# 15-150

# Principles of Functional Programming

Slides for Lecture 2

## Functions (continued)

January 15, 2026

Michael Erdmann

# Lessons:

- Recall: Declarations, Bindings, Closures
- Function Application
- Recursion
- Patterns
- Functions as first-class values
- Some comments about $\cong$ and totality

(Recall from last time:)

# Declarations and Bindings

# Recall: Declarations & Bindings

Here is one *declaration*:

```
val pi : real = 3.14159
```

*Binding* (behind the scenes in the *environment*):

[3.14159/pi]

# Recall: Declarations & Bindings

Here are two *declarations*:

```
val pi : real = 3.14159
fun area (r:real) : real = pi*r*r
```

*Bindings* (behind the scenes in the *environment*):

[3.14159/pi]

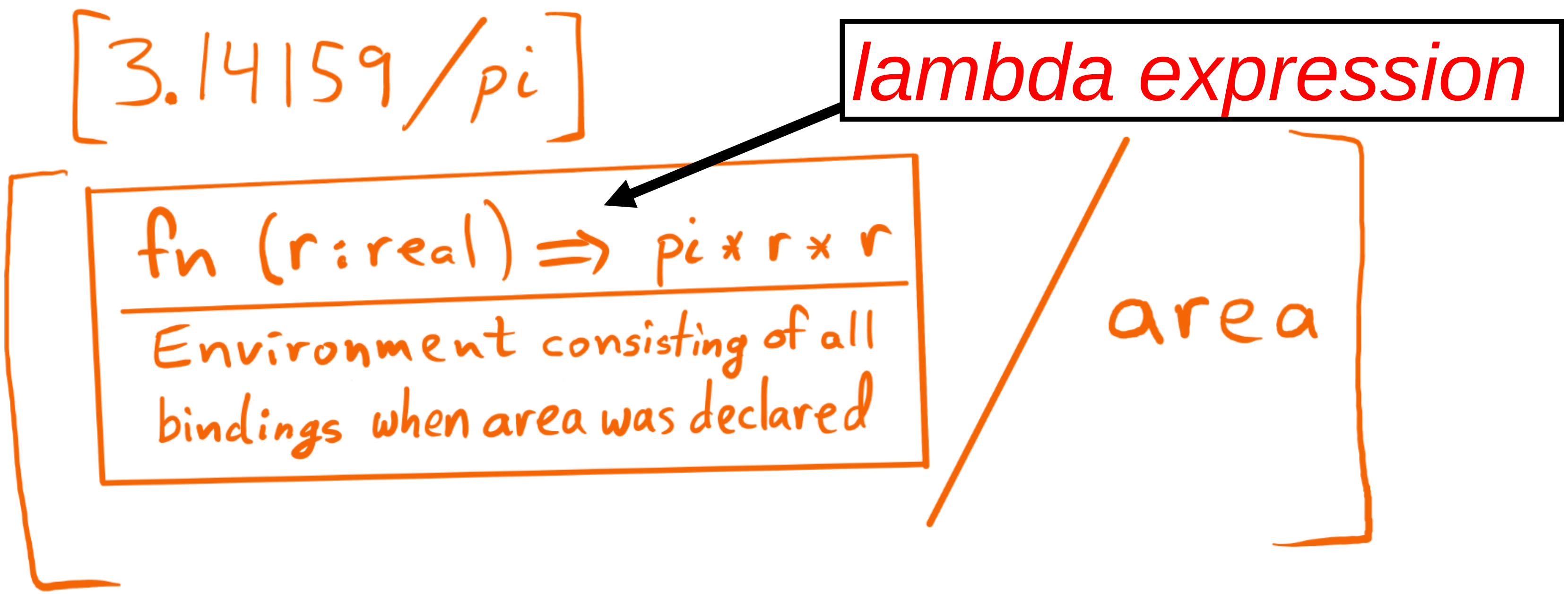
[closure/area]

# Recall: Declarations & Bindings

Here are two *declarations*:

```
val pi : real = 3.14159
fun area (r:real) : real = pi*r*r
```

*Bindings* (behind the scenes in the *environment*):



# Function Application

# Function Application

What does SML do with this expression?

```
area (2.1 + 1.9)
```

Let's look at the more general case first:

```
e2 e1
```

(Then we will come back to the specific expression.)

