# Currying and the Lambda Calculus

CS 350

---

Dr. Joseph Eremondi

Last updated: July 30, 2024

# Overview

# Objectives

- Learning Goals

# Objectives

- Learning Goals
  - To learn how multi-argument functions can be desugared into single-argument functions

# Objectives

- Learning Goals
  - To learn how multi-argument functions can be desugared into single-argument functions
    - Curly-Curry

# Objectives

- Learning Goals
  - To learn how multi-argument functions can be desugared into single-argument functions
    - Curly-Curry
  - To see that *everything* can be desugared into single-argument functions

# Objectives

- Learning Goals
  - To learn how multi-argument functions can be desugared into single-argument functions
    - Curly-Curry
  - To see that *everything* can be desugared into single-argument functions
    - by learning about the Lambda Calculus

# Objectives

- Learning Goals
  - To learn how multi-argument functions can be desugared into single-argument functions
    - Curly-Curry
  - To see that *everything* can be desugared into single-argument functions
    - by learning about the Lambda Calculus
- Core Concepts

# Objectives

- Learning Goals
  - To learn how multi-argument functions can be desugared into single-argument functions
    - Curly-Curry
  - To see that *everything* can be desugared into single-argument functions
    - by learning about the Lambda Calculus
- Core Concepts
  - Currying

# Objectives

- Learning Goals
    - To learn how multi-argument functions can be desugared into single-argument functions
        - Curly-Curry
    - To see that *everything* can be desugared into single-argument functions
        - by learning about the Lambda Calculus
- Core Concepts
    - Currying
    - Lambda Calculus

# Currying

# Executing a multiple-argument function

- Say we allow `{lambda {x y z} {+ x {* y z}}}` in Curly

## Executing a multiple-argument function

- Say we allow `{lambda {x y z} {+ x {* y z}}}` in Curly
- How can we interpret a call to this function?

# Executing a multiple-argument function

- Say we allow `{lambda {x y z} {+ x {* y z}}}` in Curly
- How can we interpret a call to this function?
    - Evaluate the body with either

# Executing a multiple-argument function

- Say we allow `{lambda {x y z} {+ x {* y z}}}` in Curly
- How can we interpret a call to this function?
  - Evaluate the body with either
    - `x,y,z` replaced by concrete argument values (substitution)

# Executing a multiple-argument function

- Say we allow `{lambda {x y z} {+ x {* y z}}}` in Curly
- How can we interpret a call to this function?
    - Evaluate the body with either
        - `x,y,z` replaced by concrete argument values (substitution)
        - `x,y,z` bound to concrete values in an environment

# Achieving this: Currying

- We can achieve this with nested lambda expressions

# Achieving this: Currying

- We can achieve this with nested lambda expressions

# Achieving this: Currying

- We can achieve this with nested lambda expressions

```
{let f {fun {x} {fun {y} {fun {z} {+ x {* y z}}}}}
  ....}
```

- To call, we do nested calls

# Achieving this: Currying

- We can achieve this with nested lambda expressions

```
{let f {fun {x} {fun {y} {fun {z} {+ x {* y z}}}}}
  ....}
```

- To call, we do nested calls

## Achieving this: Currying

- We can achieve this with nested lambda expressions

```
{let f {fun {x} {fun {y} {fun {z} {+ x {* y z}}}}}
  ....}
```

- To call, we do nested calls

```
{{{f 1} 2} 3}
```

- Step by step:

# Achieving this: Currying

- We can achieve this with nested lambda expressions

```
{let f {fun {x} {fun {y} {fun {z} {+ x {* y z}}}}}
  ....}
```

- To call, we do nested calls

```
{{{f 1} 2} 3}
```

- Step by step:
  - `f 1` produces `{fun {y} {fun {z} {+ 1 {* y z}}}}`

# Achieving this: Currying

- We can achieve this with nested lambda expressions

```
{let f {fun {x} {fun {y} {fun {z} {+ x {* y z}}}}}
  ....}
```

- To call, we do nested calls

```
{{{f 1} 2} 3}
```

- Step by step:
    - `f 1` produces `{fun {y} {fun {z} {+ 1 {* y z}}}}`
    - Calling that on 2 produces `{fun {z} {+ 1 {* 2 z}}}`

# Achieving this: Currying

- We can achieve this with nested lambda expressions

```
{let f {fun {x} {fun {y} {fun {z} {+ x {* y z}}}}}
  ....}
```

