CSE216 Programming Abstractions

Lambda Calculus

YoungMin Kwon

SUNY Korea
The State University of New York

Functions

- Function: a single-valued mapping
  - Associates every element of a set (domain) with at most one element in another set (range)
    - sqrt: $\mathbb{R} \rightarrow \mathbb{R}$
  - Function of more than one variable: functions whose domains are Cartesian products
    - add: $(\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R}$
  - Total function: if a function provides a mapping for every element of the domain
  - Partial function: a function that is not total

SUNY Korea

Functions as Sets

- Functions as sets
  - Subset of the Cartesian product of the domain and the range
    - sqrt $\subset \mathbb{R} \times \mathbb{R}$
    - add $\subset \mathbb{R} \times \mathbb{R} \times \mathbb{R}$
  - Specify which subset
    - sqrt $\equiv \{ (x, y) \in \mathbb{R} \times \mathbb{R} \mid y \geq 0 \text{ and } x = y^2 \}$
    - add $\equiv \{ (x, y, z) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R} \mid z = x + y \}$

SUNY Korea

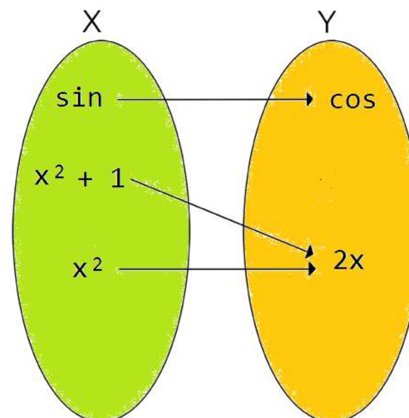
Functions as Mathematical Objects

- Function is an ordinary mathematical object
  - Function from A to B is a subset of $A \times B$
  - A function is an element of the power set of $A \times B$
  - $2^{A \times B}$: the set of all subsets of $A \times B$
    - sqrt $\in 2^{\mathbb{R} \times \mathbb{R}}$
    - add $\in 2^{\mathbb{R} \times \mathbb{R} \times \mathbb{R}}$

SUNY Korea
The State University of New York

Functions as Mathematical Objects

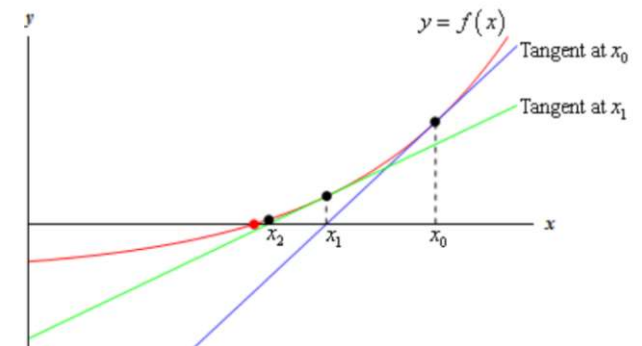
- Examples
  - High order function derivative: derivative(f) = f'
    - derivative(sin x) = cos x
    - derivative(x^3) = 3 x^2
  - derivative is a function mapping an element in $2^{A \times B}$ to an element in $2^{A \times B}$



SUNY Korea

Functions as Mathematical Objects

- Examples
  - Fixed point x of a function f: ℝ→ℝ is an x such that f(x) = x
  - Fixed point Y f of a recursive function f: f (Y f) = Y f
    - E.g. gcd = f gcd
    - gcd is the fixed point of f
    - gcd = Y f



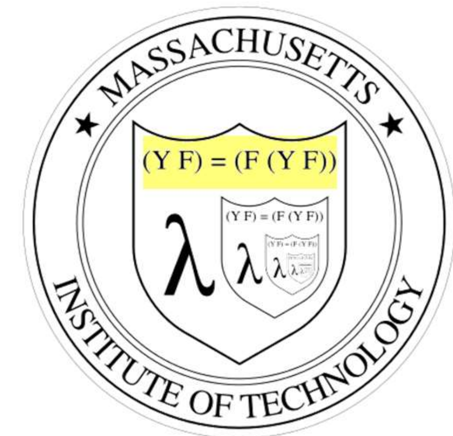
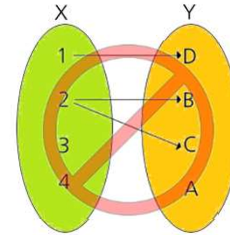


Image from Knights of Lambda Calculus

Function Space

- Function space
  - Functions are single-valued



    - They constitute only some of the elements of $2^{A \times B}$
  - Function space $A \rightarrow B$
    - Constitutes all and only those sets of $2^{A \times B}$ in which the first component of each pair is unique
    - $(A \rightarrow B) \subset 2^{A \times B}$
    - sqrt $\in (\mathbb{R} \rightarrow \mathbb{R})$
    - add $\in ((\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R})$

High Order Functions

- High order functions
  - Functions are elements of sets
    - compose $\equiv \{ (f, g, h) \mid \forall x \in \mathbb{R}, h(x) = f(g(x)) \}$
  - Domain and range of compose
    - compose $\in ((\mathbb{R} \to \mathbb{R}) \times (\mathbb{R} \to \mathbb{R})) \to (\mathbb{R} \to \mathbb{R})$

