Principles of Programming Languages

Module M02: λ-Calculus: Syntax

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

January 05, 10 & 12, 2022

Table of Contents

Relations

Relations

- r is a **relation** between two sets A and B:

$$r \subseteq A \times B \text{ or } r = \{(u, v) : u \in A, v \in B\}$$

- **Set of relations** between A and B is $2^{A \times B}$, where 2^X is the **power set** of a set X
- If $A = B$, r is said to be a **relation over** A
- r is
  - *Reflexive*: $\forall t \in A \Rightarrow (t, t) \in r$
  - *Symmetric*: $\forall u, v \in A : (u, v) \in r \Rightarrow (v, u) \in r$
  - *Transitive*: $\forall u, v, w \in A : (u, v),\ (v, w) \in r \Rightarrow (u, w) \in r$
  - *Antisymmetric*: $\forall u, v \in A : (u, v), (v, u) \in r \Rightarrow u = v$
  - *Equivalence relation*: *Reflexive*, *Symmetric*, and *Transitive*
- A relation r may be n-ary over sets $A_1, A_2, \cdots, A_n$

$$r \subseteq A_1 \times A_2 \times \cdots \times A_n$$

- An n-ary relation may be decomposed into a number of binary relations

Functions

Functions

- $f : A \rightarrow B$ is a **function** from A to B if
  - f is a relation between A and B (that is, $f \in A \times B$), and
  - $\forall (s_1, s_2), (t_1, t_2) \in f, s_1 = t_1 \Rightarrow s_2 = t_2$
- f is **total** if $\forall u \in A, \exists (u, v) \in f$
- f is **partial**, otherwise
- **Set of functions** from A to B is $B^A \subset 2^{A \times B}$
- A is the **domain**, B is the **codomain** or **range**
- **Image** $f(A)$ of f is $\{v : \forall u \in A, f(u) = v\}$
- A total function f is
  - *Injective (one-to-one)*: $\forall u, v \in A, f(u) = f(v) \Rightarrow u = v$
  - *Surjective (onto)*: $f(A) = B$
  - *Bijective (one-to-one and onto)*: *Injective* and *Surjective*
- $f^{-1} = \{(v, u) : (u, v) \in f\}$ is the **inverse** of f.
- f^{-1} is a function iff f is a bijection; relation otherwise

Function Compositions

- Given the mathematical functions:

$$f(x) = x^2, \; g(x) = x + 1$$

$f \circ g$ is the composition of f and g:

$$(f \circ g)(x) = f(g(x))$$

$$(f \circ g)(x) = f(g(x)) = f(x+1) = (x+1)^2 = x^2 + 2x + 1$$
$$(g \circ f)(x) = g(f(x)) = g(x^2) = x^2 + 1$$

- Function composition, therefore, is not commutative
- Function composition can be regarded as a (higher-order) function with the following type:

$$\circ : (Z \rightarrow Z) \times (Z \rightarrow Z) \rightarrow (Z \rightarrow Z)$$

Curried Functions

- Using **currying**[1], one-variable functions can represent multiple-variable functions
- Consider:

$$h(x, y) = x + y \text{ of type } h : Z \times Z \rightarrow Z$$

- Represent h as h^c of type[2]

$$h^c : Z \rightarrow Z \rightarrow Z \text{ or } h^c : Z \rightarrow (Z \rightarrow Z) \text{ or } h^c : Z \rightarrow Z^Z$$

  such that

$$h(x, y) = h^c(x)(y) = x + y$$

  - For example, $h^c(2) = g$, where $g(y) = 2 + y$
- h^c is the **curried version of** h.

[1] Haskell Curry used this mechanism in the study of functions. Incidentally, Moses Schönfinkel developed currying before Curry

[2] $\rightarrow$ associates to right

λ-Calculus

λ-Calculus

- Developed by Alonzo Church and his doctoral student Stephen Cole Kleene in the 1930
- Can represents all computable functions
- Has equal power as of Turing Machine

Source: *λ- Calculus Overview*

Importance of λ-Calculus

- Uncomplicated syntax and semantics provide an excellent vehicle for studying the meaning of programming language concepts
- All functional programming languages can be viewed as syntactic variations of the λ-calculus
- Denotational semantics is based on the λ-calculus and expresses its definitions using the higher-order functions of the λ-calculus

Concept of λ

- A function is a mapping from the elements of a domain set to the elements of a codomain set given by a rule
- Example,

$$cube : Integer \rightarrow Integer$$

where

$$cube(n) = n^3$$

- Questions:
  - What is the value of the identifier *cube*?
  - How can we represent the object bound to *cube*?
  - Can this function be defined without giving it a name?

Concept of λ

- λ-**notation** for an anonymous function:

$$\lambda n.\ n^3$$

  defines the function that maps each n in the domain to n^3
- Expression represented by

$$\lambda n.\ n^3$$

  is the value bound to the identifier *cube*
- To represent the function evaluation $cube(2) = 8$, we use the following λ-calculus syntax:

$$(\lambda n.\ n^3\ 2) \Rightarrow 2^3 \Rightarrow 8$$

Concept of λ: Parallel in C / C++

- **Function Abstraction**
  - *Mathematical Notation*: $cube : Integer \rightarrow Integer$
    $cube(n) = n^3$
    $n \vdash n^3$
  - *λ-notation*:
    $\lambda n.\ n^3 = \lambda n.\ (n * n * n)$
  - *C Function*:

```
int cube(int n) { return n * n * n; }
```

- **Function Application**
  - *Mathematical Notation*: $cube(2) = 8$
  - *λ-notation*:
    $(\lambda n.\ n^3)\ 2 \equiv 2 * 2 * 2 = 8$
  - *C Function*:

```
int n_cube = cube(2);
```