## ① Typecheck `e2 e1`

# Typechecking Rules `e2 e1`

- `(fn (x:`$t_1$`) => body) :` $t_1 \rightarrow t_2$

  if `body :` $t_2$ assuming `x:`$t_1$.

# Typechecking Rules **e2 e1**

- **(fn (x:$t_1$) => body) : $t_1$ -> $t_2$**

  if **body : $t_2$** assuming **x:$t_1$**.

---

The type of **x** matters!

For instance, if **body** is **x + 9**, then **body** only has a well-defined type if **x : int**.

# Typechecking Rules **e2 e1**

- **(fn (x:$t_1$) => body) : $t_1$ -> $t_2$**

  if **body : $t_2$** assuming **x:$t_1$**.

---

**(fn (x:int) => x) : ?????**

**(fn (x:real) => x): ?????**

# Typechecking Rules **e2 e1**

- **(fn (x:$t_1$) => body) : $t_1$ -> $t_2$**

  if **body : $t_2$** assuming **x:$t_1$**.

---

**(fn (x:int) => x) : int -> int**

**(fn (x:real) => x): real -> real**

# Typechecking Rules **e2 e1**

- **(fn (x:$t_1$) => body) : $t_1$ -> $t_2$**

  if **body : $t_2$** assuming **x:$t_1$**.

- **e2 e1 : $t_2$**

  if **e2 : $t_1$ -> $t_2$**
  and **e1 : $t_1$**.

# ① Typecheck `area (2.1 + 1.9)`

- `area : real -> real`

Actually, area is a variable bound to a closure containing this lambda expression.

Why?

Because `area` is the lambda expression

```
fn (r:real) => pi*r*r
```

and `pi*r*r : real`

given that `pi:real` (by its declaration)

and `r:real` (by type annotation).

# ① Typecheck `area (2.1 + 1.9)`

- `area : real -> real`
- `(2.1 + 1.9) : real`

Why?

Because `2.1 : real`
and `1.9 : real`
and the symbol `+` here represents
the addition function with type
`real * real -> real`.

# ① Typecheck `area (2.1 + 1.9)`

- `area : real -> real`
- `(2.1 + 1.9) : real`
- So `area (2.1 + 1.9) : real`

In particular, the expression is *well-typed*.

REMEMBER:

SML will *only* evaluate an expression
if the expression is well-typed.

# ② Evaluate `e2 e1`

# Evaluation Rules: `e2 e1`

(1) Reduce `e2` to a (function) value.

> Recall: This is a closure containing a lambda expression `(fn (x:t) => body)` and an environment `env` consisting of the bindings present when the function was defined.

(2) Reduce `e1` to a value `v`.

(3) Extend `env` with the binding `[v/x]`.

(4) Evaluate `body` in this extended environment.

> Step (2) occurs only if step (1) produces a value.
> Steps (3) and (4) occur only if steps (1) and (2) produce values.
> Step (4) may or may not produce a value.

# Evaluation Rules: `e2 e1`

(1) Reduce **`e2`** to a (function) value.

> Recall: This is a closure containing a lambda expression **`(fn (x:t) => body)`** and an environment **`env`** consisting of the bindings present when the function was defined.

(2) Reduce **`e1`** to a value **`v`**.

(3) Extend **`env`** with the binding **`[v/x]`**.

(4) Evaluate **`body`** in this extended environment.

If evaluation of **`body`** produces a value **`w`**, return **`w`** in the *original calling environment*.

## ② Evaluate `area (2.1 + 1.9)`

```
   area (2.1 + 1.9)

⟹ [3.14159/pi]
   (fn (r:real) => pi*r*r)(2.1 + 1.9)

⟹ [3.14159/pi](fn (r:real) => pi*r*r) 4.0

⟹ [3.14159/pi][4.0/r] pi*r*r

⟹ 50.26544
```

(Often we leave off the environments when we write reductions, but I wrote them here to be explicit.)