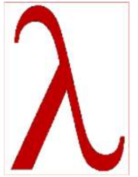
SUNY Korea
The State University of New York

High Order Functions

- Curried functions
  - Curried plus: function from reals to functions from reals to reals
    - Rather than a function from pairs of reals to reals
  - E.g.
    - add $\in (\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$, curried_add $\in \mathbb{R} \to (\mathbb{R} \to \mathbb{R})$
    - add: fun (a, b) -> a + b
    - curried_add: fun a -> (fun b -> a + b)
      $\equiv$ fun a b -> a + b

SUNY Korea

Lambda Calculus



- Function as a set
  - Does not tell how to compute the value of a function at a given point
- Lambda Calculus
  - Designed by Church
  - Addresses the limitation of function as a set
  - Lambda calculus represents everything as a function

Lambda Calculus

Alanzo Church

- Church numerals, Church Boolean, …
  - Natural numbers as lambda calculus
    - Zero and successor
  - For now, we will use ordinary arithmetic
    - Church numerals are theoretically important, but it is cumbersome in arithmetic

SUNY Korea

Lambda Expression

- Recursive definition of Lambda expression
  - A name
    - E.g. x, y, z
  - A lambda abstraction that consists of
    - The letter λ, a name, a dot, and a lambda expression
    - E.g. λ x . add x 1 (fun x -> x + 1)
    - E.g. λ x . λ y . add x y (fun x -> fun y -> x + y)
  - A function application that consists of
    - Two lambda expressions
    - E.g. (λ x . add x 1) 2 (fun x -> x + 1) 2
    - E.g. (λ x . λ y . add x y) 2 3 (fun x -> fun y -> x + y) 2 3
  - A parenthesized lambda expression
    - E.g. (λ x . add x 1)

SUNY Korea

Lambda Expression

- Function application
  - Juxtaposition of two lambda expressions
    - The first expression is a function
    - The second expression is a parameter
    - E.g. sqrt n
  - Application associates left-to-right
    - f A B is (f A) B, rather than f (A B)
  - Application has higher precedence than abstraction
    - λ x . A B is λ x . (A B), rather than (λ x . A) B

SUNY Korea

Context Free Grammar

- Context Free Grammar
  - Often used to formally define the syntax of a programming language
- Definition:
  - A set of potentially recursive production rules
  - Production rules are of the form

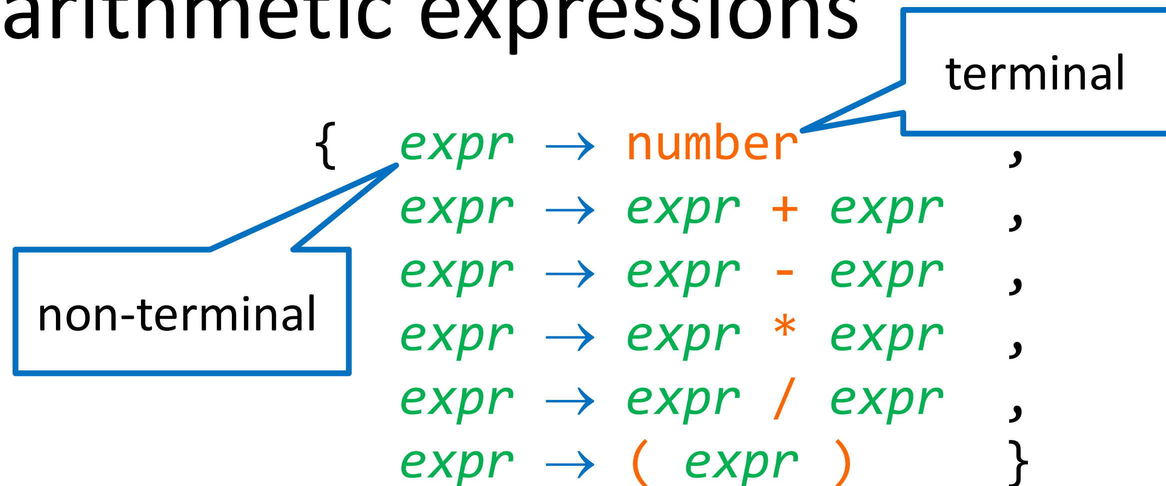
$$A \rightarrow \alpha$$

    - A: construct name (non-terminal)
    - α: a string of terminals and/or non-terminals

SUNY Korea
The State University of New York

Context Free Grammar

- E.g. arithmetic expressions



  - Alternative syntax

```
expr → number
     | expr + expr
     | expr - expr
     | expr * expr
     | expr / expr
     | ( expr )
```

SUNY Korea

Context Free Grammar

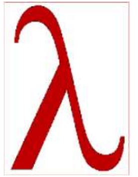
- E.g. program statements

```
statement
    → name := expr ;
    | if ( expr ) statement else statement
    | while ( expr ) statement
    | { statement_list }

statement_list
    →
    | statement_list statement
```