Concept of λ

- The number and order of the parameters to the function are specified between the λ symbol and an expression
- Example: Expression

$$n^2 + m$$

is ambiguous as the definition of a function rule:

$$(3, 4) \vdash 3^2 + 4 = 13$$

or

$$(3, 4) \vdash 4^2 + 3 = 19$$

Concept of λ

- λ-notation resolves the ambiguity by specifying the order of the parameters:

$$\lambda n.\ \lambda m.\ n^2 + m, \text{ that is, } (3,4) \vdash 3^2 + 4 = 13$$

$$\lambda m.\ \lambda n.\ n^2 + m, \text{ that is, } (3,4) \vdash 4^2 + 3 = 19$$

  Notationally (by left-to-right order 3 binds to n and 4 binds to m):
  $(\lambda n.\ \lambda m.\ (n^2 + m)\ 3\ 4) = (\lambda m.\ (3^2 + m)\ 4) = (\lambda m.\ (9 + m)\ 4) = (9 + 4) = 13$
- Most functional programming languages allow anonymous functions as values
- Example: The function $\lambda n.n^3$ is represented as
  - Scheme: $(lambda\ (n)(*\ n\ n\ n))$
  - Standard ML: $fn\ n \Rightarrow n * n * n$

Concept of λ: Parallel in C / C++

- **Function Abstraction**
  - *Mathematical Notation*: $f, g : Integer \times Integer \rightarrow Integer$
    $f(n, m) = n^2 + m$
    $g(m, n) = n^2 + m$
  - *λ-notation*:
    $\lambda n.\ \lambda m.\ n^2 + m$
    $\lambda m.\ \lambda n.\ n^2 + m$
  - *C Function*:

```
int f(int n, int m) { return n * n + m; }
int g(int m, int n) { return n * n + m; }
```

- **Function Application**
  - *Mathematical Notation*:
    $f(3, 4) = 13$
    $g(3, 4) = 19$
  - *λ-notation*:
    $(\lambda n.\ \lambda m.\ n^2 + m)\ 3\ 4 = 3^2 + 4 = 13$
    $(\lambda m.\ \lambda n.\ n^2 + m)\ 3\ 4 = 4^2 + 3 = 19$
  - *C Function*:

```
int r_f = f(3, 4); // 13
int r_g = g(3, 4); // 19
```

Syntax of λ-Calculus

Syntax of λ Calculus

λ-expressions come in four varieties:

- Variables
  - Usually, lowercase letters
- Predefined Constants
  - Act as values and operations
  - Allowed in an impure or applied λ-calculus
- Function Applications
  - Combinations
- λ-Abstractions
  - Function definitions

Syntax of λ Calculus

BNF Syntax of λ-Calculus:

$$
\begin{array}{lcll}
< expression > & ::= & < variable > & ; \text{lowercase identifiers} \\
 & | & < constant > & ; \text{predefined objects} \\
 & | & (< expression >< expression >) & ; \text{combinations} \\
 & | & (\lambda < variable > . < expression >) & ; \text{abstractions}
\end{array}
$$

In short:

$$
\begin{array}{lcll}
e & ::= & v & ; \text{variables / constants} \\
 & | & (e\ e) & ; \text{function application} \\
 & | & (\lambda v.e) & ; \text{function abstractions}
\end{array}
$$

Syntax of λ Calculus

- Identifiers of more than one letter may stand as variables and constants
- Pure λ-calculus
  - has no predefined constants, but
  - it still allows the definition of all of the common constants and functions of arithmetic and list manipulation
- Predefined constants
  - Numerals (for example, 34),
  - add (addition), mul (multiplication), succ (successor function), and sqr (squaring function)

Syntax of λ Calculus

- For a list in Lisp

head or *car*[3]	returns the first item of the list it is called on
tail or *cdr*[4]	returns a new list consisting of all but the first item of the list it is called on
cons	takes an argument and returns a new list whose head is the argument and whose tail is the list it is called on
isEmpty	returns true if the list it is called on is the empty list, returns false otherwise

- $(cons\ y\ nil) = (y)$
- $(cons\ x\ (y)) = (x\ y)$
- $(car\ (cons\ x\ y)) = x$
- $(cdr\ (cons\ x\ y)) = (y)$

[3] **C**ontents of the **A**ddress part of **R**egister number
[4] **C**ontents of the **D**ecrement part of **R**egister number

Free and Bound Variable

- In an abstraction, the variable named is referred to as the **bound** variable and the associated λ-expression is the **body** of the abstraction
- In an expression of the form:

$$\lambda v.\ e$$

  occurrences of variable v in expression e are **bound**
- All occurrences of other variables are **free**
- Example:

$$((\lambda x.\ \lambda y.\ (xy))(yw))$$

  - x, and y are **bound** in first part
  - y, and w are **free** in second part

Concept of λ: Parallel in C / C++

- **Function Abstraction**
  - *Mathematical Notation*: $f : Integer \times Integer \times Integer \times Integer \rightarrow Integer$
    $f(n, l, s, g) = n + l + s + g$
  - *λ-notation*:
    $\lambda n.\ \lambda l.\lambda s.\ n + l + s + g$ // Bound $= n, l, s$, Free $= g$
  - *C Function*:

```
int g;                          // Free, global g - to be set from environment
int f(int n) {                  // Bound, parameter n
    int l = 3;                  // Bound, automatic local l
    static int s = 7;           // Bound, static local s

    return n + l + s + g;
}
```

- **Function Application**
  - *Mathematical Notation*: $f(2, 3, 7, g) = 12 + g$
  - *λ-notation*:
    $(\lambda n.\ \lambda l.\lambda s.\ n + l + s + g)\ 2\ 3\ 7 = 2 + 3 + 7 + g = 12 + g$
  - *C Function*:

```
g = 5;  // Free global g set from environment to 5
f(2);   // 17
g = 3;  // Free global g set from environment to 3
f(2);   // 15
```

Function Application

- With a function application $(E_1\ E_2)$, it is expected that E_1 evaluates to a predefined function (a constant) or an abstraction, say $(\lambda x.\ E_3)$, in which case the result of the application will be the evaluation of E_3 after every **free** occurrence of x in E_3 has been replaced by E_2

$$(\lambda n.\ n^3\ 2) \Rightarrow 2^3 \Rightarrow 8$$

$$(\lambda n.\ (*\ (*\ n\ n)\ n)\ 2) \Rightarrow 2^3 \Rightarrow 8$$

- In a combination $(E_1\ E_2)$, the function or ope**rator** E_1 is called the **rator** and its argument or ope**rand** E_2 is called the **rand**

Notation for λ-expressions

- Uppercase letters and identifiers beginning with capital letters will be used as meta-variables ranging over λ-expressions