# Evaluation Summary

```
val pi : real = 3.14159

fun area (r:real) : real = pi*r*r

area (2.1 + 1.9) ↪ 50.26544
```

# Evaluation Summary & Question

```
val pi : real = 3.14159

fun area (r:real) : real = pi*r*r

area (2.1 + 1.9) ↪ 50.26544

val pi : real = 0.0
```

# Evaluation Summary & Question

```
val pi : real = 3.14159

fun area (r:real) : real = pi*r*r

area (2.1 + 1.9) ↪ 50.26544

val pi : real = 0.0

area (2.1 + 1.9) ↪ ?????????
```

# Evaluation Summary & Question

```
val pi : real = 3.14159

fun area (r:real) : real = pi*r*r

area (2.1 + 1.9) ↪ 50.26544

val pi : real = 0.0

area (2.1 + 1.9) ↪ ????????
```

Answer: Same as before, `50.26544`.

Why?

# Evaluation Summary & Question

```
val pi : real = 3.14159

fun area (r:real) : real = pi*r*r

area (2.1 + 1.9) ↪ 50.26544

val pi : real = 0.0

area (2.1 + 1.9) ↪ ????????
```

Answer: Same as before, `50.26544`.

Why? Because when `area` is defined, `pi` is bound to `3.14159`.

# Recursion

A math text might define the factorial function by:

$$\mathbf{fact(0) = 1},$$

$$\mathbf{fact(n) = n*(fact(n\text{-}1))}, \text{ for all } \mathbf{n > 0}.$$

(And then write **n!** as mathematical shorthand for **fact(n)**.)

A math text might define the factorial function by:

**fact(0) = 1**,

**fact(n) = n∗(fact(n-1))**, for all **n > 0**.

(And then write **n!** as mathematical shorthand for **fact(n)**.)

That math definition becomes SML code like this:

```
(* fact : int -> int
   REQUIRES: n >= 0
   ENSURES:  fact(n) ==> n!
*)

fun fact(0:int):int = 1
  | fact(n:int):int = n*(fact(n-1))
```

A math text might define the factorial function by:

**fact(0) = 1**,

**fact(n) = n*(fact(n-1))**, for all $n > 0$.

(And then write **n!** as mathematical shorthand for **fact(n)**.)

That math definition becomes SML code like this:

```
1 (* fact : int -> int
2    REQUIRES: n >= 0
3    ENSURES:  fact(n) ==> n!
*)

4 fun fact(0:int):int = 1
    | fact(n:int):int = n*(fact(n-1))

5 val 1 = fact 0
  val 720 = fact 6
```

# Patterns

# Function Clauses & Pattern Matching

```
fun fact(0:int):int = 1
  | fact(n:int):int = n*(fact(n-1))
```

There are two *function clauses* in this code.

The first clause starts with keyword **`fun`**.
The second clause starts with the "or bar" **`|`** .

After that, each clause is of the form

**`fact`** ***`pattern`*** **`=`** ***`expression`***

- When SML evaluates an expression of the form **`fact(`*`value`*`)`**, SML tries to match ***`value`*** against each ***`pattern`*** (in sequential order).
- If a pattern match succeeds, SML creates variable bindings whenever the pattern includes variables, then evaluates the corresponding ***`expression`***.
  - For **`fact(0)`**, **`0`** matches the first pattern and SML evaluates **`1`**.
  - For **`fact(3)`**, **`3`** matches the second pattern and SML creates binding **`[3/n]`**, which then is in scope for evaluation of **`n*(fact(n-1))`**.

# General Form

```
fun f p1 = e1
  | f p2 = e2
         ⋮
  | f pk = ek
```

Each **pj** is a *pattern* and each `ej` is an expression.

<u>NOTE:</u>

If `f : t -> t'`, then
each pattern `pj` must match type `t`,
and each expression `ej` must have type `t'`,
given the types of any variables in `pj`.

# General Form

```
fun f p1 = e1
  | f p2 = e2
         ⋮
  | f pk = ek
```

Each **pj** is a *pattern* and each **`ej`** is an expression.

When evaluating **`f(v)`** for some value **`v`**, SML will try to match **`v`** against **`p1`**, then **`p2`**, etc., until a match **`pj`** succeeds (including any variable bindings needed), at which point SML evaluates **`ej`**.

If no pattern matches **`v`**, evaluation will result in a fatal runtime error. For this reason, the set of patterns {**`pj`**} should cover all possibilities. SML will give a "nonexhaustive" warning if that is not the case when **`f`** is declared. SML will also raise a fatal error when **`f`** is declared if there are redundant (i.e., extra) patterns.