SUNY Korea
The State University of New York

Lambda Expression



- Syntax: CFG for lambda expressions

```
expr → name
     | number
     | λ name . expr
     | expr expr
     | ( expr )
```

Lambda Expression

- Unambiguous CFG with parentheses

```
expr → name
     | number
     | λ name . expr
     | func arg
     | ( expr )

func → name
     | ( λ name . expr )
     | func arg

arg  → name
     | number
     | ( λ name . expr )
     | ( func arg )
```

SUNY Korea

Variable Binding

- Binding parameters with λ
  - The name after the letter λ is a formal parameter

    λ x . add x x

    - a function that returns the double of its argument
  - The name after λ is said to be bound within the expression following the dot
  - The expression is the variable's scope
  - A variable that is not bound is said to be free
  - Bound variables can be a function or an argument

SUNY Korea
The State University of New York

Variable Binding

- Free variables
  - Lexical scoping,
  - A free variable needs to be defined in some surrounding scope
  - E.g.
    - x and y are bound in λ x . λ y . add x y
    - y is bound, but x is free in the inner expression λ y . add x y

SUNY Korea

Evaluation

- Evaluation rules
  - To compute with lambda calculus
  - 3 rules are sufficient for the computation
    - β-reduction
    - α-conversion
    - η-reduction

SUNY Korea
The State University of New York

Evaluation

- Beta reduction
  - $\approx$ function application
  - For any lambda abstraction $\lambda\ x\ .\ E$ and any expression M

$$(\lambda\ x\ .\ E\ )\ M \rightarrow_{\beta} E\ [\ M \setminus x\ ]$$

    - $E\ [M \setminus x]$: the expression E with all free occurrences of x replaced by M
    - E.g. $(\lambda\ x\ .\ add\ x\ x\ )\ 3 \rightarrow_{\beta} add\ 3\ 3$
  - Beta reduction is **not** permitted if any free variables in M become bound in $E\ [\ M \setminus x\ ]$
    - E.g. $(\lambda\ x\ .\ \lambda\ y\ .\ add\ x\ y)\ y$

SUNY Korea
The State University of New York

Evaluation

- Alpha conversion
  - Renaming a variable
    - Enable beta-reduction by renaming variables with fresh names
  - For any lambda abstraction $\lambda\, x \,.\, E$ and any variable y that has no free occurrences in E

$$\lambda\, x \,.\, E \rightarrow_{\alpha} \lambda\, y \,.\, E\,[\, y \setminus x \,]$$

  - E.g.

$(\, \lambda\, x \,.\, \lambda\, y \,.\, \text{add}\ x\ y \,)\ y \ \rightarrow_{\beta}\ \lambda\, y \,.\, \text{add}\ y\ y$: **incorrect!**

$$\begin{aligned}(\, \lambda\, x \,.\, \lambda\, y \,.\, \text{add}\ x\ y \,)\ y \ &\rightarrow_{\alpha} (\, \lambda\, x \,.\, \lambda\, z \,.\, \text{add}\ x\ z \,)\ y \\ &\rightarrow_{\beta}\ \lambda\, z \,.\, \text{add}\ y\ z\end{aligned}$$

SUNY Korea
The State University of New York

Evaluation

- Eta reduction
  - Eliminate surplus lambda abstractions
  - For any lambda abstraction $\lambda\, x\, .\, E$, where
    - E is of the form $F\ x$, and
    - x has no free occurrences in F

$$\lambda\, x\, .\, F\, x \rightarrow_{\eta} F$$

  - E.g.

$\lambda\, x\, .\, \text{square}\ x \rightarrow_{\eta} \text{square}$ fun x -> square x $\equiv$ square

SUNY Korea
The State University of New York

Evaluation

- Delta reduction
  - To accommodate arithmetic operations
  - An expression of the form op x y, where
    - x and y are numeric literals and
    - op is one of a small set of standard functions
  - The expression is replaced by its arithmetic value
  - E.g.

$$\text{add } 2\ 3 \rightarrow_{\delta} 5$$
$$\text{sub } 5\ 2 \rightarrow_{\delta} 3$$
$$\text{mul } 2\ 3 \rightarrow_{\delta} 6$$

SUNY Korea

Evaluation

- Reduction to simplest forms
  - Simplest form: a form in which no further β-reduction is possible
  - Through repeated application of β-reduction, α-conversion and η-reduction,
    ⇒ a lambda expression is reduced to its *simplest form*