Notation for λ-expressions

- Function application associates to the left

$$E_1 \ E_2 \ E_3$$

means

$$((E_1 \ E_2) \ E_3)$$

Notation for λ-expressions

- The scope of $\lambda <$ *variable* $>$ in an abstraction extends as far to the right as possible:

$$\lambda x.\ E_1\ E_2\ E_3$$

  means

$$(\lambda x.\ (E_1\ E_2\ E_3)) \text{ and not } ((\lambda x.\ E_1\ E_2)\ E_3)$$

- **Application** has a higher precedence than **Abstraction**
- Parentheses are needed for

$$(\lambda x.\ E_1\ E_2)\ E_3$$

  where E_3 is intended to be an argument to the function

$$\lambda x.\ E_1\ E_2$$

  and not part of the body of the function as above

Notation for λ-expressions

- An abstraction allows a list of variables that abbreviates a series of λ abstractions

$$\lambda x\ y\ z.\ E$$

means

$$(\lambda x.\ (\lambda y.\ (\lambda z.\ E)))$$

Notation for λ-expressions

- Functions defined as λ-expression abstractions are anonymous, so the λ-expression itself denotes the function
- As a notational convention, λ-expressions may be named using the syntax

$$\textit{define} \;\; < \textit{name} > = < \textit{expression} >$$

Module M02

Partha Pratim Das

Notation for λ-expressions

- For example, given

$$define\ \ Twice \ = \ \lambda f.\, \lambda x.\, f(f\ x)$$

it follows that

$$(Twice\ (\lambda n.\ (add\ \ n\ 1))\ 5) \ = \ 7$$

Concept of λ: Parallel in C / C++

- **Variables**: n
- **Constants**: $*$: $Integer \rightarrow Integer$ (binary multiplication), 2
- **Function Abstraction**
  - *Mathematical Notation*: $cube : Integer \rightarrow Integer$
    $cube(n) = n^3$
  - *λ-notation*:
    $cube \equiv \lambda n.\ n^3 = \lambda n.\ (n * n * n)$ // Untyped λ
    $cube \equiv \lambda(n : int).\ n^3$ // Typed λ, return type inferred
  - *C Function*:
    ```
    int cube(int n) { return n * n * n; } // return type explicit
    ```
  - *C++ λ Function*:
    ```
    auto cube = [](int n) { return n * n * n; }; // return type inferred
    auto cube = [](int n) -> int { return n * n * n; }; // return type explicit
    ```
- **Function Application**
  - *Mathematical Notation*: $cube(2) = 8$
  - *λ-notation*:
    $(\lambda n.\ n^3)\ 2 \equiv 2 * 2 * 2 = 8$
  - *C / C++ λ Function*:
    ```
    int n_cube = cube(2);
    ```

Notation for λ-expressions: Example

- Group the terms in the following λ-expression

$$(\lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x))\ (\lambda g.\ \lambda y.\ g\ y)$$

- λ Abstractions

$$\begin{array}{ll} (\lambda x.\ f\ (n\ f\ x)) & (\lambda y.\ g\ y) \\ (\lambda f.\ (\lambda x.\ f\ (n\ f\ x))) & (\lambda g.\ (\lambda y.\ g\ y)) \\ (\lambda n.\ (\lambda f.\ (\lambda x.\ f\ (n\ f\ x)))) & \end{array}$$

- Completely parenthesized expression:

$$((\lambda n.\ (\lambda f.\ (\lambda x.\ (f\ ((n\ f)\ x)))))\ (\lambda g.\ (\lambda y.\ (g\ y))))$$

Examples of λ-expressions

- Elementary
  - Identity Function
  - Successor Function
  - Constant Function
- Composition
  - Application
    - twice
    - thrice
  - Composition
- Church Boolean
  - Selector Function ($TRUE$, $FALSE$)
  - Conditional Test IF
  - Boolean Algebra
- Church Numerals
- Recursion
  - Self Application
  - Y Combinator
    - factorial

λ-expressions: Identity Function

- The λ-expression

$$ID = \lambda x.\ x$$

  denotes the **identity function** in the sense that

$$((\lambda x.\ x)\ E) = E$$

  for any λ-expression E
  - Identity function has type $A \rightarrow A$ for every type A
  - Functions that allow arguments of many types, such as this identity function, are known as **polymorphic operations**
  - The λ-expression $(\lambda x.\ x)$ acts as an identity function on the set of integers, on a set of functions of some type, or on any other kind of object
- The token ID is not part of the λ-calculus – just an abbreviation for the term $(\lambda x.\ x)$

λ-expressions: Successor Function

- The λ-expression

$$\lambda n.\ (add\ \ n\ 1)$$

denotes the **successor function** on the integers so that

$$(\lambda n.\ (add\ \ n\ 1))\ 5\ \ = 6$$

- *add* and 1 need to be predefined constants to define this function, and 5 must be predefined to apply the function

λ-expressions: Constant Function

- The λ-expression

$$K = \lambda x.\ \lambda y.\ x$$

  builds a constant function (generator)
- $(K\ 0) = (\lambda x.\ \lambda y.\ x)\ 0 = \lambda y.\ 0 = 0$, is a constant function returning 0
- $(K\ 1) = (\lambda x.\ \lambda y.\ x)\ 1 = \lambda y.\ 1 = 1$, is a constant function returning 1
- $\cdots$
- $(K\ n) = (\lambda x.\ \lambda y.\ x)\ n = \lambda y.\ n = n$, is a constant function returning n
    - $(\lambda y.\ n)\ 0 = 0$
    - $(\lambda y.\ n)\ 12 = 12$
    - $(\lambda y.\ n)\ 935 = 935$
    - $(\lambda y.\ n)\ m = m$, m is a constant

λ-expressions: Application

- The λ-expression

$$apply = \lambda f.\ \lambda x.\ f\ x$$

  takes a function and a value as argument and applies the function to the argument
- Since f is a function and it takes x as an argument, say of type A, then f must be of type $A \rightarrow B$ for some B
- Type of *apply* then is: $(A \rightarrow B) \rightarrow A \rightarrow B$
- $A \rightarrow B$ is a possible type of f, A is the possible type of x, and B is the result type of *apply* which is the same as result type of f

