Lecture Outline

- Why study λ-calculus?
- Lambda calculus
  - Syntax
  - Evaluation
  - Relationship to programming languages
- Church-Rosser theorem
- Completeness of Lambda Calculus: Turing Complete

λ-calculus

- A framework developed in 1930s by Alonzo Church to study computations with functions
- Church wanted a minimal notation
  - to expose only what is essential
  - Similar to BNF, Normal form in Rel Database
- Two operations with functions are essential:
  - function creation
  - function application

What Is Calculus?

- In School:
  \[ \frac{d}{dx} x^n = nx^{n-1} \] [Power Rule]
  \[ \frac{d}{dx} (f + g) = \frac{d}{dx} f + \frac{d}{dx} g \] [Sum Rule]

Calculus is a branch of mathematics that deals with limits and the differentiation and integration of functions of one or more variables...

Why Study λ-calculus?

- Basic syntactic notions
  - Free and bound variables
  - Functions, and Declarations
- λ-Calculation rule
  - Symbolic evaluation useful for discussing programs
  - Used in optimization (inlining), macro expansion
    - Correct macro processing requires variable renaming
    - Illustrates some ideas about scope of binding

Why Study λ-calculus?

- Tremendous influence on design and analysis of programming languages
- Realistic languages are too large and complex to study from scratch as a whole
  - Typical approach is to modularize the study into one feature at a time
    - E.g., recursion, looping, exceptions, objects, etc.
  - Then we assemble the features together
Why Study λ-calculus?
- λ-calculus is the standard testbed for studying programming language features
  - Because of its minimality
  - Despite its syntactic simplicity the λ-calculus can easily encode:
    - numbers, recursive data types, modules, imperative features, exceptions, etc.
- Certain language features necessitate more substantial extensions to λ-calculus:
  - Distributed & parallel languages: π-calculus
  - Object oriented languages: σ-calculus

“Whatever the next 700 languages turn out to be, they will surely be variants of λ-calculus.”
(Landin 1966)

Backus’ Turing Award
- John Backus: 1977 Turing Award
  - Designer of Fortran, BNF, etc.
- Turing Award lecture
  - Functional programming better than imperative programming
  - Easier to reason about functional programs
  - More efficient due to parallelism
  - Algebraic laws
    - Reason about programs
    - Optimizing compilers

Reasoning About Programs
- To prove a program correct, must consider everything a program depends on
- In functional programs, dependence on any data structure is explicit
  - fully and clearly expressed
- Therefore, it’s easier to reason about functional programs

Function Application
- The only thing that we can do with a function is to apply it to an argument
- Church used the notation \( \lambda x. E \)
  to denote a function with formal argument \( x \) and with body \( E \)
- Functions do not have names
  - names are not essential for the computation
- Functions have a single argument
  - Once we understand how functions with one argument work we can generalize to multiple args.

Function Creation
- Church introduced the notation \( \lambda x. E \)
  to denote a function with formal argument \( x \) and with body \( E \)
**Syntax of Lambda Calculus**

- Only three kinds of expressions
  
  \[ E ::= x \quad \text{variables} \]
  
  \[ | E_1, E_2 \quad \text{function application} \]
  
  \[ | \lambda x. E \quad \text{function creation} \]

- The form \( \lambda x. E \) is also called lambda abstraction, or simply abstraction

- E are called \( \lambda \)-terms or \( \lambda \)-expressions

**Scope of Variables**

- As in all languages with variables, it is important to discuss the notion of scope
  
  - Recall: the scope of an identifier is the portion of a program where the identifier is accessible

- An abstraction \( \lambda x. E \) binds variable \( x \) in \( E \)
  
  - \( x \) is the newly introduced variable
  
  - \( E \) is the scope of \( x \)
  
  - We say \( x \) is bound in \( \lambda x. E \)

- Just like formal function arguments are bound in the function body

**Free and Bound Variables**

- Bound variable is a “placeholder”
  
  - Variable \( x \) is bound in \( \lambda x. (x+y) \)
  
  - Function \( \lambda x. (x+y) \) is same function as \( \lambda z. (z+y) \)