SUNY Korea

Evaluation

- Reduction of a lambda expression

$$(\lambda\, f\, .\, \lambda\, x\, .\, \lambda\, y\, .\, f\, x\, y)\ (\lambda\, x\, .\, \lambda\, y\, .\, \text{add}\ x\, y)\ 2\ 3$$

$$\rightarrow_\alpha (\lambda\, f\, .\, \lambda\, a\, .\, \lambda\, b\, .\, f\, a\, b)\ (\lambda\, x\, .\, \lambda\, y\, .\, \text{add}\ x\, y)\ 2\ 3$$

$$\rightarrow_\beta (\lambda\, a\, .\, \lambda\, b\, .\, (\lambda\, x\, .\, \lambda\, y\, .\, \text{add}\ x\, y)\ a\, b)\ 2\ 3$$

$$\rightarrow_\beta (\lambda\, b\, .\, (\lambda\, x\, .\, \lambda\, y\, .\, \text{add}\ x\, y)\ 2\ b)\ 3$$

$$\rightarrow_\beta (\lambda\, x\, .\, \lambda\, y\, .\, \text{add}\ x\, y)\ 2\ 3$$

$$\rightarrow_\beta (\lambda\, y\, .\, \text{add}\ 2\ y)\ 3$$

$$\rightarrow_\beta \text{add}\ 2\ 3$$

$$\rightarrow_\delta 5$$

SUNY Korea
The State University of New York

Evaluation

$$
\begin{array}{lll}
 & (\lambda f.\lambda g.\lambda h.fg(h\,h))(\lambda x.\lambda y.x)h(\lambda x.x\,x) & \\
\rightarrow_\beta & (\lambda g.\lambda h.(\lambda x.\lambda y.x)g(h\,h))h(\lambda x.x\,x) & (1) \\
\rightarrow_\alpha & (\lambda g.\lambda k.(\lambda x.\lambda y.x)g(k\,k))h(\lambda x.x\,x) & (2) \\
\rightarrow_\beta & (\lambda k.(\lambda x.\lambda y.x)h(k\,k))(\lambda x.x\,x) & (3) \\
\rightarrow_\beta & (\lambda x.\lambda y.x)h((\lambda x.x\,x)\,(\lambda x.x\,x)) & (4) \\
\rightarrow_\beta & (\lambda y.h)((\lambda x.x\,x)\,(\lambda x.x\,x)) & (5) \\
\rightarrow_\beta & h & (6)
\end{array}
$$

SUNY Korea
The State University of New York

Order of Evaluation

- Choices in the reduction of lambda expression
  - In (5): (λ **y** . h) [(λ **x** . x x) (λ x . x x)]
- Normal order
  - When more than one β-reduction is possible, choose the one whose λ is left-most
  - In (5): substitute **y** with [(λ x. x x) (λ x. x x)] first
- Applicative order
  - Reduce the function and the argument first then substitute
  - In (5): substitute the argument **x** with (λ x. x x) first

Church-Rosser Theorem

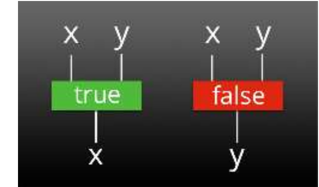
- Simplest form is unique
  - Any series of reductions that *terminates* in a non-reducible expression will produce *the same result*

- If any evaluation order terminates, normal order terminates
  - Not all reduction terminates
  - There are reductions where *normal order terminates*, but *applicative order does not*

$$
\begin{aligned}
& (\lambda x.x\,x)\,(\lambda x.x\,x) \\
\rightarrow_\beta\; & (\lambda x.x\,x)\,(\lambda x.x\,x) \\
\rightarrow_\beta\; & (\lambda x.x\,x)\,(\lambda x.x\,x) \\
\rightarrow_\beta\; & (\lambda x.x\,x)\,(\lambda x.x\,x) \\
& \ldots
\end{aligned}
$$

Conditional Expression

- Church Boolean
  - T ≡ $\lambda x . \lambda y . x$ (true: select first)
  - F ≡ $\lambda x . \lambda y . y$ (false: select second)



- Logical functions
  - and ≡ $\lambda a . \lambda b . a\ b\ F$
  - or ≡ $\lambda a . \lambda b . a\ T\ b$
  - not ≡ $\lambda a . a\ F\ T$

SUNY Korea

Conditional Expression

- If-then-else function
  - if ≡ λ c . λ t . λ e . c t e

$$
\begin{aligned}
\text{if } T\,3\,4 &\equiv (\lambda c.\lambda t.\lambda e.c\,t\,e)\,(\lambda x.\lambda y.x)\,3\,4 \\
&\rightarrow^*_\beta (\lambda x.\lambda y.x)\,3\,4 \\
&\rightarrow^*_\beta 3 \\
\text{if } F\,3\,4 &\equiv (\lambda c.\lambda t.\lambda e.c\,t\,e)\,(\lambda x.\lambda y.y)\,3\,4 \\
&\rightarrow^*_\beta (\lambda x.\lambda y.y)\,3\,4 \\
&\rightarrow^*_\beta 4
\end{aligned}
$$

SUNY Korea
The State University of New York

Recursion

- How can we define a recursive function using the lambda expression only?

```
let rec fact n =
    if   n = 0
    then 1
    else n * fact (n-1)
```

SUNY Korea
The State University of New York

Recursion: GCD Example

- First attempt

```
gcd ≡ λ a . λ b . (if (a = b)
                     a
                     (if (a > b)
                         (gcd (a - b) b)
                         (gcd (b - a) a)))
```