λ-expressions: twice

- The λ-expression

$$twice = \lambda f.\ \lambda x.\ f\ (f\ x)$$

  is similar to *apply* but applies the function f twice
- It applies f to x obtaining a result, and applies f to this result once more
- Unlike *apply*, since f is applied again to the result of f, the argument and result types of f should be the same, say A
- So, the type of *twice* is $(A \rightarrow A) \rightarrow A \rightarrow A$
- If *sqr* is the (predefined) integer function, then

$$((twice\ sqr)\ 3) \Rightarrow (((\lambda f.\ (\lambda x.\ (f\ (f\ x))))\ sqr)\ 3) \Rightarrow$$

$$((\lambda x.\ (sqr\ (sqr\ x)))\ 3) \Rightarrow (sqr\ (sqr\ 3)) \Rightarrow (sqr\ 9) \Rightarrow 81$$

- Similarly, $(twice\ (\lambda n.\ (add\ n\ 1))\ 5)\ =\ 7$

λ-expressions: thrice

- The λ-expression

$$thrice = \lambda f.\ \lambda x.\ f\ (f\ (f\ x))$$

  applies f thrice

- The type of *thrice* is $(A \rightarrow A) \rightarrow A \rightarrow A$
- If *sqr* is the (predefined) integer function, then

$$((thrice\ sqr)\ 3) \Rightarrow (((\lambda f.\ (\lambda x.\ f\ (f\ (f\ x))))\ sqr)\ 3) \Rightarrow$$

$$((\lambda x.\ (sqr\ (sqr\ (sqr\ x))))\ 3) \Rightarrow (sqr\ (sqr\ (sqr\ 3))) \Rightarrow$$

$$(sqr\ (sqr\ 9)) \Rightarrow (sqr\ 81) \Rightarrow 6561$$

- Similarly, $(thrice\ (\lambda n.\ (add\ n\ 1))\ 5)\ =\ 8$

λ-expressions: Composition

- The λ-expression

$$comp = \lambda g.\ \lambda f.\ \lambda x.\ g\ (f\ x)$$

  is the mathematical composition: $(comp\ g\ f) \equiv g \circ f$
- If f is of type $A \to B$ and g is of type $B \to C$, then type of $g \circ f$ is $A \to C$
- Given an argument, $g \circ f$ first applies f to the argument and then applies g to the result of this application
- The type of *comp* is $(B \to C) \to (A \to B) \to (A \to C)$
- $twice\ f \equiv (comp\ f\ \ f)$
- $thrice\ f \equiv (comp\ f\ (comp\ f\ \ f)) \equiv (comp\ (comp\ f\ \ f)\ f)$

λ-expressions: Selector Function

- The λ-expression

$$TRUE = fst = \lambda x.\ \lambda y.\ x$$

  denotes the **fst selector function**
- It takes two arguments and returns the first argument as the result (ignoring the second argument)
- Note: $(\lambda x.\ \lambda y.\ x)\ M\ N \equiv (\lambda y.\ M)\ N \equiv M$
  - The **fst** function is first given an argument, say of type A (of M), and it returns a function
  - This (returned) function takes another argument, say of type B (of N), and returns the original first argument (of type A)
  - Hence, the type of **fst** is $A \rightarrow (B \rightarrow A)$
- The token *TRUE* is not part of the *lambda*-calculus – just an abbreviation for the term $(\lambda x.\ \lambda y.\ x)$

λ-expressions: Selector Function

- The λ-expression

$$FALSE = snd = \lambda x.\ \lambda y.\ y$$

 denotes the **snd selector function**
- It takes two arguments and returns the second argument as the result (ignoring the first argument)
- Note: $(\lambda x.\ \lambda y.\ y)\ M\ \ N \equiv (\lambda y.\ y)\ N \equiv N$
  - The **snd** function is first given an argument, say of type A (of M), and it returns a function
  - This (returned) function takes another argument, say of type B (of N), and returns the same argument (of type B)
  - Hence, it has a type $A \to (B \to B)$
- The token $FALSE$ is not part of the *lambda*-calculus – just an abbreviation for the term $(\lambda x.\ \lambda y.\ y)$

λ-expressions: Conditional Test *IF*

- *IF* should take three arguments b, t, f, where b is a Boolean value and t, f are arbitrary terms
- The function should return t if $b = TRUE$ and f if $b = FALSE$
- Now $(TRUE\ t\ f) \equiv t$ and $(FALSE\ t\ f) \equiv f$
- *IF* has to apply its Boolean argument to the other two arguments:

$$IF = \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$$

- If b is not of Boolean type, the result is undefined

λ-expressions: Boolean Algebra

- Boolean operators can be defined using IF, $TRUE$, and $FALSE$:

$$
\begin{aligned}
AND &= \lambda b.\ \lambda b'.\ IF\ b\ b'\ FALSE \\
OR &= \lambda b.\ \lambda b'.\ IF\ b\ TRUE\ b' \\
NOT &= \lambda b.\ IF\ b\ FALSE\ TRUE
\end{aligned}
$$

- Using the above definitions prove the De Morgan's Laws of Boolean Algebra

λ-expressions: Practice

- $(\lambda z.\ z)(\lambda y.\ y\ y)(\lambda x.\ x\ a)$
- $(\lambda z.\ z)(\lambda z.\ z\ z)(\lambda z.\ z\ y)$
- $(\lambda x.\ \lambda y.\ x\ y\ y)(\lambda a.\ a)\ b$
- $((\lambda x.\ \lambda y.\ x\ y\ y)(\lambda y.\ y)\ y$
- $(\lambda x.\ x\ x)(\lambda y.\ y\ x)\ z$

Church Numerals: Links

- http://www.cs.unc.edu/~stotts/723/Lambda/church.html
- http://www.cs.cornell.edu/courses/cs312/2008sp/recitations/rec26.html
- http://www.shlomifish.org/lecture/Lambda-Calculus/slides/lc_church_ops.scm.html
- http://okmij.org/ftp/Computation/lambda-calc.html
- https://en.wikipedia.org/wiki/Church_encoding

Source: http://www.wikibooks.org; Wikibooks home

Church Numerals

- Natural numbers are non-negative
- Given a successor function, *succ*, which adds one, we can define the natural numbers in terms of *zero* (0) and *succ*:

$$
\begin{aligned}
1 &= (succ\ 0) \\
2 &= (succ\ 1) \\
&= (succ\ (succ\ 0)) \\
3 &= (succ\ 2) \\
&= (succ\ (succ\ (succ\ 0))) \\
\ldots
\end{aligned}
$$

