CSE216 Programming Abstractions

Procedural Abstraction

YoungMin Kwon

Elements of Programming

- Primitive expressions
  - The simplest entries the language is concerned with
- Means of combination
  - By which compound elements are built from simpler ones
- Means of abstraction
  - By which compound elements can be named and manipulated as units

Numbers

- Number
  - A primitive expression
  - Type 123;; to the OCaml interactive system (a.k.a. toplevel)

prompt

a number expr.

ask top level to evaluate

```
# 123;;
- : int = 123
```

result type

result value

Combining Numbers

- Arithmetic operators
  - Using arithmetic operators
    - `+ - * / mod    +. -. *. /. **`
  - Type 1 + 2 * 3 in the OCaml top level

```
# 1 + 2 * 3;;
- : int = 7
```

  - Type 1. +. 2. *. 3.

```
# 1. +. 2. *. 3.;;
- : float = 7.
```

Combining Numbers

- Arithmetic operators
  - For each operator, there is a corresponding function

```
# (+);;
- : int -> int -> int = <fun>
```

  - Function application
    - No parenthesis around parameters
    - Parameters are separated by spaces

```
# (+) 1 2;;
- : int = 3
```

Combining Numbers

- Type coercion is not automatic in OCaml

```
# 1.0 + 2.0;;
Characters 0-3:
  1.0 + 2.0;;
  ^^^
Error: This expression has type float but an expression
         was expected of type int

# 1.0 +. 2.0;;
- : float = 3.
# (+.);;
-  : float -> float -> float = <fun>

# float_of_int 1;;  (* or float 1 *)
- : float = 1.
# int_of_float 1.5;;
- : int = 1
```

Abstraction by Names

- Names are to refer to objects
  - Name: variable
  - Its value: object
  - Names provide a mean of abstraction

- Create a variable to name a value
  - let binding

```
let <variable> = <expr>

# let x = 1 + 2;;
val x : int = 3

# let add = (+);;
val add : int -> int -> int = <fun>
```

Abstraction by Names

- Environment
  - A data structure that keeps track of name-value pairs

```
# x;;
-  : int = 3

# add;;
-  : int -> int -> int = <fun>

# add x 1;;
- : int = 4
```

Evaluating Combinations

- How to evaluate a combination (prefix operator case)

    1. Evaluate the sub-expressions of the combination

    2. Apply the function (the first sub-expr) to the arguments (the other sub-exprs)

Evaluating Combinations

- To evaluate sub-expressions
  - If a sub-expr is a combination: recursively evaluate the combination
  - If a sub-expr is a primitive expression
    - Number: the value of the number
    - Built-in operator: the code that executes the operation
    - Name: object associated with the name in the environment