- Compare to Calculus

  \[ \int x+y \, dx = \int z+y \, dz \quad \forall x \quad P(x) = \forall z \quad P(z) \]

**Expressions and Functions**

- Expressions

  \[ x + y \quad x + 2*y + z \]

- Functions

  \[ \lambda x. (x+y) \quad \lambda z. (x + 2*y + z) \]

- Application

  \[ (\lambda x. (x+y)) \, 3 \quad = \quad 3 + y \]
  
  \[ (\lambda z. (x + 2*y + z)) \, 5 \quad = \quad x + 2*y + 5 \]

- Parsing: \( \lambda x. f (x) = \lambda x. ( f (f (x))) \)

**Free and Bound Variables (Cont.)**

- Just like in any language with static nested scoping, we have to worry about variable shadowing
  
  - An occurrence of a variable might refer to different things in different context

- E.g., in CoolFun: let \( x \leftarrow E \) in \( x + (\text{let } x \leftarrow E' \text{ in } x) + x \)

- In \( \lambda \)-calculus: \( \lambda x. (\lambda x. x) \times \)

- Renaming \( \lambda x. x \times (\lambda z. z) \times \)

- Name of free (i.e., unbound) variable matters!
  
  - Variable \( y \) is free in \( \lambda x. (x+y) \)
  
  - Function \( \lambda x. (x+y) \) is not same as \( \lambda x. (x+z) \)

- Occurrences

  - \( y \) is free and bound in \( \lambda x. ((\lambda x. y+z) \times) + y \)
### Renaming Bound Variables

- Two \( \lambda \)-terms that can be obtained from each other by a renaming of the bound variables are considered identical
  \[ \lambda x. x \text{ is identical to } \lambda y. y \text{ and to } \lambda z. z \]
- Intuition:
  - by changing the name of a formal argument and of all its occurrences in the function body, the behavior of the function does not change
- \[ \text{fun } (x)=x^*x = \text{fun } (y)=y^*y = \text{fun } (z)=z^*z \]
- in \( \lambda \)-calculus such functions are considered identical

### Examples of Lambda Expressions

- The identity function:
  \[ I \text{ def } \lambda x. x \]
- A function that given an argument \( y \) discards it and computes the identity function:
  \[ \lambda y. (\lambda x. x) \]
- A function that given a function \( f \) invokes it on the identity function:
  \[ \lambda f. f (\lambda x. x) \]

### Syntactic Sugar

- Syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express.
- It makes the language "sweeter" for human use:
- things can be expressed more clearly, more concisely, or in an alternative style that some may prefer.

```java
context.checking(
    new Expectations(){
      //Better one
      oneOf(alarm).getAttackAlarm(null);
    });

Expectations exp = new Expectations();
exp.oneOf(alarm).getAttackAlarm(null);
context.checking(exp);
```

### Notational Conventions

- Application associates to the left
  \[ x \ y \ z \text{ parses as } (x \ y) \ z \]
- Abstraction extends to the right as far as possible
  \[ \lambda x. \lambda y. x \ y \ z \text{ parses as } \lambda x. (\lambda y. ((x \ y) \ z)) \]
- And yields the the parse tree:

### Declarations as "Syntactic Sugar"

```java
function f(x) {
  return x+2;
}
f(5);
(l.f f(5)) (\lambda x. x+2)
```

**Same as** \( \lambda x. x+2 \)

---

**Note:**

"Syntactic sugar is syntax within a programming language that is designed to make things easier to read or to express. It makes the language "sweeter" for human use: things can be expressed more clearly, more concisely, or in an alternative style that some may prefer."
Substitution

- The substitution of E’ for x in E (written \([E'/x]E\))
  - Step 1. Rename bound variables in E and E’ so they are unique
  - Step 2. Perform the textual substitution of E’ for x in E
- Example: \([y (\lambda x. x) / x] \lambda y. (\lambda x. x) y x\)
  - After renaming: \([y (\lambda v. v)/x] \lambda z. (\lambda u. u) z x\)
  - After substitution: \(\lambda z. (\lambda u. u) z (y (\lambda v. v))\)

\[\text{Example: } [y (\lambda x. x) / x] \lambda y. (\lambda x. x) y x\]
\[\text{Example: } \lambda x. M N \rightarrow M [x \mapsto N]\]