- Problem: gcd appears on both sides of ≡

SUNY Korea
The State University of New York

Recursion: GCD Example

- Second attempt
  - Rewrite to use beta-abstraction

```
gcd ≡ [λ g. λ a. λ b. (if (a = b)
                          a
                          (if (a > b)
                              (g (a - b) b)
                              (g (b - a) a)))] gcd
    ≡ f gcd
```

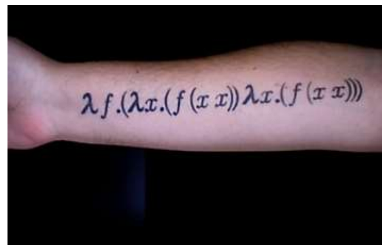
  - The equation has the form: *gcd* ≡ *f gcd*, where *f* is a non-recursive lambda expression
  - gcd is a *fixed point* of *f*

SUNY Korea

Recursion: Fixed Point Combinator

- Fixed point combinator *Y*
  - For any function *f* in a lambda expression, we can find the *least fixed point* of *f*, if there is one, by applying *Y*

$$Y \equiv \lambda h . (\lambda x . h (x x)) (\lambda x . h (x x))$$

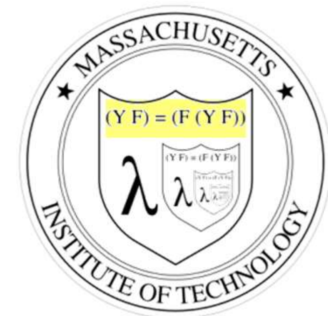


Fixed Point Combinator

- Fixed point combinator *Y*
  - For a lambda expression f,
    if the *normal-order* evaluation of Y f terminates,
    then f (Y f) and Y f will reduce to the same simplest form
  - i.e. Y f is a fixed point of f

```
Y ≡ λ h . (λ x . h (x x)) (λ x . h (x x))

Y f -> (λ x . f (x x)) (λ x . f (x x))
    ≡  (k k), where k ≡ λ x . f (x x)
    -> f (k k)
    ≡ f (Y f)      ∴ Y f  is a fixed point of f
```



SUNY Korea

Recursion: Factorial Example

- Example: factorial

```
fact' ≡ (λ f . λ n .
              if (n = 0)
                  1
                  (n * f (n - 1)))

fact
  ≡  Y fact'
  -> (k k), where k ≡ λ x . fact' (x x)
  -> fact' (k k)
  -> (λ f . λ n . if (n = 0)  1  (n * f      (n - 1))  ) (k k)
  ->          λ n . if (n = 0)  1  (n * (k k) (n - 1))  )
```

SUNY Korea

Recursion: GCD Example

- With the fixed point combinator Y, GCD can be defined as

```
gcd' ≡ (λ g. λ a. λ b. (if (a = b)
                          a
                          (if (a > b)
                              (g (a - b) b)
                              (g (b - a) a)))))

gcd ≡ Y gcd'
    -> (k k) where k ≡ λ x. gcd' (x x)
    -> gcd' (k k)
    -> λ a. λ b. (if (a = b)
                     a
                     (if (a > b)
                         ((k k) (a - b) b)
                         ((k k) (b - a) a))
```

SUNY Korea

Traces of gcd 2 4

$$
\begin{aligned}
\text{gcd } 2\,4 &\equiv \mathbf{Y} f\ 2\,4 \\
&\equiv ((\lambda h.(\lambda x.h(x\,x))\ (\lambda x.h(x\,x)))f)\ 2\,4 \\
&\rightarrow_\beta ((\lambda x.f\,(x\,x))\ (\lambda x.f\,(x\,x)))\ 2\,4 \\
&\equiv (k\,k)\ 2\,4, \quad \text{where } k \equiv \lambda x.f\,(x\,x) \\
&\rightarrow_\beta (f\,(k\,k))\ 2\,4 \\
&\equiv ((\lambda g.\lambda a.\lambda b.(\text{if } (=a\,b)\ a\ (\text{if } (>a\,b)\ (g(-\,a\,b)\ b)\ (g(-\,b\,a)\ a))))\ (k\,k))\ 2\,4 \\
&\rightarrow_\beta (\lambda a.\lambda b.(\text{if } (=a\,b)\ a\ (\text{if } (>a\,b)\ ((k\,k)(-\,a\,b)\ b)\ ((k\,k)(-\,b\,a)\ a))))\ 2\,4 \\
&\rightarrow_\beta^* \text{if } (=2\,4)\ 2\ (\text{if } (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\equiv (\lambda c.\lambda t.\lambda e.c\,t\,e)\ (=2\,4)\ 2\ (\text{if } (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\rightarrow_\beta^* (=2\,4)\ 2\ (\text{if } (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\rightarrow_\delta F\ 2\ (\text{if } (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\equiv (\lambda x.\lambda y.y)\ 2\ (\text{if } (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\rightarrow_\beta^* \text{if } (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2) \\
&\rightarrow \ldots
\end{aligned}
$$

