have a better understanding of computability, and one could add any modern general-purpose programming language such as Java, C, Scheme, or OCaml (suitably abstracted) to this list, but Church and his colleagues did not have that luxury.

---

**The λ-Calculus**

• The λ-calculus is like OCaml, except much simpler:
  o *Everything* is a function.
    ▪ There are no other types—no integers, strings, lists, booleans, etc.
    ▪ There is no unit (\(\)).
    ▪ If you want these things, you must encode them using functions.
    ▪ Luckily, functions are powerful enough to do this.
  o There are no notions of state or side effects. It is purely functional.
    ▪ Thus we can think exclusively in terms of the substitution model.
  o The order of evaluation is irrelevant.
    ▪ OCaml imposes a certain evaluation order, namely *eager evaluation*—evaluate the arguments before applying the function.
    ▪ The λ-calculus does not specify an evaluation order.
  o All functions are *unary* (take one argument). Functions of multiple arguments are encoded by currying.
Syntax

\( \lambda \)-terms \( E, F, G, \ldots \) are defined inductively:

1. **variables**: any variable \( x, y, z, \ldots \) is a \( \lambda \)-term.
2. **application**: if \( E \) and \( F \) are \( \lambda \)-terms, then \( EF \) is a \( \lambda \)-term.
3. **abstraction**: if \( E \) is a \( \lambda \)-term and \( x \) is a variable, then \( \lambda x. E \) is a \( \lambda \)-term.

Notes:

1. This is the basis of the inductive definition of \( \lambda \)-terms.

2. The expression \( EF \) denotes application of \( E \) (considered as a function) to \( F \) (considered as an input). Think of this as an expression with an invisible infix operator *apply* between the \( E \) and \( F \), like + or *. This might also be written \( (EF) \) or \( E(F) \) with parentheses as necessary to avoid ambiguity. The apply operator is not associative, so \( (EF)G \) is not the same as \( E(FG) \) in general. The expression \( EFG \) written without parentheses should be read as \( (EF)G \). This is the same convention as in OCaml.

3. The expression \( \lambda x. E \) would be written \( \text{fun} \ x \rightarrow E \) in OCaml. The application operator binds tighter than abstraction operator, so \( \lambda x. \lambda x. xy \) should be read as \( \lambda x. (\lambda x. (xy)) \).
Substitution

Computation in the \( \lambda \)-calculus is effected by substitution. There are two rules:

1. **\( \alpha \)-conversion** (renaming bound variables): \( \lambda x.E \Rightarrow \lambda y.(E[y/x]) \), where \( E[y/x] \) denotes the result of replacing all free occurrences of \( x \) in \( E \) by \( y \), provided \( y \) would not be captured by a \( \lambda y \) already in \( E \).
   - **Examples:**
     - OK: \( \lambda x.\lambda y.xy \Rightarrow \lambda z.\lambda y.zy \) (rename \( x \) to \( z \), nothing bad happened)
     - not OK: \( \lambda x.\lambda y.xy \Rightarrow \lambda y.\lambda y.yy \) (rename \( x \) to \( y \), \( y \) was captured)

2. **\( \beta \)-reduction** (substitution rule): \( (\lambda x.E)F \Rightarrow E[F/x] \), where \( E[F/x] \) denotes the result of replacing all free occurrences of \( x \) in \( E \) by \( F \). Before doing this, bound variables in \( E \) are renamed by \( \alpha \)-conversion if necessary to avoid capturing free variables in \( F \).
   - **Examples:**
     - OK: \( (\lambda x.\lambda y.xy)(\lambda z.z) \Rightarrow \lambda y.(\lambda z.z)y \)
     - not OK: \( (\lambda x.\lambda y.xy)(\lambda z.yz) \Rightarrow \lambda y.(\lambda z.yz)y \) (\( y \) was captured)
     - but if we rename the bound \( y \) first, it's OK:
       \( (\lambda x.\lambda y.xy)(\lambda z.yz) \Rightarrow (\lambda x.\lambda a.xa)(\lambda z.yz) \Rightarrow \lambda a.(\lambda z.yz)a \)

Examples in OCaml:

1. \( \text{fun } x \rightarrow x + 3 \) is equivalent to \( \text{fun } y \rightarrow y + 3 \).
2. This is just the substitution rule. \( (\text{fun } x \rightarrow x + 3) \ 7 \Rightarrow 7 + 3 \).