Church Numerals

- A number n will be that number of successors of zero
- If f and x are λ-terms, and $n > 0$ a natural number, write $f^n x$ for the term $f(f(...(f\ x)...))$, where f occurs n times
- For each natural number n, we define a λ-term $\overline{n}$, called the n^{th} **Church Numeral**, as

$$\overline{n} = \lambda f.\ \lambda x.\ f^n\ x$$

- First few Church numerals are:

$$\begin{aligned}
C_0 = \overline{0} &= \lambda f.\ \lambda x.\ x \\
C_1 = \overline{1} &= \lambda f.\ \lambda x.\ (f\ x) \\
C_2 = \overline{2} &= \lambda f.\ \lambda x.\ (f\ (f\ x)) \\
C_3 = \overline{3} &= \lambda f.\ \lambda x.\ (f\ (f\ (f\ x))) \\
C_n = \overline{n} &= \lambda f.\ \lambda x.\ f^n\ x
\end{aligned}$$

Church Numerals: Successor

- The successor is defined as:

$$succ = \lambda n.\ \lambda f.\ \lambda x.\ (f\ ((n\ f)\ x))$$

- Apply f on n applications of f (that is, $\overline{n}$)
- Hence it leads to $n+1$ applications of f (that is, $\overline{n+1}$):

$$\begin{aligned}
succ\ \overline{0} &= (\lambda n.\ \lambda f.\ \lambda x.\ (f\ ((n\ f)\ x)))(\lambda f.\ \lambda x.\ x) \\
&= \lambda f.\ \lambda x.\ (f\ (((\lambda f.\ \lambda x.\ x)\ f)\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ (((\lambda g.\ \lambda y.\ y)\ f)\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ ((\lambda y.\ y)\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ x) \\
&= \overline{1}
\end{aligned}$$

Church Numerals: Successor

$$
\begin{aligned}
succ\ \overline{1} &= (\lambda n.\ \lambda f.\ \lambda x.\ (f\ ((n\ f)\ x)))(\lambda f.\ \lambda x.\ (f\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ (((\lambda f.\ \lambda x.\ (f\ x))\ f)\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ (((\lambda g.\ \lambda y.\ (g\ y))\ f)\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ ((\lambda y.\ (f\ y))\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ (f\ x)) \\
&= \overline{2}
\end{aligned}
$$

Church Numerals: Successor

- $succ = \lambda n.\ \lambda f.\ \lambda x.\ (f\ ((n\ f)\ x))$

$$
\begin{aligned}
succ\ \overline{n} &= (\lambda n.\ \lambda f.\ \lambda x.\ (f\ ((n\ f)\ x)))\ \overline{n} \\
&= \lambda f.\ \lambda x.\ (f\ ((\overline{n}\ f)\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ (((\lambda f.\ \lambda x.\ (f^n\ x))\ f)\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ (((\lambda g.\ \lambda y.\ (g^n\ y))\ f)\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ ((\lambda y.\ (f^n\ y))\ x)) \\
&= \lambda f.\ \lambda x.\ (f\ (f^n\ x)) \\
&= \lambda f.\ \lambda x.\ (f^{n+1}\ x) \\
&= \overline{n+1}
\end{aligned}
$$

Church Numerals: Addition

- $succ = \lambda n.\ \lambda f.\ \lambda x.\ (f\ ((n\ f)\ x))$ goes one step from $\overline{n}$
- For addition of $\overline{m}$ with $\overline{n}$, we need to go $\overline{n}$ steps from $\overline{m}$
- The addition is defined as:

$$add = \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ ((((m\ succ)\ n)\ f)\ x)$$

- Compute $\overline{n}$ successor of $\overline{m}$. Apply n applications of f on $\overline{m}$
- $succ \equiv add\ \overline{1}$

Church Numerals: Addition

- Example:

$$
\begin{aligned}
(add\ \overline{2}\ \overline{2}) &= ((add\ \overline{2})\ \overline{2}) \\
&= ((\lambda m.\lambda n.\lambda f.\lambda x.((((m\ succ)\ n)\ f)\ x)\ \overline{2})\ \overline{2}) \\
&= (\lambda n.\lambda f.\lambda x.((((\overline{2}\ succ)\ n)\ f)\ x) \\
&= \lambda f.\lambda x.((((\overline{2}\ succ)\ \overline{2})\ f\ x) \\
&= \lambda f.\lambda x.((((\lambda g.\lambda y.(g\ (g\ y))\ succ)\ \overline{2})\ f\ x) \\
&= \lambda f.\lambda x.(((\lambda y.(succ\ (succ\ y))\ \overline{2})\ f)\ x) \\
&= \lambda f.\lambda x.(((succ\ (succ\ \overline{2}))\ f)\ x) \\
&= \lambda f.\lambda x.(((succ\ \overline{3})\ f)\ x) \\
&= \lambda f.\lambda x.(((\overline{4})\ f)\ x) \\
&= \overline{4}
\end{aligned}
$$

Church Numerals: Multiplication

- The multiplication function is defined as:

$$mul = \lambda m.\ \lambda n.\ \lambda x.\ (m\ (n\ x))$$

- Apply n applications of f $(\overline{n})$ m times

Church Numerals: Multiplication

- Example:

$$
\begin{aligned}
(mult\ \overline{2}\ \overline{3}) &= ((mult\ \overline{2})\ \overline{3}) \\
&= ((\lambda m.\ \lambda n.\ \lambda x.\ (m\ (n\ x))\ \overline{2})\ \overline{3}) \\
&= (\lambda n.\ \lambda x.\ (\overline{2}\ (n\ x))\ \overline{3}) \\
&= \lambda x.\ (\overline{2}\ (\overline{3}\ x)) \\
&= \lambda x.\ (\overline{2}\ (\lambda g.\ \lambda y.\ (g\ (g\ (g\ y)))\ x)) \\
&= \lambda x.\ (\overline{2}\ (\lambda y.\ (x\ (x\ (x\ y))))) \\
&= \lambda x.\ (\lambda f.\ \lambda z.\ (f\ (f\ z)))\ \lambda y.\ (x\ (x\ (x\ y)))) \\
&= \lambda x.\ \lambda z.\ (\lambda y.\ (x\ (x\ (x\ y)))\ (\lambda y.\ (x\ (x\ (x\ y)))\ z)) \\
&= \lambda x.\ \lambda z.\ (\lambda y.\ (x\ (x\ (x\ y)))\ (x\ (x\ (x\ z)))) \\
&= \lambda x.\ \lambda z.\ (x\ (x\ (x\ (x\ (x\ (x\ z)))))) \\
&= \overline{6}
\end{aligned}
$$