# What is a pattern?

For now, a pattern can be any of the following:

- a constant (e.g., **`3`**, **`true`**, **`"abc"`**; no reals)
- a variable
- a tuple of subpatterns
- the wildcard **`_`** (which matches anything)

Patterns must be linear, meaning any variable can appear at most once in any one pattern.

In the future, we will see additional patterns coming from datatypes (such as lists).

# Tuples

# Patterns can appear in declarations

Example:

```
val (k,r) : int * real = (2, 3.14)
```

This pattern is a tuple -- a pair whose two subpatterns are each variables.

# Patterns can appear in declarations

Example:

```
val (k,r) : int * real = (2, 3.14)
```

The declaration creates two variable bindings (behind the scenes in the environment):

```
[2/k, 3.14/r]
```

# Patterns can appear in declarations

Example:

```
val 49 : int  =  square(7)
```

This pattern is a constant.

This “declaration” contains no variables.
It will succeed only if the value
returned by **`square`** is **`49`**.
So it amounts to a test.
(Tests can have more elaborate patterns.)

# Patterns can appear in declarations

In this example, a pattern extracts tuple elements:

```
(* fibb : int -> int * int
   REQUIRES: n >= 0
   ENSURES:  fibb(n) ==> (f_n, f_n-1)
   with f_n the n^th Fibonacci number (let f_-1 = 0).
*)

fun fibb (0:int):int*int = (1,0)
  | fibb  n =
      let
         val (a:int, b:int) = fibb(n-1)
      in

      end
```

| | |
|---|---|
| $n$ | 0,1,2,3,4,5, 6, 7 |
| $f_n$ | 1,1,2,3,5,8,13,21 |

**This is how you should extract elements from a tuple.**

# Patterns can appear in declarations

In this example, a pattern extracts tuple elements:

| | |
|---|---|
| n | 0,1,2,3,4,5, 6, 7 |
| $f_n$ | 1,1,2,3,5,8,13,21 |

```
(* fibb : int -> int * int
   REQUIRES: n >= 0
   ENSURES:  fibb(n) ==> (f_n, f_n-1)
   with f_n the n^th Fibonacci number (let f_-1 = 0).
*)

fun fibb (0:int):int*int = (1,0)
  | fibb  n =
      let
         val (a:int, b:int) = fibb(n-1)
      in
           ?????????????
      end
```

# Patterns can appear in declarations

In this example, a pattern extracts tuple elements:

```
(* fibb : int -> int * int
   REQUIRES: n >= 0
   ENSURES:  fibb(n) ==> (f_n, f_{n-1})
   with f_n the n^th Fibonacci number (let f_{-1} = 0).
*)

fun fibb (0:int):int*int = (1,0)
  | fibb  n =
      let
         val (a:int, b:int) = fibb(n-1)
      in
         (a+b, a)
      end

val (21, 13) = fibb 7
```

| $n$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $f_n$ | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 |

# Patterns can appear in declarations

In this example, a pattern extracts tuple elements:

| | |
|---|---|
| $n$ | 0,1,2,3,4,5, 6, 7 |
| $f_n$ | 1,1,2,3,5,8,13,21 |

```
(* fibb : int -> int * int
   REQUIRES: n >= 0
   ENSURES:  fibb(n) ==> (f_n, f_n-1)
   with f_n the n^th Fibonacci number (let f_-1 = 0).
*)

fun fibb (0:int):int*int = (1,0)
  | fibb  n =
      let
         val (a:int, b:int) = fibb(n-1)
      in
         (a+b, a)
      end

val (21, _) = fibb 7
```

case

# Patterns appear in case expressions

```
(case e of
      p1 => e1
    | p2 => e2
        ⋮
    | pk => ek)
```

Note: `=>` (not `=`).

# Patterns appear in case expressions

```
(case e of
        p1 => e1
      | p2 => e2
           ⋮
      | pk => ek)
```

- Semantics similar to functions, with `e` playing role of argument.
- Typechecking:
  - Expression `e` must have a type `t'` and all `pj` must be able match type `t'`.
  - The expressions `ej` must all have the same type, call it `t` (given types of variables in associated patterns).
  - Type `t` is the overall type of the case expression.