An \( \alpha \)- or \( \beta \)-reduction step can be performed at any time to any subterm of a \( \lambda \)-term. Write \( E \Rightarrow F \) if \( E \) goes to \( F \) in some finite number of \( \alpha \)- or \( \beta \)-reduction steps.
Order of Evaluation

- In the λ-calculus, a reduction rule can be performed at any time to any subterm of a λ-term at which the rule is applicable. OCaml specifies an order of evaluation, but the λ-calculus does not.
  - It turns out that it does not matter because of the Church-Rosser theorem: If \( E \Rightarrow F_1 \) and \( E \Rightarrow F_2 \), then there always exists a \( G \) such that \( F_1 \Rightarrow G \) and \( F_2 \Rightarrow G \).

- So if you diverge by applying rules to different parts of the term, you can always get back together again by applying more rules. Example:

```
(fun x -> x + x) (2*3)
\ / \ eager
 lazy/ \ / \ / \ / \ / \ / \\
(2*3) + (2*3) (fun x -> x + x) 6
\ / \ / \ / \\
6 + 6
```

- Terms are equivalent if they have a common reduct; that is, \( E_1 = E_2 \) if there exists an \( F \) such that \( E_1 \Rightarrow F \) and \( E_2 \Rightarrow F \).
The relation \( \equiv \) is an equivalence relation on terms (reflexive, symmetric, transitive). To show transitivity, we use the Church-Rosser property: if \( E_1 \equiv E_2 \) and \( E_2 \equiv E_3 \), then there exist \( F_1 \) and \( F_2 \) such that:

\[
\begin{array}{c}
E_1 \\
\downarrow \\
F_1

de \equiv \\
\uparrow \\
E_2 \\
\downarrow \\
F_2
\end{array}
\]

\[
\begin{array}{c}
E_3 \\
\downarrow \\
F_2

de \equiv \\
\uparrow \\
\begin{array}{c}
E_2 \\
\downarrow \\
\begin{array}{c}
E_1 \\
\downarrow \\
\begin{array}{c}
E_3 \\
\downarrow \\
\begin{array}{c}
F_1 \\
\downarrow \\
\begin{array}{c}
F_1 \\
\downarrow \\
\begin{array}{c}
F_1

de \equiv \\
\uparrow \\
G
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\end{array}
\]

By the Church-Rosser property, there exists a \( G \) such that \( F_1 \Rightarrow G \) and \( F_2 \Rightarrow G \):

\[
\begin{array}{c}
E_1 \\
\downarrow \\
F_1

de \equiv \\
\uparrow \\
E_2 \\
\downarrow \\
F_2
\end{array}
\]

\[
\begin{array}{c}
E_3 \\
\downarrow \\
F_2
\end{array}
\]

Thus \( E_1 \Rightarrow G \) and \( E_3 \Rightarrow G \), which proves that \( E_1 \equiv E_3 \).
Values

- A λ-term is said to be in normal form if there are no β-reductions that apply; that is, if there are no subterms of the form \((\lambda x. M) N\). For example, the terms \(x\) and \(\lambda x. y\) are in normal form. Terms in normal form correspond to values in OCaml. They are objects that cannot be evaluated further.

- A term is said to be normalizable if there is a sequence of α- and β-reductions that lead to a normal form. Because of the Church-Rosser property, if a term is normalizable, then its normal form is unique (up to α-conversion). A sequence of reductions leading to a normal form corresponds roughly to a halting computation in OCaml.

- Some terms are not normalizable. For example, \((\lambda x. xx) (\lambda x. xx)\) is not. Even if a term is normalizable, there may be infinite reduction sequences that never lead to a normal form. For example, \((\lambda y. z) ((\lambda x. xx) (\lambda x. xx))\) reduces to the normal form \(z\) if it is evaluated lazily, but does not terminate if evaluated eagerly.
Encoding Other Data Types

The λ-calculus is powerful enough to encode other data types and operations.