- To call, we do nested calls

```
{{{f 1} 2} 3}
```

- Step by step:
    - `f 1` produces `{fun {y} {fun {z} {+ 1 {* y z}}}}`
    - Calling that on 2 produces `{fun {z} {+ 1 {* 2 z}}}`
    - Calling that on 3 produces `{+ 1 {* 2 3}}}`

## Achieving this: Currying

- We can achieve this with nested lambda expressions

```
{let f {fun {x} {fun {y} {fun {z} {+ x {* y z}}}}}
  ....}
```

- To call, we do nested calls

```
{{{f 1} 2} 3}
```

- Step by step:
  - `f 1` produces `{fun {y} {fun {z} {+ 1 {* y z}}}}`
  - Calling that on 2 produces `{fun {z} {+ 1 {* 2 z}}}`
  - Calling that on 3 produces `{+ 1 {* 2 3}}}`
    - Exactly what we want for `{f 1 2 3}`

# Definition

- The approach of simulating multiple-argument functions with nested single-argument functions is called **Currying**

# Definition

- The approach of simulating multiple-argument functions with nested single-argument functions is called **Currying**
  - Named after Haskell Curry, and American logician

# Definition

- The approach of simulating multiple-argument functions with nested single-argument functions is called **Currying**
    - Named after Haskell Curry, and American logician
        - Also the namesake of the Haskell programming language

# Definition

- The approach of simulating multiple-argument functions with nested single-argument functions is called **Currying**
    - Named after Haskell Curry, and American logician
        - Also the namesake of the Haskell programming language
- A function written in this style is *curried*

# Desugaring with Currying

- We can add multiple-argument functions to our Surface Language *without* changing the core language

## Desugaring with Currying

- We can add multiple-argument functions to our Surface Language *without* changing the core language
  - Handle in `elab`

# Desugaring with Currying

- We can add multiple-argument functions to our Surface Language *without* changing the core language
  - Handle in `elab`
  - Doesn't matter if do substitution or environment-based version, translation is the exact same

# AST for Multi-Argument Functions

# AST for Multi-Argument Functions

```
(define-type SurfExpr
....
 (SurfFun [xs : (Listof Symbol)]
          [body : SurfExpr]))
 (SurfCall [fun : SurfExpr]
           [args : (Listof SurfExpr)])
```

- Functions now take a list of variables

# AST for Multi-Argument Functions

```
(define-type SurfExpr
....
 (SurfFun [xs : (Listof Symbol)]
          [body : SurfExpr]))
 (SurfCall [fun : SurfExpr]
           [args : (Listof SurfExpr)])
```

- Functions now take a list of variables
- Calls now take a list of arguments

# Elaborating Multi-Argument Functions

- Use recursion to iterate through the list

# Elaborating Multi-Argument Functions

- Use recursion to iterate through the list

# Elaborating Multi-Argument Functions

- Use recursion to iterate through the list

```
(define (elab [surfExpr : SurfExpr])
  (type-case SurfExpr surfExpr
             ....
   [(SurfFun xs body)
     (type-case (Listof Symbol) xs
       [empty
         (elab body)]
       [(cons x rest)
         ;; Could also do with a helper fn
         (Fun x (elab (SurfFun rest body)))])]))
```

- No arguments: just produce the body

# Elaborating Multi-Argument Functions

- Use recursion to iterate through the list

```
(define (elab [surfExpr : SurfExpr])
  (type-case SurfExpr surfExpr
              ....
   [(SurfFun xs body)
     (type-case (Listof Symbol) xs
       [empty
         (elab body)]
       [(cons x rest)
         ;; Could also do with a helper fn
         (Fun x (elab (SurfFun rest body)))])]))
```

- No arguments: just produce the body
- At least one argument: curry the rest of the arguments, and wrap the result in a lambda

# Multi-Argument Calls with Tail Recursion

# Multi-Argument Calls with Tail Recursion

```
(define (callHelper [args : (Listof SurfExpr)]
                    [accum : Expr]) : Expr
  (type-case (Listof SurfExpr) args
      [empty accum]
      [(cons arg rest)
         ;; tail recursion
         (callHelper rest (Call accum (elab arg)) )]))
```

- Given a list of argument to apply, build up one giant expression with nested calls

# Building The Entire Call

# Building The Entire Call

```
(define (elab [surfExpr : SurfExpr])
  (type-case SurfExpr surfExpr
             ....
   [(SurfCall funExpr args)
      (callHelper args (elab funExpr))]))
```