Church Numerals: Exponentiation

- The exponentiation (n^m) function is defined as:

$$exp = \lambda m.\ \lambda n.\ (m\ n)$$

- Example:

$$
\begin{aligned}
(exp\ \overline{2}\ \overline{3}) &= ((exp\ \overline{2})\ \overline{3}) \\
&= ((\lambda m.\ \lambda n.\ (m\ n)\ \overline{2})\ \overline{3}) \\
&= (\lambda n.\ (\overline{2}\ n)\ \overline{3}) \\
&= (\overline{2}\ \overline{3}) \\
&= (\lambda f.\ \lambda x.\ (f(f\ x))\ \overline{3}\ ) \\
&= \lambda x.\ (\overline{3}\ (\overline{3}\ x)) \\
&= \lambda x.\ (\overline{3}\ (\lambda g.\ \lambda y.\ (g\ (g\ (g\ y)))\ x)) \\
&= \lambda x.\ (\overline{3}\ \lambda y.\ (x\ (x\ (x\ y)))) \\
&= \lambda x.\ (\lambda g.\ \lambda z.\ (g\ (g\ (g\ z)))\ \lambda y.\ (x\ (x\ (x\ y)))) \\
&= \lambda x.\ \lambda z.\ (\lambda y.\ (x\ (x\ (x\ y)))(\lambda y.\ (x\ (x\ (x\ y)))(\lambda y.\ (x\ (x\ (x\ y)))\ z))) \\
&= \lambda x.\ \lambda z.\ (\lambda y.\ (x\ (x\ (x\ y)))(\lambda y.\ (x\ (x\ (x\ y)))(x\ (x\ (x\ z))))) \\
&= \lambda x.\ \lambda z.\ (\lambda y.\ (x\ (x\ (x\ y)))(x\ (x\ (x\ (x\ (x\ (x\ z))))))) \\
&= \lambda x.\ \lambda z.\ (x\ (x\ (x\ (x\ (x\ (x\ (x\ (x\ (x\ z))))))))) \\
&= \overline{9}
\end{aligned}
$$

Church Numerals: Predecessor

- The predecessor is defined as:

$$
\begin{aligned}
\mathit{pair} &= \lambda x.\ \lambda y.\ \lambda f.\ ((f\ x)\ y) \\
\mathit{prefn} &= \lambda f.\ \lambda p.\ ((\mathit{pair}\ (f\ (p\ \mathit{first})))\ (p\ \mathit{first})) \\
\mathit{pred} &= \lambda n.\ \lambda f.\ \lambda x.\ (((n\ (\mathit{prefn}\ f))\ (\mathit{pair}\ x\ x))\ \mathit{second})
\end{aligned}
$$

- Example: Show: $(\mathit{pred}\ \overline{3}) = \overline{2}$
- Note:
  - *Kleene discovered how to express the operation of subtraction within Church's scheme (yes, Church was unable to implement subtraction and subsequently division, within that calculus)!*
  - *Other landmarks then followed, such as the recursive function Y.*
  - *In 1937 Church and Turing, independently, showed that every computable operation (algorithm) can be achieved in a Turing machine and in the Lambda Calculus, and therefore the two are equivalent.*
  - *Similarly Godel introduced his description of computability, again independently, in 1929, using a third approach which was again shown to be equivalent to the other 2 schemes.*
  - *It appears that there is a "platonic reality" about computability. That is, it was "discovered" (3 times independently) rather than "invented". It appears to be natural in some sense.*

Source: *Natural Numbers as Church Numerals*

Church Numerals
Practice Problems

- Show: $add\ \overline{2}\ \overline{3} = \overline{5}$
- Show: $mul\ \overline{2}\ \overline{3} = \overline{6}$
- Show: $exp\ \overline{3}\ \overline{2} = \overline{8}$
- Show: $add\ \overline{n}\ \overline{0} = \overline{n}$
- Show: $mul\ \overline{n}\ \overline{1} = \overline{n}$
- Show: $exp\ \overline{0}\ \overline{n} = \overline{1}$
- Prove: *add* and *mul* are commutative
- Prove: *add* and *mul* are associative
- Prove: $mul\ \overline{c}\ (add\ \overline{a}\ \overline{b}) = add\ (mul\ \overline{c}\ \overline{a})\ (mul\ \overline{c}\ \overline{b})$
- Define: $sub\ \overline{m}\ \overline{n}$, where $sub(m, n) = (m - n \geq 0)?\ m - n :\ 0$
- Define: $div\ \overline{m}\ \overline{n}$, where $div(m, n) = (m - m\ \%\ n)/n$

λ-expressions: Self Application

- The λ-expression
$$sa = \lambda x.\ x\ x$$
takes an argument x, which is apparently a function and applies the function to itself and returns whatever is the result
- x is a function that can take itself as an argument!
- $(sa\ id) = id\ id = (\lambda x.\ x)\ id\ =\ id$
- $(sa\ fst) = fst\ fst = (\lambda x.\ \lambda y.\ x)\ fst\ =\ \lambda y.\ fst$
- $(sa\ snd) = snd\ snd = (\lambda x.\ \lambda y.\ y)\ snd\ =\ id$
- $(sa\ twice) = twice\ twice = (\lambda f.\ \lambda x.\ f\ (f\ x))\ twice\ = (\lambda x.\ twice\ (twice\ x))\ = comp\ twice\ twice$
- Finally! $(sa\ sa) = sa\ sa = (\lambda x.\ x\ x)\ sa = sa\ sa$
  - Infinite Loop in λ-Calculus, denoted by Ω