Evaluating Combinations

- Example

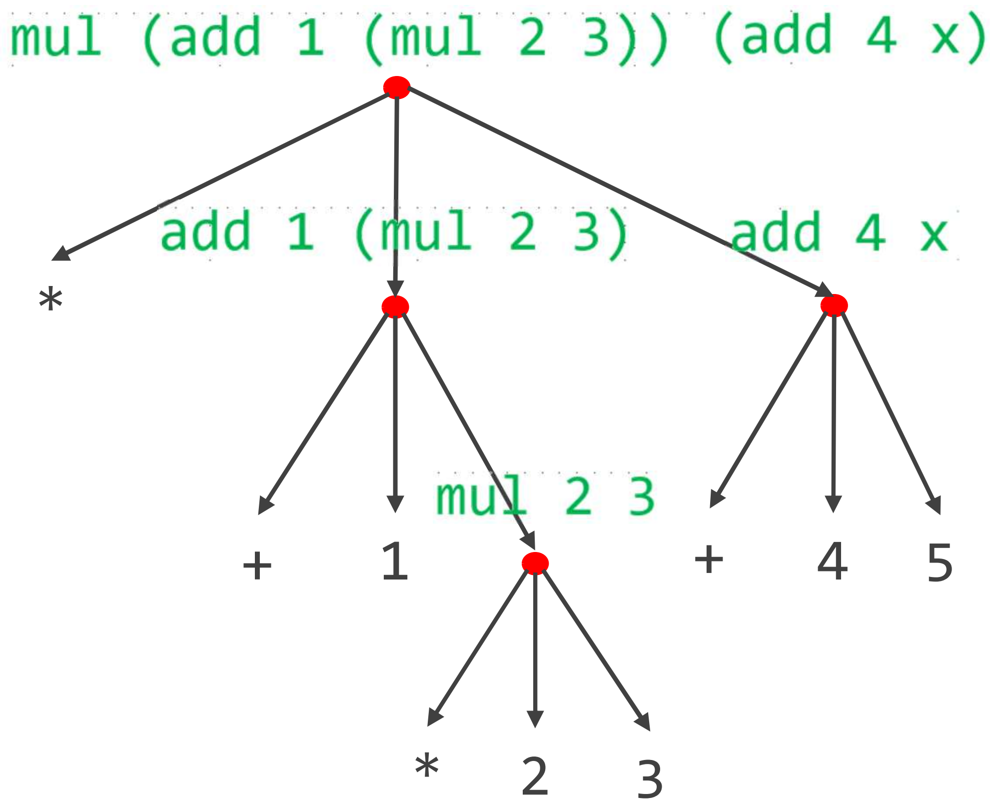
```
# let add = (+);;
val add : int -> int -> int = <fun>

# let mul = ( *);;
val mul : int -> int -> int = <fun>

# let x = 5;;
val x : int = 5

# mul (add 1 (mul 2 3))
      (add 4 x);;
- : int = 63
```

notice the space:
(would start a comment*

Evaluating Combinations

- Example

```
# mul (add 1 (mul 2 3))
      (add 4 x);;
- : int = 63
```



Abstraction by Functions

- Function definition
  - With compound operations, it provides a powerful *abstraction* mechanism

```
let <name> <formal parameters> = <body>
```

```
# let square x = x * x;;
val square : int -> int = <fun>
```

name

formal parameter

body

Abstraction by Functions

- Function application

<operator-expr> <operand-expr>

```
# square 2;;
- : int = 4
```

function

actual parameter

Abstraction by Functions

- Examples

```
# square 3;;
- : int = 9

# square (1+2);;
- : int = 9

# square (square 3);;
- : int = 81

# let sum_of_squares x y = square x + square y;;
val sum_of_squares : int -> int -> int = <fun>

# sum_of_squares 3 4;;
- : int = 25
```

square is used as a building block of another procedure

Abstraction by Functions

- Anonymous function definition

```
fun <formal parameters> -> <body>
```

```
# fun x -> x * x;;
- : int -> int = <fun>
```

anonymous function

formal parameter

body

```
# (fun x -> x * x) 3;;
- : int = 9

# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

Abstraction by Functions

- Multi-parameter functions
  - Nested single parameter functions

```
let add x y = x + y
≡ let add = fun x -> fun y -> x + y
≡ let add = fun x -> (fun y -> x + y)

add 2 3
≡ (add 2) 3

≡ let add2 = add 2
  add2 3
```

  - Pattern matching on a tuple

```
let add (x, y) = x + y
add (2, 3)

≡ let p = (2, 3)
  add p
```

Currying

- Currying
  - Tuple parameter function → nested single parameter functions

```
# let add (a, b) = a + b;;
val add : int * int -> int = <fun>
# add (1, 2);;
- : int = 3

# let add' a b = a + b;;
val add' : int -> int -> int = <fun>
# add' 1 2;;
- : int = 3

# let inc = add' 1;;
val inc : int -> int = <fun>
# inc 2;;
- : int = 3
```

Currying

- A curry function

```
(*take x and y separately and apply them together as a pair*)
let curry f = fun x -> fun y -> f (x, y)

let add' = curry add
let inc = add' 1
let _ = inc 3


val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
val add : int * int -> int = <fun>
val add' : int -> int -> int = <fun>
val inc : int -> int = <fun>
- : int = 4
```

Function Composition

- Function composition operator (|>)
  - A way to avoid nested function calls
  - A way to bind a temporary result to a variable