## A Word of Warning

- Plait does *not* use currying by default

## A Word of Warning

- Plait does *not* use currying by default
    - Multiple argument functions are *not* desugared into single-argument ones

# A Word of Warning

- Plait does *not* use currying by default
  - Multiple argument functions are *not* desugared into single-argument ones
- We can convert between curried and uncurried functions with combinators

# A Word of Warning

- Plait does *not* use currying by default
  - Multiple argument functions are *not* desugared into single-argument ones
- We can convert between curried and uncurried functions with combinators
  - See lecture on Higher Order Functions

## A Word of Warning

- Plait does *not* use currying by default
    - Multiple argument functions are *not* desugared into single-argument ones
- We can convert between curried and uncurried functions with combinators
    - See lecture on Higher Order Functions
- A `0`-argument `(lambda () e)` is *NOT* the same as `e` in Plait

# A Word of Warning

- Plait does *not* use currying by default
  - Multiple argument functions are *not* desugared into single-argument ones
- We can convert between curried and uncurried functions with combinators
  - See lecture on Higher Order Functions
- A `0`-argument `(lambda () e)` is *NOT* the same as `e` in Plait
  - But it is in Curly, if we use this desugaring

# Another Desugaring: Let

- Recall that `letvar` let us define a local variable to have the value of some expression, that we could then use to build another expression

# Another Desugaring: Let

- Recall that `letvar` let us define a local variable to have the value of some expression, that we could then use to build another expression
- We can make `letvar` syntactic sugar using lambda

# Another Desugaring: Let

- Recall that `letvar` let us define a local variable to have the value of some expression, that we could then use to build another expression
- We can make `letvar` syntactic sugar using lambda

# Another Desugaring: Let

- Recall that `letvar` let us define a local variable to have the value of some expression, that we could then use to build another expression
- We can make `letvar` syntactic sugar using lambda

```
(define (elab [surfExpr : SurfExpr])
  (type-case SurfExpr surfExpr
             ....
   [(SurfLetVar x xExpr body)
      (Call (Fun x (elab body))
            (elab xExpr))]))
```

- Define a function whose body is the body of the `letvar`

# Another Desugaring: Let

- Recall that `letvar` let us define a local variable to have the value of some expression, that we could then use to build another expression
- We can make `letvar` syntactic sugar using lambda

```
(define (elab [surfExpr : SurfExpr])
  (type-case SurfExpr surfExpr
             ....
   [(SurfLetVar x xExpr body)
      (Call (Fun x (elab body))
            (elab xExpr))]))
```

- Define a function whose body is the body of the `letvar`
- Immediately call it with the value we're giving to the defined variable

# Another Desugaring: Let

- Recall that `letvar` let us define a local variable to have the value of some expression, that we could then use to build another expression
- We can make `letvar` syntactic sugar using lambda

```
(define (elab [surfExpr : SurfExpr])
  (type-case SurfExpr surfExpr
             ....
   [(SurfLetVar x xExpr body)
      (Call (Fun x (elab body))
            (elab xExpr))]))
```

- Define a function whose body is the body of the `letvar`
- Immediately call it with the value we're giving to the defined variable
- Letvar: executes the body in the environment extended with the variable's value

## Another Desugaring: Let

- Recall that `letvar` let us define a local variable to have the value of some expression, that we could then use to build another expression
- We can make `letvar` syntactic sugar using lambda

```
(define (elab [surfExpr : SurfExpr])
  (type-case SurfExpr surfExpr
             ....
   [(SurfLetVar x xExpr body)
      (Call (Fun x (elab body))
            (elab xExpr))]))
```

- Define a function whose body is the body of the `letvar`
- Immediately call it with the value we're giving to the defined variable
- Letvar: executes the body in the environment extended with the variable's value
  - This does the exact same thing

# Lambda The Ultimate

# Functions As Sugar

- So far we've seen that single argument functions can simulate

# Functions As Sugar

- So far we've seen that single argument functions can simulate
  - Multi-argument functions

# Functions As Sugar

- So far we’ve seen that single argument functions can simulate
  - Multi-argument functions
  - Local variable definitions

# Functions As Sugar

- So far we've seen that single argument functions can simulate
    - Multi-argument functions
    - Local variable definitions
- What else can se simulate?