# Patterns appear in case expressions

```
(case (e : t′) of
      p1 => e1
    | p2 => e2
        ⋮
    | pk => ek)
```

: t

- Semantics similar to functions, with `e` playing role of argument.
- Typechecking:
  - Expression `e` must have a type `t′` and all `pj` must be able match type `t′`.
  - The expressions `ej` must all have the same type, call it `t` (given types of variables in associated patterns).
  - Type `t` is the overall type of the case expression.

# Patterns appear in case expressions

```
(case e of
      p1 => e1
    | p2 => e2
         ⋮
    | pk => ek)
```

- Semantics similar to functions, with **`e`** playing role of argument.
- Typechecking:
  - Expression **`e`** must have a type **`t'`** and all **`pj`** must be able match type **`t'`**.
  - The expressions **`ej`** must all have the same type, call it **`t`** (given types of variables in associated patterns).
  - Type **`t`** is the overall type of the case expression.
- If typechecking succeeds, SML evaluates **`e`**. If **`e`** reduces to value **`v`**, SML matches **`v`** against **`p1`**, **`p2`**, …, then evaluates **`ej`** of first matching **`pj`** (if any). If **`ej`** reduces to a value **`w`**, SML returns **`w`** as the value of the **`case`**.

# `case` is useful to avoid nested `if-then-else`

```
(* example : int -> int
   REQUIRES: true
   ENSURES:  example(x) returns
                              0 if x = 1,
                        x*x - 1 if x < 1,
                  and 1 - x*x*x if x > 1.
*)

fun example (x:int):int =
      (case (square x, x > 0) of


       |
       |                              )
```

# `case` is useful to avoid nested `if-then-else`

```
(* example : int -> int
   REQUIRES: true
   ENSURES:  example(x) returns
                             0 if x = 1,
                       x*x - 1 if x < 1,
               and 1 - x*x*x if x > 1.
*)

fun example (x:int):int =
      (case (square x, x > 0) of
         (1, true)     => 0
       |
       |                              )
```

## `case` is useful to avoid nested `if-then-else`

```
(* example : int -> int
   REQUIRES: true
   ENSURES:  example(x) returns
                                0 if x = 1,
                          x*x - 1 if x < 1,
                      and 1 - x*x*x if x > 1.
*)

fun example (x:int):int =
      (case (square x, x > 0) of
         (1, true)    => 0
        | (sqr, false) => sqr - 1
        |                          )
```

# `case` is useful to avoid nested `if-then-else`

```
(* example : int -> int
   REQUIRES: true
   ENSURES:  example(x) returns
                                     0 if x = 1,
                               x*x - 1 if x < 1,
                         and 1 - x*x*x if x > 1.
*)
```

If second clause is relevant, get binding `[v/sqr]`,with `v` value of `square x`.

```
fun example (x:int):int =
      (case (square x, x > 0) of
         (1, true)     => 0
       | (sqr, false) => sqr - 1
       |                               )
```

## `case` is useful to avoid nested `if-then-else`

```
(* example : int -> int
   REQUIRES: true
   ENSURES:  example(x) returns
                          0 if x = 1,
                    x*x - 1 if x < 1,
              and 1 - x*x*x if x > 1.
*)

fun example (x:int):int =
      (case (square x, x > 0) of
          (1, true)       => 0
        | (sqr, false) => sqr - 1
        | (sqr, _)     => 1 - x*sqr)
```

# `case` is useful to avoid nested `if-then-else`

```
(* example : int -> int
   REQUIRES: true
   ENSURES:  example(x) returns
                                     0 if x = 1,
                               x*x - 1 if x < 1,
                         and 1 - x*x*x if x > 1.
*)
```

If third clause is relevant, get binding `[v/sqr]`, with `v` value of `square x`.

```
fun example (x:int):int =
      (case (square x, x > 0) of
         (1, true)     => 0
       | (sqr, false) => sqr - 1
       | (sqr, _)     => 1 - x*sqr)
```

# Functions as First-Class Values

# Passing a function as an argument

```
(* sqrf : (int -> int) * int -> int
   REQUIRES: true
   ENSURES:  sqrf (f, x) ==> (f(x))*(f(x))
*)
```

The argument type is a pair consisting of an `int -> int` function and an `int`.