```
(*function composition operator*)

let (|>) x f = f x

let inc x = (+)  1

let _ =  1 |> inc |> inc |> inc
- : int = 4
```

Function Composition

```
(*function composition*)
let (|>) x f = f x
val ( |> ) : 'a -> ('a -> 'b) -> 'b = <fun>


let square x = x * x
val square : int -> int = <fun>


let sum_of_squares x y =
    square x |> fun xx -> (*bind result to a temp. var. xx*)
    square y |> fun yy -> xx + yy
val sum_of_squares : int -> int -> int = <fun>


let _ = sum_of_squares 3 4
- : int = 25
```

Order of Evaluation

- Application order
  - Evaluate the parameters and then apply the function

```
# let if_then_else p t f = if p then t else f;;
val if_then_else : bool -> 'a -> 'a -> 'a = <fun>

# if_then_else (1 < 2) 1 2;;
- : int = 1

# if_then_else (1 < 2) (1 / 1) (1 / 0);;
Exception: Division_by_zero.
```

Order of Evaluation

- Substitution model for function application
  - Evaluate the body with each formal parameter replaced by its actual parameter

```
sum_of_squares 3 4

  ⇒ square 3 + square 4

  ⇒ (mul 3 3) + (mul 4 4)

  ⇒ (( * ) 3 3) + (( * ) 4 4))

  ⇒ 9 + 16

  ⇒ 25
```

```
let mul = ( * )

let square x = mul x x

let sum_of_squares x y =
    square x + square y
```

Order of Evaluation

- Normal order
  - Substitute operand expressions for parameters until only primitive expressions left
    - Do not evaluate the operands until their values are needed

```
# if_then_else (1 < 2) (1 / 1) (1 / 0);; ??

# if 1 < 2 then 1 / 1 else 1 / 0;;
- : int = 1
```

Boolean Expression

- Bool
  - Primitive expressions

prompt

a bool expr.

ask top level to evaluate

```
# true;;
- : bool = true
# false;;
- : bool = false
```

result type

result value

Comparisons

```
# (=);;
- : 'a -> 'a -> bool = <fun>
# (<>);;
- : 'a -> 'a -> bool = <fun>
# (>);;
- : 'a -> 'a -> bool = <fun>

# 2 > 1;;
- : bool = true

# 2. > 1.;;
- : bool = true

# 2 > 1.;;
Characters 4-6:
  2 > 1.;;
      ^^
Error: This expression has
Type float but an expression
was expected of type int
```

```
# int_of_float 1. > 2;;
- : bool = false

# float_of_int 1 > 2.;;
- : bool = false

# float 1 > 2.;;
- : bool = false

# "abc" = "abc";;
- : bool = true

# "abc" <> "abc";;
- : bool = false

# "abc" < "def";;
- : bool = true
```

Comparisons

```
(* =,  <>: compare structures,
   ==, !=: compare addresses *)
# (==);;
- : 'a -> 'a -> bool = <fun>
# (!=);;
- : 'a -> 'a -> bool = <fun>

# "hello" = "hello";;
- : bool = true

# "hello" <> "hello";;
- : bool = false

# "hello" == "hello";;
- : bool = false

# "hello" != "hello";;
- : bool = true
```

```
# let v = "hello";;
val v : string = "hello"
# v = v;;
- : bool = true
# v <> v;;
- : bool = false
# v == v;;
- : bool = true
# v != v;;
- : bool = false

# let u = v;;
val u : string = "hello"
# u == v;;
- : bool = true
# u != v;;
- : bool = false
```

Logical Connectives

- Logical connectives: &&, ||, not
  - ! is a dereference operator