Booleans

Define

\[ \text{true} = \lambda x.\lambda y. x \]
\[ \text{false} = \lambda x.\lambda y. y \]

- In other words, \text{true} and \text{false} are curried two-argument functions that just return the first and second argument, respectively. We can then encode the conditional \text{if-then-else} as the identity function \( \lambda x.x \):

\[
\text{if-then-else true B C} \\
=> (\lambda x.x) \text{true} B C \\
=> \text{true} B C \\
=> (\lambda y.B) C \\
=> B
\]

\[
\text{if-then-else false B C} \\
=> (\lambda x.x) \text{false} B C \\
=> \text{false} B C \\
=> (\lambda y.y) C \\
=> C
\]

We can define other Boolean operations:

\[ \text{and} = \lambda x.\lambda y.\text{xyx} \]
\[ \text{or} = \lambda x.\lambda y.\text{xyx} \]
\[ \text{not} = \lambda x.(x \text{false true}) \]

For example,

\[
\text{and true false} \\
=> (\lambda x.\lambda y.\text{xyx}) \text{true} \text{false} \\
=> \text{true} \text{false} \text{true} \\
=> (\lambda x.\lambda y.x) \text{false} \text{true} \\
=> \text{false}
\]
Pairs and Tuples

- In OCaml, all the pairing operator `( , )` does is package up two things in such a way that they can be extracted again by `fst` and `snd`. To code this in the λ-calculus, we can take two terms `A` and `B` and produce a function that will apply another function to them. Define

\[ \text{pair} = \lambda x.\lambda y.\lambda c.cxy \]

If we apply this to `A` and `B` we get

\[ \text{pair } A \hspace{0.2cm} B = \lambda c.(c \hspace{0.2cm} A \hspace{0.2cm} B) \]

- and this is how we encode the pair `(A, B)`. Now we can extract `A` and `B` again by applying this function to `true` and `false`, respectively:

\[ (\lambda c.(c \hspace{0.2cm} A \hspace{0.2cm} B)) \hspace{0.2cm} \text{true} \]
\[ => \hspace{0.2cm} \text{true } A \hspace{0.2cm} B \]
\[ => \hspace{0.2cm} (\lambda x.\lambda y.x) \hspace{0.2cm} A \hspace{0.2cm} B \]
\[ => \hspace{0.2cm} A \]

\[ (\lambda c.(c \hspace{0.2cm} A \hspace{0.2cm} B)) \hspace{0.2cm} \text{false} \]
\[ => \hspace{0.2cm} \text{false } A \hspace{0.2cm} B \]
\[ => \hspace{0.2cm} (\lambda x.\lambda y.y) \hspace{0.2cm} A \hspace{0.2cm} B \]
\[ => \hspace{0.2cm} B \]

- so we should define

\[ \text{fst} = \lambda d.(d \hspace{0.2cm} \text{true}) \]
\[ \text{snd} = \lambda d.(d \hspace{0.2cm} \text{false}) \]
Lists

- We can encode lists as iterated tuples. Thus `::` is the same as `pair`, `hd` is `fst`, and `tl` is `snd`. If we define the empty list `[]` appropriately, we can get a null test. Define

\[
[] = \lambda x.\text{true} \\
\text{null} = \lambda d.(d \lambda x.\lambda y.\text{false})
\]

Then

\[
\text{null} [] \\
\quad \Rightarrow \lambda d.(d \lambda x.\lambda y.\text{false}) (\lambda x.\text{true}) \\
\quad \Rightarrow (\lambda x.\text{true}) (\lambda x.\lambda y.\text{false}) \\
\quad \Rightarrow \text{true}
\]

\[
\text{null} (\text{pair } A B) \\
\quad \Rightarrow \lambda d.(d \lambda x.\lambda y.\text{false}) (\lambda c.(c A B)) \\
\quad \Rightarrow \lambda c.(c A B) (\lambda x.\lambda y.\text{false}) \\
\quad \Rightarrow (\lambda x.\lambda y.\text{false}) A B \\
\quad \Rightarrow \text{false}
\]
Numbers and Arithmetic

- In OCaml, numbers are a built-in primitive type. In the \(\lambda\)-calculus, we have to encode them as \(\lambda\)-terms. Church came up with a neat way to do this. His encoding is called the Church numerals.
- The idea is to encode the number \(n\) as a \(\lambda\)-term \(n\) representing a function that takes another function \(f\) as input and produces the \(n\)-fold composition of \(f\) with itself.

\[
\begin{align*}
0 &= \lambda f. \lambda x. x \\
1 &= \lambda f. \lambda x. fx \\
2 &= \lambda f. \lambda x. f(fx) \\
3 &= \lambda f. \lambda x. f(f(fx)) \\
\vdots \\
n &= \lambda f. \lambda x. f(...(f(f(fx)))...) \\
\end{align*}
\]

- Note that \(0\) is just \text{false} after \(\alpha\)-conversion. Let's write \(f^n x\) as an abbreviation for the term \(f(...(f(f(fx)))...)\) with \(n\) applications of \(f\). Then

\[
n = \lambda f. \lambda x. f^n x
\]