λ-expressions: Y Combinator

- The λ-expression

$$Y = \lambda u.\ (\lambda x.\ u\ (x\ x))\ (\lambda x.\ u\ (x\ x))$$

is called the **Y combinator**

- Consider:

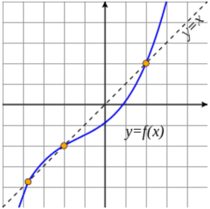
$$\begin{aligned} Y\ t &= (\lambda x.\ t\ (x\ x))\ (\lambda x.\ t\ (x\ x)) \\ &= (\lambda y.\ t\ (y\ y))\ (\lambda x.\ t\ (x\ x)) \\ &= t\ ((\lambda x.\ t\ (x\ x))\ (\lambda x.\ t\ (x\ x))) \\ &= t\ (Y\ t) \end{aligned}$$

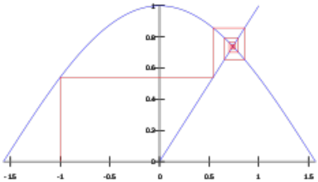
- $(Y\ t)$ is function t applied to itself! Repeatedly unfolding:

$$Y\ t = t\ (Y\ t) = t\ (t\ (Y\ t)) = t\ (t\ (t\ (Y\ t))) = \cdots$$

- Another form of an infinite loop? No – it is quite useful
- Used to encode recursive functions in λ-calculus

λ-expressions: Fixed Point

- The fixed point of a function $f : A \rightarrow A$ is a value $x \in A$ such that $f(x) = x$
- Examples:
  - $f(x) = x^2 - 3x + 4$ has a fixed point $f(2) = 2$
  - $f(x) = x^3$ has 3 fixed points $f(-1) = -1$, $f(0) = 0$, and $f(+1) = +1$
  - $\cos x = x$ has a fixed point $\cos 0.739085133 = 0.739085133$
  - 



  - $f(x) = \frac{x}{2} + \frac{1}{x}$ has a fixed point $f(\sqrt{2}) = \sqrt{2}$. Starting with $x_0 = 1$, we have: $x_0 = 1$, $x_1 = 1/2 + 1/1 = 3/2$, $x_2 = 3/4 + 2/3 = 17/12$, $x_3 = 17/24 + 12/17 \approx 1.41421569$, $\cdots$
  - Not all functions have fixed points:
    - $f(x) = x + 1$
    - Collatz Sequence $f(n) = (n \mod 2)?\ 3n + 1 :\ n/2$ cycles between 4, 2, 1.

λ-expressions: Fixed Point

- The fixed point of a function $f : N \rightarrow N$ is a value $x \in N$ such that

$$f\ x = x$$

- Since $y\ f = f\ (y\ f)$
  - $(y\ f)$ is a fixed point of the function f
  - Hence, y is called the **fixed point combinator**
    - When y is applied to a function, it answers a value x in that function's domain
    - When we apply the function to x, we get x

λ-expressions: Y Combinator – factorial

- $define\ \ factorial = \lambda n.\ if\ \ (= n\ 1)\ 1\ (*n\ (factorial\ \ (-n\ 1)))$
- The above is circular. So rewrite as:

$$\begin{array}{lcl} define & factorial & = \underline{T}\ factorial \\ define & \underline{T} & = \lambda f.\ \lambda n.\ if\ \ (= n\ 1)\ 1\ (*n\ (f\ \ (-n\ 1))) \end{array}$$

- $Y\ \ \underline{T} = \underline{T}\ (Y\ \ \underline{T})$, is then the *factorial*

$$factorial = (Y\ \ \underline{T})$$

λ-expressions: Y Combinator – factorial

- *define* $\underline{T} = \lambda f.\ \lambda n.\ if\ (= n\ 1)\ 1\ (*n\ (f\ (-n\ 1)))$
- Sample:

$$
\begin{array}{lcl}
(Y\ \ T)\ 1 & = & T\ (Y\ \ T)\ 1 = \lambda n.\ if\ (= n\ 1)\ 1\ (*n\ ((Y\ \ T)\ (-n\ 1)))\ 1 \\
& = & if\ (= 1\ 1)\ 1\ (*\ 1\ ((Y\ \ T)\ (-1\ 1))) \\
& = & 1 \\
(Y\ \ T)\ 2 & = & T\ (Y\ \ T)\ 2 = \lambda n.\ if\ (= n\ 1)\ 1\ (*n\ ((Y\ \ T)\ (-n\ 1)))\ 2 \\
& = & if\ (= 2\ 1)\ 1\ (*\ 2\ ((Y\ \ T)\ (-2\ 1))) \\
& = & (*\ 2\ ((Y\ \ T)\ 1)) \\
& = & (*\ 2\ 1) \\
& = & 2 \\
(Y\ \ T)\ 3 & = & T\ (Y\ \ T)\ 3 = \lambda n.\ if\ (= n\ 1)\ 1\ (*n\ ((Y\ \ T)\ (-n\ 1)))\ 3 \\
& = & if\ (= 3\ 1)\ 1\ (*\ 3\ ((Y\ \ T)\ (-3\ 1))) \\
& = & (*\ 3\ ((Y\ \ T)\ 2)) \\
& = & (*\ 3\ 2) \\
& = & 6
\end{array}
$$

λ-expressions: Fibonacci Function

- The Fibonacci function in the λ-calculus

$$
\begin{aligned}
fibo(n) &= fibo(n-1) + fibo(n-2), && if\ \ n > 1 \\
&= 1, && if\ \ n = 1 \\
&= 0, && if\ \ n = 0
\end{aligned}
$$

- Using the Y combinator, we can define Fibonacci function in the λ-calculus
- Define function $\underline{F}$, whose fixed-point will be *Fibonacci*:

$$\underline{F} = \lambda f.\ \lambda n.\ (if (= 0\ n)\ 0\ (if (= 1\ n)\ 1\ (+\ (f\ (-\ n\ 1)\ f\ (-\ n\ 2)))))$$

- Then take the fixed point of $\underline{F}$:

$$fibo = (Y\ \underline{F})$$

- Show: $fibo(5) = 5$

λ-expressions: Ackermann Function

- The Ackermann function $A(x, y)$ is defined for integers x and y by:

$$
\begin{array}{lcll}
A(x, y) & = & y + 1, & \textit{if } x = 0 \\
 & = & A(x-1, 1), & \textit{if } y = 0 \\
 & = & A(x-1, A(x, y-1)), & \text{otherwise}
\end{array}
$$

Special values for x include the following:

$$
\begin{array}{lcl}
A(0, y) & = & y + 1 \\
A(1, y) & = & y + 2 \\
A(2, y) & = & 2 * y + 3 \\
A(3, y) & = & 2^{y+3} - 3 \\
A(4, y) & = & 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} - 3
\end{array}
$$

λ-expressions: Ackermann Function

- The Ackermann function grows faster than any primitive recursive function, that is: for any primitive recursive function f, there is an n such that

$$A(n, x) > f\ x$$

- So A cannot be primitive recursive
- Can we define A in the λ-calculus?