```
# (&&);;
-  : bool -> bool -> bool = <fun>

# let inside lb ub x = lb <= x && x <= ub;;
val inside : 'a -> 'a -> 'a -> bool = <fun>

# inside 0 10 5;;
-  : bool = true

# let outside lb ub x = not (inside lb ub x);;
val outside : 'a -> 'a -> 'a -> bool = <fun>

# outside 0 10 5;;
- : bool = false
```

Logical Connectives

- Evaluation order of && and ||

```
# false && 1/0 > 0;;
- : bool = false

# (&&) false (1/0 > 0);; (* not exactly normal order
- : bool = false            eval., but similar to it *)

# true || 1/0 > 0;;
- : bool = true

# false || 1/0 > 0;;
Exception: Division_by_zero.
```

Conditional Expressions

- Predicate
  - An expression whose value is interpreted as either true or false
- Conditional expression

```
if <predicate> then <consequent>
               else <alternative>
```

```
# let abs x = if x >= 0 then x else - x;;
val abs : int -> int = <fun>

# abs (-3);;
- : int = 3
```

Conditional Expressions

- Example: factorial

```
# let rec factorial x =
    if x = 0
    then 1
    else x * factorial (x - 1);;
val factorial : int -> int = <fun>

# factorial 4;;
- : int = 24
```

- To define a *recursive function*, use let rec

Conditional Expressions

- Example: even and odd

```
# let rec even x =
    if x = 0 then true else odd (x - 1)
  and odd x =
    if x = 0 then false else even (x - 1);;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>

# even 3;;
-  : bool = false

# odd 3;;
- : bool = true
```

- To define *mutually recursive functions*, use let rec and

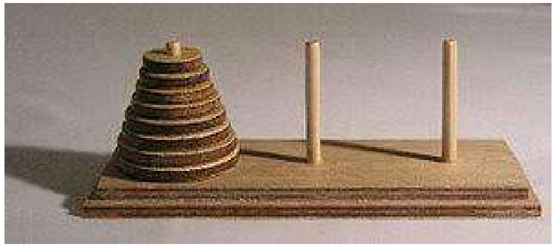
Conditional Expressions

- Example gcd

```
# let rec gcd x y =
    if      x > y then gcd (x - y) y
    else if x < y then gcd (y - x) x
    else x;;
val gcd : int -> int -> int = <fun>

# gcd 15 6;;
- : int = 3
```

Assignment 1

- Implement move function
  - Download TowerOfHanoi.ml and implement its move function

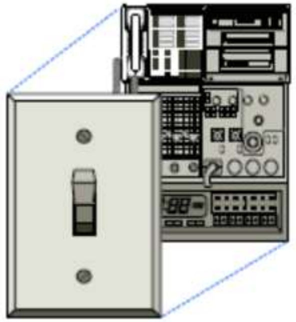


  - Upload TowerOfHanoi.ml to Brightspace
  - Due date: 3/14/2024

```ocaml
(* Tower of Hanoi
*)

(*  TODO: implement move function

    move n src dst aux:
        moves n disks from src to dst using aux

    if n is 1,
        print the movement from src to dst
    otherwise,
        move n-1 disks from src to aux,
        move 1 disk from src to dst, and
        move n-1 disks from aux to dst.
    hint: use Printf.printf "move from %s to %s\n" ...
    hint: for a series of expressions use begin ... end
          e.g. begin move...; move...; move... end
*)

let main () =
    move 3 "A" "B" "C"

let _ = main ()
```

```
(*
expected result:
#use "TowerOfHanoi.ml";;
val move : int -> string -> string -> string -> unit = <fun>
val main : unit -> unit = <fun>
move from A to B
move from A to C
move from B to C
move from A to B
move from C to A
move from C to B
move from A to B
- : unit = ()
*)
```

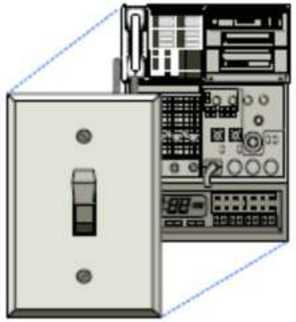
Procedural Abstraction



- Procedural abstraction
  - Regard procedures as a black box
  - Concern only with the fact that a procedure computes the correct result, but not with how
  - Any procedures that compute the result are equally good

Procedural Abstraction

- Example