- This is nice, but in order to do anything with it, we have to encode arithmetic. Let's start with the simplest operation, namely successor. We would like to define a \(\lambda\)-term \(\text{add1}\) that when applied to any Church numeral \(n\) produces \(n+1\). Note that

\[
nfx \Rightarrow f^0 x
\]

so

\[
f(nfx) \Rightarrow f(f^0 x) = f^{n+1} x
\]

- \(\lambda\)-abstracting, we get

\[
\begin{align*}
\lambda f. \lambda x. f(nfx) \\
&\Rightarrow \lambda f. \lambda x. f^{n+1} x \\
&= n+1
\end{align*}
\]

Therefore we want our successor function to produce \(\lambda f. \lambda x. f(nfx)\) from \(n\). Thus we just define

\[
\text{add1} = \lambda n. \lambda f. \lambda x. f(nfx)
\]
How do we test for 0?

\[ \text{isZero} = \lambda n. (n \ \lambda z. \text{false} \ \text{true}) \]

Then

\[
\begin{align*}
\text{isZero } 0 & \Rightarrow \lambda n. (n \ \lambda z. \text{false} \ \text{true}) \ \lambda f. \lambda x. x \\
& \Rightarrow \lambda f. \lambda x. x \\
& \Rightarrow \text{true} \\
\text{isZero } n+1 & \Rightarrow \lambda n. (n \ \lambda z. \text{false} \ \text{true}) \ \lambda f. \lambda x. f^{n+1} x \\
& \Rightarrow \lambda f. \lambda x. f^{n+1} x \ \lambda z. \text{false} \ \text{true} \\
& \Rightarrow \lambda x. (\lambda z. \text{false})^{n+1} x \ \text{true} \\
& \Rightarrow (\lambda z. \text{false})^{n+1} \text{true} \\
& \Rightarrow \text{false}
\end{align*}
\]

Addition is just iterated successor. Note that

\[ m \ \text{add1 } \Rightarrow \lambda f. \lambda x. f^{m} x \ \text{add1 } \Rightarrow \lambda x. (\text{add1}^{m} x) \]

thus

\[ m \ \text{add1 } n \Rightarrow \lambda x. (\text{add1}^{m} x) \ n \Rightarrow \text{add1}^{m} \ n \Rightarrow m+n \]

This suggests that we should define

\[ \text{add} = \lambda n. \lambda m. (m \ \text{add1} n) \]

Multiplication can be defined by iterated addition and exponentiation by iterated multiplication in a similar fashion.

\[ \text{mul} = \lambda n. \lambda m. (m \ \text{add} n) \ 0 \]
\[ \text{exp} = \lambda n. \lambda m. (m \ \text{mul} n) \ 1 \]

Subtraction is more difficult, but it can be done. As above, we start by defining the predecessor function sub1, where

\[ \text{sub1 } n+1 = n \]
\[ \text{sub1 } 0 = 0 \]

Recall from our encoding of lists above that

\[
\begin{align*}
[&] = \lambda x. \text{true} \\
[n; \ n-1; \ n-2; \ \ldots; \ 2; \ 1; \ 0] = \text{pair } n \ [n-1; \ n-2; \ \ldots; \ 2; \ 1; \ 0]
\end{align*}
\]

so \[ \ldots \] represents a list in the usual OCaml sense under our encoding. Our approach is to come up with a term \( F \) such that
\[ n \ F \ [0] \Rightarrow [n; n-1; n-2; \ldots; 2; 1; 0] \]

If we can concoct such a term \( F \), then we can take

\[ \text{sub1} = \lambda n. (\text{if-then-else} \ (\text{isZero} \ n) \ 0 \ \text{(fst \ (snd \ (n \ F \ [0])))}) \]

Note that

\[ n \ F \ [0] \Rightarrow F^n \ [0] \]

so this will work provided for all \( n \) we have

\[ F \ [n; n-1; n-2; \ldots; 2; 1; 0] \Rightarrow [n+1; n; n-1; n-2; \ldots; 2; 1; 0] \]

This can be achieved by defining

\[ F = \lambda x. (\text{pair} \ (\text{add1} \ (\text{fst} \ x)) \ x) \]

Proper subtraction \((\text{sub} \ n \ m = 0 \ \text{if} \ n < m, \ n - m \ \text{if} \ n \geq m)\) can be effected by iterating \text{sub1} exactly as \text{add} was defined by iterating \text{add1}.