SUNY Korea
The State University of New York

Traces of gcd 2 4 (continued)

$$
\begin{array}{ll}
\rightarrow & (k\,k)\,(-\,4\,2)\,2 \\
\equiv & ((\lambda x.f\,(x\,x))k)\,(-\,4\,2)\,2 \\
\rightarrow_\beta & (f\,(k\,k))\,(-\,4\,2)\,2 \\
\equiv & ((\lambda g.\lambda a.\lambda b.(\text{if }(=a\,b)\,a\,(\text{if }(>a\,b)\,(g(-\,a\,b)\,b)\,(g(-\,b\,a)\,a))))\,(k\,k))\,(-\,4\,2)\,2 \\
\rightarrow_\beta & (\lambda a.\lambda b.(\text{if }(=a\,b)\,a\,(\text{if }(>a\,b)\,((k\,k)(-\,a\,b)\,b)\,((k\,k)(-\,b\,a)\,a))))\,(-\,4\,2)\,2 \\
\rightarrow_\beta^* & \text{if }(=(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if }(>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
\equiv & (\lambda c.\lambda t.\lambda e.c\,t\,e) \\
 & (=(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if }(>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
\rightarrow_\beta^* & (=(-\,4\,2)\,2)\,(-\,4\,2)\,(\text{if }(>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
\rightarrow_\delta & (=2\,2)\,(-\,4\,2)\,(\text{if }(>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
\rightarrow_\delta & T\,(-\,4\,2)\,(\text{if }(>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
\equiv & (\lambda x.\lambda y.x)\,(-\,4\,2)\,(\text{if }(>(-\,4\,2)\,2)\,((k\,k)\,(-\,(-\,4\,2)\,2)\,2)\,((k\,k)\,(-\,2\,(-\,4\,2))\,(-\,4\,2))) \\
\rightarrow_\beta^* & (-\,4\,2) \\
\rightarrow_\delta & 2
\end{array}
$$

SUNY Korea
The State University of New York

Lists

- List processing functions
  - cons ≡ λ first . λ second . λ sel . sel first second
  - car ≡ λ c . c T (T: select first)
  - cdr ≡ λ c . c F (F: select second)
  - nil ≡ λ x . T
  - is_nil ≡ λ c . c (λ x . λ y . F)
    - If c is nil, (λ x . T) (λ x . λ y . F) will return T
    - If not, (λ x . λ y . F) first second will return F

SUNY Korea
The State University of New York

Lists

- List operator identities

$$
\begin{aligned}
\mathsf{car}(\mathsf{cons}\ A\ B) &\equiv (\lambda l.l\ \mathsf{select_first})\ (\mathsf{cons}\ A\ B) \\
&\rightarrow_\beta (\mathsf{cons}\ A\ B)\ \mathsf{select_first} \\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \mathsf{select_first} \\
&\rightarrow_\beta^* (\lambda x.x\ A\ B)\ \mathsf{select_first} \\
&\rightarrow_\beta \mathsf{select_first}\ A\ B \\
&\equiv (\lambda x.\lambda y.x)\ A\ B \\
&\rightarrow_\beta^* A
\end{aligned}
$$

SUNY Korea
The State University of New York

Lists

- List operator identities

$$
\begin{aligned}
\mathsf{cdr}(\mathsf{cons}\ A\ B) &\equiv (\lambda l.l\ \mathsf{select_second})\ (\mathsf{cons}\ A\ B) \\
&\rightarrow_\beta (\mathsf{cons}\ A\ B)\ \mathsf{select_second} \\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \mathsf{select_second} \\
&\rightarrow_\beta^* (\lambda x.x\ A\ B)\ \mathsf{select_second} \\
&\rightarrow_\beta \mathsf{select_second}\ A\ B \\
&\equiv (\lambda x.\lambda y.y)\ A\ B \\
&\rightarrow_\beta^* B
\end{aligned}
$$

SUNY Korea
The State University of New York

Lists

- List operator identities

$$
\begin{aligned}
\text{null? (cons } A\ B) &\equiv (\lambda l.l\,(\lambda x.\lambda y.\text{select_second}))\,(\text{cons } A\ B)\\
&\rightarrow_\beta (\text{cons } A\ B)\,(\lambda x.\lambda y.\text{select_second})\\
&\equiv ((\lambda a.\lambda d.\lambda x.x\,a\,d)\,A\ B)\,(\lambda x.\lambda y.\text{select_second})\\
&\rightarrow_\beta^* (\lambda x.x\,A\ B)\,(\lambda x.\lambda y.\text{select_second})\\
&\rightarrow_\beta (\lambda x.\lambda y.\text{select_second})\,A\ B\\
&\rightarrow_\beta^* \text{select_second}\\
&\equiv F
\end{aligned}
$$

null? means is_nil

SUNY Korea

Numbers

- Church numerals
  - n: a function that takes f and x and apply f to x n times
  - zero $\equiv \lambda f . \lambda x . x$
    - Apply f 0 times to x
  - succ $\equiv \lambda n . \lambda f . \lambda x . f (n f x)$
    - Successor: apply f 1 more time to n
  - is_zero $\equiv \lambda n . n (\lambda x . F) T$
    - If $(\lambda x . F)$ is applied 0 times to T, it will return T otherwise it will return F