```
# let square x = x *. x;;
val square : float -> float = <fun>

# let square x = exp (log x +. log x);;
val square : float -> float = <fun>
```

- A user should not need to know how the procedure is implemented in order to use it

- Procedure definitions should be able to suppress details

Procedural Abstraction

- Local names
  - Formal parameter names should not matter to the user of the procedure
    - Parameter names should be local to procedure body

```
# let square x = x *. x;;
val square : float -> float = <fun>

# let square y = y *. y;;
val square : float -> float = <fun>
```

    - These procedures should not be distinguishable

Procedural Abstraction

- Local names

```
# let square x = x *. x;;
val square : float -> float = <fun>

# let sum_of_squares x y = (square x) +. (square y);;
val sum_of_squares : float -> float -> float = <fun>
```

  - x in the body of square should be different from the x in the body of sum_of_squares

Procedural Abstraction

- Computing π (Nilakantha series)

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

- How to run a program from a file
  - To test large programs.
  - Write pi.ml with the definition of pi above
  - In the OCaml top level type `#use "pi.ml";;`

Procedural Abstraction

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

```
(* pi.ml
   Computes pi using Nilakantha series
*)

let abs x =
    if x < 0. then -. x else x

let good_enough guess old_guess tol =
    (abs (guess -. old_guess)) <= tol;;

let term x sign =
    sign *. 4. /. (x *. (x +. 1.) *. (x +. 2.))
```

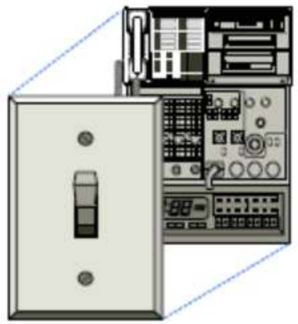
Procedural Abstraction

```
let rec pi_iter guess old_guess x sign tol =
    if good_enough guess old_guess tol
    then guess
    else pi_iter (guess +. (term x sign))
                 guess
                 (x +. 2.)
                 (-. sign)
                 tol
let pi tol =
    pi_iter 3. 0. 2. 1. tol

let _ = pi 1e-10
```

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

```
# #use "pi.ml";;
val abs : float -> float = <fun>
val good_enough : float -> float -> float -> bool = <fun>
val term : float -> float -> float = <fun>
val pi_iter : float -> float -> float -> float -> float -> float…
val pi : float -> float = <fun>
- : float = 3.141592653598846
```

Procedural Abstraction



- Internal definitions
  - In the previous program,
    - pi is the only procedure that is important to users
    - The other procedures only clutter up their minds
  - Solution ⇒ allow procedures to have internal definitions that are local to the procedure

Procedural Abstraction

- Block structure
  - Nesting of definitions

    let <variable> = <expr1> in <expr2>

    - In expr2, variable is equal to expr1
    - let binding is equivalent to

      (fun <variable> -> <expr2>) <expr1> or
      <expr1> |> fun <variable> -> <expr2>

```
let foo () =
    let x = 1 in
    let y = x + 1 in
    let z = y + 1 in
    z + 3
```

```
let foo' () =
    (fun x ->
        (fun y ->
            (fun z -> z + 3)
            (y + 1))
        (x + 1))
    1
```

```
let foo'' () =
    1     |> fun x ->
    x + 1 |> fun y ->
    y + 1 |> fun z ->
    z + 3
```

```ocaml
(* compute pi, Nilakantha series *)
let pi tol =
    let rec pi_iter guess old_guess step sign =
        let good_enough () =    (*(), called unit, is like void*)
            let abs x =
                if x < 0. then -. x else x in
            (abs (guess -. old_guess)) <= tol in

        let term x =
            sign *. 4. /. (x *. (x +. 1.) *. (x +. 2.)) in

        if good_enough ()
        then guess
        else pi_iter (guess +. (term step))
                     guess
                     (step +. 2.)
                     (-. sign) in
    pi_iter 3. 0. 2. 1.