# Functions As Sugar

- So far we've seen that single argument functions can simulate
  - Multi-argument functions
  - Local variable definitions
- What else can se simulate?
- **NOTE** I won't ask about the following desugarings on an exam

# Functions As Sugar

- So far we've seen that single argument functions can simulate
    - Multi-argument functions
    - Local variable definitions
- What else can se simulate?
- **NOTE** I won't ask about the following desugarings on an exam
    - But they're an important introduction to the "science" of computer science and the "mathematics" of informatics

# The Smallest Language We Can Imagine

# The Smallest Language We Can Imagine

```
(define-type UTLC
  ;; A variable
  (Var [x : Symbol])
  ;; Function application (call)
  (App [fun : UTLC]
       [arg : UTLC])
  ;; Anonymous function (lambda)
  (Lam [param : Symbol]
       [body : UTCL]))
```

- Stands for “Un-Typed Lambda Calculus”

# The Smallest Language We Can Imagine

```
(define-type UTLC
  ;; A variable
  (Var [x : Symbol])
  ;; Function application (call)
  (App [fun : UTLC]
       [arg : UTLC])
  ;; Anonymous function (lambda)
  (Lam [param : Symbol]
       [body : UTCL]))
```

- Stands for “Un-Typed Lambda Calculus”
- All you can do is define an anonymous function (lambda) or call a function (application)

# The Smallest Language We Can Imagine

```
(define-type UTLC
  ;; A variable
  (Var [x : Symbol])
  ;; Function application (call)
  (App [fun : UTLC]
       [arg : UTLC])
  ;; Anonymous function (lambda)
  (Lam [param : Symbol]
       [body : UTCL]))
```

- Stands for “Un-Typed Lambda Calculus”
- All you can do is define an anonymous function (lambda) or call a function (application)
- **The Untyped Lambda Calculus is Turing Complete**

# The Smallest Language We Can Imagine

```
(define-type UTLC
  ;; A variable
  (Var [x : Symbol])
  ;; Function application (call)
  (App [fun : UTLC]
       [arg : UTLC])
  ;; Anonymous function (lambda)
  (Lam [param : Symbol]
       [body : UTCL]))
```

- Stands for “Un-Typed Lambda Calculus”
- All you can do is define an anonymous function (lambda) or call a function (application)
- **The Untyped Lambda Calculus is Turing Complete**
    - Any program you can write, you can write an equivalent UTLC Program

# Interpreting the UTLC

- Works just like we've seen so far

# Interpreting the UTLC

- Works just like we've seen so far
  - `Lam` is like `Fun`

## Interpreting the UTLC

- Works just like we've seen so far
    - `Lam` is like `Fun`
    - `App` is like `Call`

# Interpreting the UTLC

- Works just like we've seen so far
  - `Lam` is like `Fun`
  - `App` is like `Call`
- Substitution version

## Interpreting the UTLC

- Works just like we've seen so far
  - `Lam` is like `Fun`
  - `App` is like `Call`
- Substitution version

# Interpreting the UTLC

- Works just like we've seen so far
  - `Lam` is like `Fun`
  - `App` is like `Call`
- Substitution version

```
(define (interpUTLC [e : UTLC] : UTLC)
  (type-case UTLC e
    [(App fun arg)
       (type-case (interpUTLC fun)
          [(Lam x body)
             (interpUTLC (subst x arg body))]
          [else (error 'x "Undefined variable")])]
    ;; Function and variables don't do any computation,
    ;; they just return themselves
    ;; Could have variables return an error,
    [else e]))
```

- Can also do with environments, like Curly-Lambda-Env

# Booleans

- Note: I'll write the desugarings as functions, rather than with `elab`, just to keep things self contained

# Booleans

- Note: I'll write the desugarings as functions, rather than with `elab`, just to keep things self contained
- You can imagine these functions as constructors we'd use for the surface syntax, that produce `UTLC` directly

# Booleans

- Note: I'll write the desugarings as functions, rather than with `elab`, just to keep things self contained
- You can imagine these functions as constructors we'd use for the surface syntax, that produce UTLC directly

# Booleans

- Note: I'll write the desugarings as functions, rather than with `elab`, just to keep things self contained
- You can imagine these functions as constructors we'd use for the surface syntax, that produce UTLC directly

```
(define True
  (Lam 'x (Lam 'y (Var 'x))))
(define False
  (Lam 'x (Lam 'y (Var 'y))))
;; curried (test thenCase elseCase)
(define (If test thenCase elseCase)
  (App (App test thenCase) elseCase))
```