SUNY Korea

Numbers

- Arithmetic operators
  - add $\equiv \lambda\, m\, .\, \lambda\, n\, .\, \lambda\, f\, .\, \lambda\, x\, .\, m\, f\, (\, n\, f\, x\,)$
  - mul $\equiv \lambda\, m\, .\, \lambda\, n\, .\, \lambda\, f\, .\, \lambda\, x\, .\, m\, (\, n\, f\,)\, x$

  - sub $\equiv \lambda\, m\, .\, \lambda\, n\, .\, n\ \text{pred}\ m$ (* m – n *)
  - pred $\equiv \lambda\, n\, .\, \text{car}\ (\, n\ \text{shift}\ (\, \text{cons zero zero}\,)\,)$ (* m – 1 *)
  - shift $\equiv \lambda\, \text{pair}\, .\, \text{cons}\ (\, \text{cdr pair}\,)\ (\, \text{succ}\ (\, \text{cdr pair}\,)\,)$
    - apply shift 3 times to (0, 0): (0, 0) $\rightarrow$ (0, 1) $\rightarrow$ (1, 2) $\rightarrow$ (2, 3)
    - pred 3: car (2, 3) $\rightarrow$ 2

SUNY Korea

Numbers

- Example (add 2 3)
  - add ≡ λ m . λ n . λ f . λ x . m f (n f x)

```
add (λg.λy.gg y) (λh. λz.hhh z)
→ λf. λx. (λg.λy.gg y) f [ (λh.λz.hhh z) f x]
→ λf. λx. (λg.λy.gg y) f [ (λz.fff z) x]
→ λf. λx. (λg.λy.gg y) f (fff x)
→ λf. λx. (λy.ff y) (fff x)
→ λf. λx. ff fff x
```

SUNY Korea
The State University of New York

Numbers

- Example (mul 2 3)
  - mul ≡ λ m . λ n . λ f . λ x . m (n f) x

```
mul (λg.λy.gg y) (λh. λz.hhh z)
→ λf. λx. (λg.λy.gg y) [ (λh.λz.hhh z) f ] x
→ λf. λx. (λg.λy.gg y) (λz.fff z) x
→ λf. λx. λy.{ (λz.fff z) [(λz.fff z) y] } x
→ λf. λx. λy.{ (λz.fff z) (fff y) } x
→ λf. λx. λy.{ fff fff y} x
→ λf. λx. fff fff x
```

SUNY Korea The State University of New York

Numbers

- Example (pred 3)
  - pred ≡ λ n . car (n shift (cons zero zero))

```
pred (λf. λx. fff x)
→  car { (λf. λx. fff x) shift (cons λf. λx.x λf.λx.x) }
→  car { (λx. shift shift shift x) (cons λf.λx.x λf.λx.x)) }
→  car { shift shift shift (cons λf. λx.x  λf.λx.x)   }
→  car { shift shift       (cons λf. λx.x  λf.λx.fx)  }
→  car { shift             (cons λf. λx.fx λf.λx.ffx) }
→  car (cons λf. λx.ffx λf.λx.fffx)
→  λf.λx.ffx
```

SUNY Korea
The State University of New York

Numbers

- Example (sub 3 2)
  - sub ≡ λ m . λ n . n pred m

```
sub (λf.λy.fff y) (λg. λz.gg z)
→ (λg. λz.gg z) pred (λf.λx.fff x)
→ (λz.pred pred z) (λf.λx.fff x)
→ pred (pred (λf.λx.fff x))
→ pred (λf.λx.ff x)
→ λf.λx.f x
```

SUNY Korea
The State University of New York

Assignment 7

- Church encoding
  - Download tiny_eval.zip and make it a lazy evaluator
  - Download church.ml
  - Implement the TODOs
  - Submit globals.ml, eval.ml, and church.ml in a single zip file
- Due date: 5/9/2024

SUNY Korea

```
(*--Helpers----------------------*)
(*eval_str: evaluates string expression in env*)
let eval_str str_expr env =
    parse str_expr
    |> fun expr -> eval expr env

(*define: extends env with the value of string expression
*)
let define name str_expr env =
    eval_str str_expr env
    |> fun v -> (name, v)::env



(*--Boolean literals----------------------*)
(*TODO: implement TRUE and FALSE.
    you can use lambda.
*)
let str_true = "(lambda (x y) x)"


let str_false =
```

SUNY Korea

```
let env = []
        |> define "TRUE"  str_true
        |> define "FALSE" str_false

let _ = Printf.printf "--Test Boolean----------\n";

eval_str "(TRUE  true false)" env |> fun x ->
    print x;
    assert (x = BOOL true);

eval_str "(FALSE true false)" env |> fun x ->
    print x;
    assert (x = BOOL false)

(* expected results
BOOL(true)
BOOL(false)
*)

(*-- move this line to the next TODO when done ----------
```

SUNY Korea

```
(*--Logical operators----------------------*)
(*TODO: implement AND, OR, NOT.
    you can use lambda, TRUE, FALSE
*)
let str_and =


let str_or =


let str_not =


let env = env
        |> define "AND" str_and
        |> define "OR"  str_or
        |> define "NOT" str_not
```