\(\beta\)-Reduction

(the source of all computation)

\[\lambda x. M N \rightarrow M [x \mapsto N]\]

Replace all \(x\)’s in \(M\) with \(N\)

Evaluation and the Static Scope

- The definition of substitution guarantees that evaluation respects static scoping:
  \((\lambda x. (\lambda y. y) x) \rightarrow \lambda z. z (y (\lambda u. v))\)
  \((y\text{ remains free, i.e., defined externally})\)

Evaluating Lambda Expressions

- \textit{redex}: Term of the form \((\lambda x. M)N\)
  - Something that can be \(\beta\)-reduced
  - An expression is in \textit{normal form} if it contains no redexes (redices).
  - To evaluate a lambda expression, keep doing reductions until you get to \textit{normal form}.

\(\alpha\)-reduction (or renaming)

\[\lambda y. M \Rightarrow_{\alpha} \lambda y. (M[y \mapsto v])\]

where \(v\) does not occur in \(M\).

Example: \([y (\lambda x. x) / x] \lambda y. (\lambda x. x) y x\)

\[= [y (\lambda v. v)/x] \lambda z. (\lambda u. u) z x\]

\(\beta\)-reduction (or substitution)

\[\lambda x. M N \Rightarrow_{\beta} M [x \mapsto N]\]

Note the syntax is different from Scheme:

\[\lambda x. M N \rightarrow ((\lambda x. M) N)\]

Examples of Evaluation

- The identity function:
  \((\lambda x. x) E \rightarrow [E / x] x = E\)
  - Another example with the identity:
    \((\lambda f. f (\lambda x. x)) (\lambda x. x) \rightarrow [\lambda x. x f f (\lambda x. x)] = (\lambda x. x) (\lambda y. y) \rightarrow \lambda y. y\)

- A non-terminating evaluation:
  \((\lambda x. xx)(\lambda x. xx) \rightarrow [\lambda x. xx / x] xx = [\lambda y. yy / x] xx = (\lambda y. yy)(\lambda y. yy) \rightarrow \ldots\)
The Order of Evaluation

- In a λ-term, there could be more than one instance of \( (\lambda x. E)^f \)
  \[ (\lambda y. (\lambda x. y) y) E \]
  - could reduce the inner or the outer lambda
  - which one should we pick?
  \[ (\lambda y. (\lambda x. x) y) E \]

Church-Rosser theorem: Order of Evaluation

- The Church-Rosser theorem says that any order will compute the same result
  - A result is a λ-term that cannot be reduced further
- But we might want to fix the order of evaluation when we model a certain language

Higher-Order Functions

- Given function f, return function \( f \circ f \)
  \[ \lambda f. \lambda x. f (f x) \]
- How does this work?
  \[ (\lambda f. \lambda x. f (f x)) \ (\lambda y. y + 1) \]
  \[ = \lambda x. (\lambda y. y + 1) (\lambda y. y + 1) \ x ]
  \[ = \lambda x. (\lambda y. y + 1) \ x + 1 ]
  \[ = \lambda x. \ (x + 1) + 1 ]

Same Procedure (ML)

- Given function f, return function \( f \circ f \)
  \[ fn f \Rightarrow fn x \Rightarrow f(f(x)) \]
- How does this work?
  \[ fn f \Rightarrow fn x \Rightarrow f(f(x)) \]
  \[ = fn x \Rightarrow ((fn y \Rightarrow y + 1) ((fn y \Rightarrow y + 1) x)) \]
  \[ = fn x \Rightarrow ((fn y \Rightarrow y + 1) \ x + 1) \]
  \[ = fn x \Rightarrow ((x + 1) + 1) \]

Lambda Calculus and Programming Languages

- Pure lambda calculus has only functions
- What if we want to compute with Booleans, numbers, lists, etc.?
  - All these can be encoded in pure λ-calculus
  - The trick: do not encode what a value is but what we can do with it!
- For each data type, we have to describe how it can be used, as a function
  - then we write that function in λ-calculus