- If we give `If` the boolean `True` it produces the then-case

# Booleans

- Note: I'll write the desugarings as functions, rather than with `elab`, just to keep things self contained
- You can imagine these functions as constructors we'd use for the surface syntax, that produce UTLC directly

```
(define True
  (Lam 'x (Lam 'y (Var 'x))))
(define False
  (Lam 'x (Lam 'y (Var 'y))))
;; curried (test thenCase elseCase)
(define (If test thenCase elseCase)
  (App (App test thenCase) elseCase))
```

- If we give `If` the boolean `True` it produces the then-case
- If we give `If` the boolean `False` it produces the else-case

# Numbers

- Define $n$ to be the function that takes in another function and an argument, and applies it $n$ times

# Numbers

- Define $n$ to be the function that takes in another function and an argument, and applies it $n$ times
  - $\lambda f \lambda x. f(f(f...(fx)))$

# Numbers

- Define $n$ to be the function that takes in another function and an argument, and applies it $n$ times
  - $\lambda f \lambda x. f(f(f ...(fx)))$

# Numbers

- Define $n$ to be the function that takes in another function and an argument, and applies it $n$ times
  - $\lambda f \lambda x. f(f(f...(fx)))$

```
(define Zero
  (Lam 'f (Lam 'x (Var 'x)))) ;; Function that returns its argument
;; Add one to a number
(define (Add1 n)
  (Lam 'f (Lam 'x) (App (Var 'f) (App (App n f) x))))
(define (Plus m n)
  (Lam 'f (Lam 'x) (App m (App (App n f) x))))
```

# Pairs

- You can view a pair as a function that takes in a boolean and returns either the first or second value depending on that boolean

# Pairs

- You can view a pair as a function that takes in a boolean and returns either the first or second value depending on that boolean

# Pairs

- You can view a pair as a function that takes in a boolean and returns either the first or second value depending on that boolean

```
(define (Pair x y)
  (Lam 'z (If (Var 'z) x y)))
(define (Fst pr)
  (App pr True))
(define (Snd pr)
  (App pr False))
```

# Recursion: The Y Combinator

- Not the startup funder

# Recursion: The Y Combinator

- Not the startup funder
- $\lambda f.\,(\lambda x.\,f\ (x\ x))(\lambda x.\,f\ (x\ x)))$

# Recursion: The Y Combinator

- Not the startup funder
- $\lambda f.\,(\lambda x.\,f\ (x\ x))(\lambda x.\,f\ (x\ x)))$

## Recursion: The Y Combinator

- Not the startup funder
- $\lambda f.\,(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)))$

```
(define Y
  (Fun 'f (App (Fun 'x (App f (App x x)))
               (Fun 'x (App f (App x x))))))
```

- Takes a function `f` with an extra parameter `self`

# Recursion: The Y Combinator

- Not the startup funder
- $\lambda f.\,(\lambda x.\,f\ (x\ x))(\lambda x.\,f\ (x\ x)))$

```
(define Y
  (Fun 'f (App (Fun 'x (App f (App x x)))
               (Fun 'x (App f (App x x))))))
```

- Takes a function `f` with an extra parameter `self`
- Makes a new function where each call to `self` is replaced by a call to `f`

# Recursion: The Y Combinator

- Not the startup funder
- $\lambda f.\,(\lambda x.\,f\ (x\ x))(\lambda x.\,f\ (x\ x)))$

```
(define Y
  (Fun 'f (App (Fun 'x (App f (App x x)))
               (Fun 'x (App f (App x x))))))
```

- Takes a function `f` with an extra parameter `self`
- Makes a new function where each call to `self` is replaced by a call to `f`
- You don't need to know the details of how this work, just that it's possible to do recursion in the UTLC

# Recursion: The Y Combinator

- Not the startup funder
- $\lambda f.\, (\lambda x.\, f\ (x\ x))(\lambda x.\, f\ (x\ x)))$

```
(define Y
  (Fun 'f (App (Fun 'x (App f (App x x)))
               (Fun 'x (App f (App x x))))))
```

- Takes a function `f` with an extra parameter `self`
- Makes a new function where each call to `self` is replaced by a call to `f`
- You don't need to know the details of how this work, just that it's possible to do recursion in the UTLC
- Once we have recursion, we have loops