SUNY Korea
The State University of New York

```
(*--List operators-----------------------*)
(*TODO: implement CONS, CAR, CDR, NIL, IS_NIL
    you can use lambda, TRUE, FALSE
*)
let str_cons =


let str_car =


let str_cdr =


let str_nil =


let str_is_nil =


let env = env
        |> define "CONS"   str_cons
        |> define "CAR"    str_car
        |> define "CDR"    str_cdr
        |> define "NIL"    str_nil
        |> define "IS_NIL" str_is_nil
```

SUNY Korea

```
(*--Numbers-----------------------*)
(*TODO: implement ZERO, SUCC (successor), IS_ZERO
    you can use lambda, TRUE, FALSE
*)
let str_zero =


let str_succ =


let str_is_zero =


let env = env
        |> define "ZERO"    str_zero
        |> define "SUCC"    str_succ
        |> define "IS_ZERO" str_is_zero
```

SUNY Korea
The State University of New York

```
(*for testing*)
let env = env
          |> define "ONE"    "(SUCC ZERO)"
          |> define "TWO"    "(SUCC ONE)"
          |> define "THREE"  "(SUCC TWO)"
          |> define "FOUR"   "(SUCC THREE)"
          |> define "FIVE"   "(SUCC FOUR)"
          |> define "SIX"    "(SUCC FIVE)"
          |> define "SEVEN"  "(SUCC SIX)"
          |> define "EIGHT"  "(SUCC SEVEN)"
          |> define "NINE"   "(SUCC EIGHT)"
          |> define "TEN"    "(SUCC NINE)"

let env = env
          |> define "INC"    "(lambda (x) (+ x 1))"
```

SUNY Korea
The State University of New York

```
(*--Arithmetic operators----------------------*)
(*TODO: implement ADD, MUL and SUB.
    you can use lambda, PRED
*)
let str_add =


let str_mul =


let str_sub =


let str_pred =
    "(  (lambda (shift n)
            (CAR (n shift (CONS ZERO ZERO)))
        (lambda (pair)
            (CONS (CDR pair) (SUCC (CDR pair)))))"

let env = env
        |> define "ADD"  str_add
        |> define "MUL"  str_mul
        |> define "PRED" str_pred
        |> define "SUB"  str_sub
```

SUNY Korea
The State University of New York

```
(*--Comparison operators----------------------*)
let str_geq =
    "(lambda (m n)
        (IS_ZERO (SUB n m)))"

let str_equ =
    "(lambda (m n)
        (AND (GEQ m n) (GEQ n m)))"

let env = env
        |> define "GEQ" str_geq
        |> define "EQU" str_equ
```

SUNY Korea
The State University of New York

```
(*--Conditional operator----------------------*)
(*TODO: implement IF.
    you can use lambda
*)
let str_if =


let env = env
        |> define "IF" str_if



(*--Y-combinator----------------------*)
(*TODO: implement Y.
    you can use lambda
*)
let str_y =


let env = env
        |> define "Y" str_y
```

SUNY Korea
The State University of New York

```
(*max function*)
(*TODO: implement max function
    you can use lambda, IF, GEQ
*)
let str_max =


let _ = Printf.printf "--Test max----------\n";
define "max" str_max env
    |> eval_str "((max TWO TEN) INC 0)"
    |> fun x -> print x; x
    |> fun x -> assert (x = NUM 10)
(* expected results
NUM(10)
*)
```

SUNY Korea
The State University of New York

```
(*sum function*)
(*TODO: implement sum function
    you can use lambda, Y, IF, EQU, GEQ, ZERO, ONE, ADD, SUB
*)
let str_sum =


let _ = Printf.printf "--Test sum----------\n";
define "sum" str_sum env
    |> eval_str "((sum TEN) INC 0)"
    |> fun x -> print x; x
    |> fun x -> assert (x = NUM 55)
(* expected results
NUM(55)
*)
```

SUNY Korea
The State University of New York

```
(*gcd function*)
(*TODO: implement gcd function
    you can use lambda, Y, IF, EQU, GEQ, ZERO, ONE, ADD, SUB
*)
let str_gcd =


let _ = Printf.printf "--Test gcd----------\n";
define "gcd" str_gcd env
    |> eval_str "((gcd EIGHT SIX) INC 0)"
    |> fun x -> print x; x
    |> fun x -> assert (x = NUM 2)
(* expected results
NUM(2)
*)
```

SUNY Korea
The State University of New York

```
(*index and a stream of natural numbers*)
(*TODO: implement index function and nat, a stream of natural numbers
    you can use lambda, Y, IF, EQU, GEQ, ZERO, ONE, ADD, SUB,
    CONS, CAR, CDR
*)
let str_index =


let str_nat =



let _ = Printf.printf "--Test nat----------\n";
env |> define "index" str_index
    |> define "nat"   str_nat
    |> eval_str "((index nat TEN) INC 0)"
    |> fun x -> print x; x
    |> fun x -> assert (x = NUM 10)
(* expected results
NUM(10)
*)
```

SUNY Korea