Expressiveness of Lambda Calculus

- The λ-calculus can express
  - data types (integers, Booleans, lists, trees, etc.)
  - branching (using Booleans)
  - recursion
- This is enough to encode Turing machines
- Encodings are fun
- But programming in pure λ-calculus is painful
  - we will add constants (0, 1, 2, ..., true, false, if-then-else, etc.)
  - and we will add types
Encoding Booleans in Lambda Calculus

- What can we do with a Boolean?
  - we can make a binary choice
- A Boolean is a function that given two choices selects one of them
  - true = \( \lambda x. \lambda y. x \)
  - false = \( \lambda x. \lambda y. y \)
  - if \( E_1 \) then \( E_2 \) else \( E_3 \) = \( \text{def} E_1 E_2 E_3 \)
- Example: if true then \( u \) else \( v \) is
  \[ (\lambda x. \lambda y. x) u v \rightarrow_{\beta} (\lambda y. u) v \rightarrow_{\beta} u \]

Encoding Pairs in Lambda Calculus

- What can we do with a pair?
  - we can select one of its elements
- A pair is a function that given a boolean returns the left or the right element
  \[ \text{mkpair} \ x \ y = \text{def} \ \lambda b. \ x \ y \]
  \[ \text{fst} \ p = \text{def} \ p \ \text{true} \]
  \[ \text{snd} \ p = \text{def} \ p \ \text{false} \]
- Example:
  \[ \text{fst} \ (\text{mkpair} \ x \ y) \rightarrow (\text{mkpair} \ x) \ y \rightarrow \text{true} \rightarrow x \ y \rightarrow x \]

Encoding Natural Numbers in Lambda Calculus

- What can we do with a natural number?
  - we can iterate a number of times
- A natural number is a function that given an operation \( f \) and a starting value \( s \), applies \( f \) a number of times to \( s \):
  \[ 0 = \text{def} \ \lambda f. \ \lambda s. \ s \]
  \[ 1 = \text{def} \ \lambda f. \ \lambda s. \ f \ s \]
  \[ 2 = \text{def} \ \lambda f. \ \lambda s. \ f \ (f \ s) \]
  and so on

Computing with Natural Numbers

- The successor function
  \[ \text{succ} \ n = \text{def} \ \lambda f. \ \lambda s. \ f \ (n \ f \ s) \]
- Addition
  \[ \text{add} \ n_1 \ n_2 = \text{def} \ n_1 \ \text{succ} \ n_2 \]
- Multiplication
  \[ \text{mult} \ n_1 \ n_2 = \text{def} \ n_1 \ (\text{add} \ n_2) \ 0 \]
- Testing equality with 0
  \[ \text{iszero} \ n = \text{def} \ n \ (\lambda b. \ \text{false}) \ \text{true} \]

Computing with Natural Numbers. Example

\[
\begin{align*}
\text{mult} \ 2 \ 2 & \rightarrow \\
2 \ (\text{add} \ 2) \ 0 & \rightarrow \\
(\text{add} \ 2) \ ((\text{add} \ 2) \ 0) & \rightarrow \\
2 \ \text{succ} \ (\text{add} \ 2) \ 0 & \rightarrow \\
2 \ \text{succ} \ (2 \ \text{succ} \ 0) & \rightarrow \\
\text{succ} \ (\text{succ} \ (\text{succ} \ (\text{succ} \ 0))) & \rightarrow \\
\text{succ} \ (\text{succ} \ (\text{succ} \ ((\lambda f. \ \lambda s. \ f \ (0 \ f \ s)))) & \rightarrow \\
\text{succ} \ (\text{succ} \ ((\lambda f. \ \lambda s. \ f \ s))) & \rightarrow \\
\text{succ} \ ((\lambda g. \ \lambda \ y. \ g \ ((\lambda f. \ \lambda s. \ f \ s) \ g \ y))) & \rightarrow \\
\text{succ} \ ((\lambda g. \ \lambda \ y. \ g \ (g \ y))) & \rightarrow 4
\end{align*}
\]