λ-expressions: Ackermann Function

- The Ackermann function in the λ-calculus

$$
\begin{array}{rcll}
A(x, y) & = & y + 1, & \textit{if } x = 0 \\
 & = & A(x-1, 1), & \textit{if } y = 0 \\
 & = & A(x-1, A(x, y-1)), & \text{otherwise}
\end{array}
$$

- Using the Y combinator, we can define Ackermann function in the λ-calculus, even though it is not primitive recursive!
- Define function aG, whose fixed-point will be *ackermann*:

$$(\mathit{if}\,(=0\ x)\ (\mathit{succ}\ y)\ (\mathit{if}\,(=0\ y)\ (f\ (\mathit{pred}\ x)\ 1)\ (f\ (\mathit{pred}\ x)\ (f\ x\ (\mathit{pred}\ y)))))$$

- Then take the fixed point of aG:

$$\mathit{ackermann} = (y\ aG)$$

Multi-variable Functions

- λ-calculus directly permits functions of a single variable only
- The abstraction mechanism allows for only one parameter at a time
- Many useful functions, such as binary arithmetic operations, require more than one parameter; for example,

$$sum(a, b) = a + b$$

  matches the syntactic specification

$$sum : N \times N \rightarrow N$$

  where N denotes the natural numbers
- λ-calculus admits two solutions for this

Multi-variable Functions: *Using Ordered Pairs*

- Allow ordered pairs as λ-expressions
- Use the notation $< x, y >$, and define the addition function on pairs:

$$sum < a, b > = a + b$$

  - Pairs can be provided by using a predefined *cons* operation as in Lisp, or
  - Pairing operation can be defined in terms of primitive λ-expressions in the pure λ-calculus

Multi-variable Functions: *Using Curried Functions*

- Use the curried version of the function with the property that arguments are supplied one at a time[5]:

$$add: \ N \rightarrow N \rightarrow N$$

  where $add\ a\ b\ =\ a\ +\ b$

- Now

$$(add\ a): \ N \rightarrow N$$

  is a function with the property that

$$((add\ a)\ b)\ =\ a\ +\ b$$

  Thus, the successor function can be defined as $(add\ 1)$

[5] $\rightarrow$ associates to the right and function application associates to the left

Curried Functions

- The operations of currying and uncurrying a function can be expressed in the λ-calculus as

$$\begin{aligned} define\ Curry &= \lambda f.\ \lambda x.\ \lambda y.\ f < x, y > \\ define\ Uncurry &= \lambda f.\ \lambda p.\ f\ (head\ p)(tail\ p) \end{aligned}$$

  provided the pairing operation $< x, y >= (cons\ x\ y)$ and the functions $(head\ p)$ and $(tail\ p)$ are available, either as predefined functions or as functions defined in the pure λ-calculus

- The two versions of the addition operation are related as:

$$Curry\ sum\ =\ add \text{ and } Uncurry\ add\ =\ sum$$

Higher Order Functions

- Currying permits the **partial application** of a function
- Consider an example using *Twice* that takes advantage of the currying of functions:

$$define\ Twice\ =\ \lambda f.\ \lambda x.\ f(f\ x)$$

- *Twice* is a polymorphic function as it may be applied to any function and element as long as that element is in the domain of the function and its image under the function is also in that domain
- The mechanism that allows functions to be defined to work on a number of types of data is also known as **parametric polymorphism**

Higher Order Functions

- If D is any domain, the syntax (or signature) for $Twice$ can be described as

$$Twice : (D \rightarrow D) \rightarrow D \rightarrow D$$

Given the square function, $sqr : N \rightarrow N$ where N stands for the natural numbers, it follows that

$$(Twice\ sqr) : N \rightarrow N$$

is itself a function. This new function can be named

$$define\ FourthPower \ = \ Twice\ sqr$$

Higher Order Functions

- *FourthPower* is defined without any reference to its argument
- Defining new functions in this way embodies the spirit of functional programming
- Power of a functional programming language lies in its ability to define and apply higher-order functions
  - functions that take functions as arguments and/or return a function as their result
  - *Twice* is higher-order since it maps one function to another

Source: Higher-order_functions in Multiple Languages

C++11: Functors

- **Function objects** (Functors) are objects specifically designed to be used with a syntax similar to that of functions. In C++, this is achieved by defining member function `operator()` in their class, like for example:

```
// Function Objects
struct myclass {
    int operator()(int a) { return a; }
} myobject;

int x = myobject(0); // function-like syntax with object myobject
```

- They are typically used as arguments to functions, such as predicates or comparison functions passed to standard algorithms.

Source: <functional> in STL

C++11: λ

```
#include <iostream>
#include <functional> // Provides template <class Ret, class... Args> class function<Ret(Args...)>;
using namespace std;

// lambda expressions
auto f =     [](int i) { return i + 3; };
auto twice = [](const function<int(int)>& g, int v) { return g(g(v)); };
auto sqr =   [](int i) { return i * i; };
auto comp =  [](const function<int(int)>& g, const function<int(int)>& h, int v) { return g(h(v)); };

int main() {
    auto a = 7, b = 5, c = 3; // Type inferred as int

    cout << f(a) << endl;                                   // 10
    cout << twice(f, a) << " " << comp(f, f, a) << endl;       // 13 13
    cout << twice(sqr, b) << " " << comp(sqr, sqr, b) << endl; // 625 625
    cout << comp(sqr, f, c) << " " << comp(f, sqr, c) << endl; // 36 12
}
```

Source: <functional> in STL; std::function in <functional>