# Passing a function as an argument

```
(* sqrf : (int -> int) * int -> int
   REQUIRES: true
   ENSURES:  sqrf (f, x) ==> (f(x))*(f(x))
*)

fun sqrf (f : int -> int, x : int) : int =
      square(f(x))


(* Testing *)
val 36 = sqrf (fn (n:int) => n + 2, 4)
```

# Passing a function as an argument

```
(* sqrf : (int -> int) * int -> int
   REQUIRES: true
   ENSURES:  sqrf (f, x) ==> (f(x))*(f(x))
*)

fun sqrf (f : int -> int, x : int) : int =
      square(f(x))


(* Testing *)
val 36 = sqrf (fn (n:int) => n + 2, 4)
```

Notice how we can write an anonymous lambda expression inline.

# Passing a function as an argument

```
(* sqrf : (int -> int) * int -> int
   REQUIRES: true
   ENSURES:  sqrf (f, x) ==> (f(x))*(f(x))
*)

fun sqrf (f : int -> int, x : int) : int =
     square(f(x))
```

## Puzzle:

```
fun dotwice (f : int -> int, x : int) : int =
      sqrf (fn (n:int) => sqrf(f,n), x)
```

# Passing a function as an argument

```
(* sqrf : (int -> int) * int -> int
   REQUIRES: true
   ENSURES:  sqrf (f, x) ==> (f(x))*(f(x))
*)

fun sqrf (f : int -> int, x : int) : int =
     square(f(x))
```

## Puzzle:

```
fun dotwice (f : int -> int, x : int) : int =
      sqrf (fn (n:int) => sqrf(f,n), x)

dotwice (fn (k:int) => k, 3) ↪ ????????
```

identity function

# Passing a function as an argument

```
(* sqrf : (int -> int) * int -> int
   REQUIRES: true
   ENSURES:  sqrf (f, x) ==> (f(x))*(f(x))
*)

fun sqrf (f : int -> int, x : int) : int =
      square(f(x))
```

## Puzzle:

```
fun dotwice (f : int -> int, x : int) : int =
      sqrf (fn (n:int) => sqrf(f,n), x)

dotwice (fn (k:int) => k, 3) ↪ ????????
```

identity function

Answer: 81

# Some comments about $\cong$

- If $e_1 \hookrightarrow v$ and $e_2 \hookrightarrow v$, with $v$ a value, then $e_1 \cong e_2$.

- If $e_1 \Longrightarrow e_2$, then $e_1 \cong e_2$.

- If $e_1 \Longrightarrow e$ and $e_2 \Longrightarrow e$, with $e$ an expression, then $e_1 \cong e_2$.

- Caution: $e_1 \cong e_2$ does <u>not</u> necessarily imply that $e_1 \Longrightarrow e_2$ or $e_2 \Longrightarrow e_1$.

  Ex: $1+1+1+7 \cong 2*5$.

# A comment about $\cong$

$$[3/y, 5/z]\,(\text{fn } (x:\text{int}) \Rightarrow x+y+z)$$

$$\cong$$

$$(\text{fn } (x:\text{int}) \Rightarrow x+8)$$

However, they are not equal, nor does one reduce to the other.

In particular, the addition $y+z$ does **not** happen when $(\text{fn } (x:\text{int}) \Rightarrow x+y+z)$ is written/defined.
(The body $x+y+z$ is evaluated when the function is applied to an argument as per our evaluation rules.)

# A comment about $\cong$

[Basis Library bindings] [3/y, 5/z] (fn (x:int) => x+y+z)

$$\cong$$

[Basis Library bindings] (fn (x:int) => x+8)

However, they are not equal, nor does one reduce to the other.

In particular, the addition y+z does **not** happen when (fn (x:int) => x+y+z) is written/defined.
(The body x+y+z is evaluated when the function is applied to an argument as per our evaluation rules.)

# A comment about totality

Suppose $f : int \to int$.

If $f$ is total, then

$$f(1) + f(2) \cong f(2) + f(1).$$

Why?

If $f$ is possibly not total, then maybe

$$f(1) + f(2) \not\cong f(2) + f(1).$$

Why?

That is all.

Please have a good weekend.

See you Tuesday.