let _ = pi 1e-10
```

```
# #use "pi_iter2.ml";;
val pi : float -> float = <fun>
- : float = 3.141592653539846
```

Variable Binding

- Variable binding
  - Associate variable names with values
  - *Bound* variable: a variable that is bound to a value
  - *Free* variable: a variable that is not bound
  - *Scope*: the set of expressions for which a binding defines a name

Variable Binding

- Variable binding
  - Formal parameters are bound to actual parameters
  - The scope of formal parameters is the procedure body

y is free

```
# (fun x y -> x + y) 1;;
- : int -> int = <fun>
```

x is bound to 1

scope of x and y

Variable Binding

- Lexical (static) scoping
  - Find the binding from the closest nesting procedures and let bindings
- Dynamic scoping
  - Each time a function is invoked, a new scope is pushed onto the stack

Variable Binding

```
let first x =
    let a = 1 in
    let second x =
        let b = 2 in
        let third x =
            x + a + b in
        let fourth x =
            let a = 3 in
            let b = 4 in
            x + third b in
        x + fourth b in
    x + second a in
first 10 -> 20
```

Lexical scoping

```
third:
x = 4

fourth:
x = 2, a = 3, b = 4

second:     x = 1, b = 2,
third = …, fourth = …

first:
x = 10, a = 1, second = …

first 10 -> 24
```

Dynamic scoping

Higher-Order Procedures

- First-class elements
  - Named by variables
  - Passed as arguments to procedures
  - Returned as the results of procedures
  - Included in data structures

- Procedures are a first-class element

Higher-Order Procedures

- Abstractions with higher-order procedures
  - The *same programming pattern* will be used with *different procedures*
  - To express such patterns as concepts, we need *higher-order procedures*

- Higher-order procedures are procedures that
  - Accept procedures as arguments
  - Return procedures as values

Higher-Order Procedures

- Example
  - Sigma notation: an abstraction of summation of a series

$$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b)$$

```
let rec sum term n next b =
    if n > b then 0.
    else (term n) +. (sum term (next n) next b)
```

Higher-Order Procedures

```
let sum_cubes a b =
    let cube x = x ** 3. in
    let inc x = x +. 1. in
    sum cube a inc b

let _ = sum_cubes 0. 3.

- : float = 36.


let sum_ints a b =
    let identity x = x in
    let inc x = x +. 1. in
    sum identity a inc b

let _ = sum_ints 0. 10.

- : float = 55.
```

Higher-Order Procedures

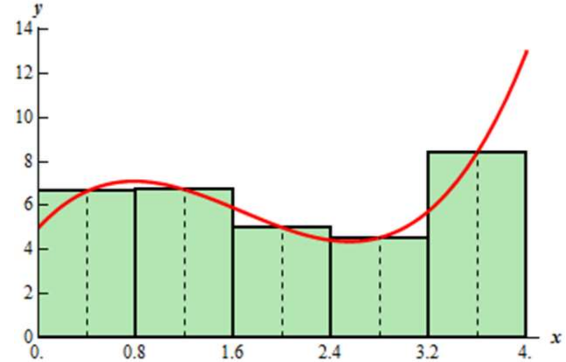
- Computing π (Nilakantha series)

$$\pi = 3 + \frac{4}{2\times3\times4} - \frac{4}{4\times5\times6} + \frac{4}{6\times7\times8} - \frac{4}{8\times9\times10} + \cdots$$

```
let sum_pi n =
    let term x =
        let y = x *. 2. in
        let sign = -1. ** (x +. 1.) in
        sign *. 4. /. (y *. (y +. 1.) *. (y +. 2.)) in
    let inc x = x +. 1. in
    3. +. sum term 1. inc n

let _ = sum_pi 100.

- : float = 3.1415924109719806
```

Higher-Order Procedures

- Numerical integration



$$\int_a^b f = \left[f\left(a + \frac{dx}{2}\right) + f\left(a + dx + \frac{dx}{2}\right) + f\left(a + 2dx + \frac{dx}{2}\right) + \ldots \right] dx$$

```
let integral f a b dx =
    let term x = f (x +. dx /. 2.) in
    let next x = x +. dx in
    dx *. (sum term a next b)

let _ = integral sin 0. 3.141592 0.001

- : float = 2.0000000003679608
```

Lambda

- Anonymous function definition

```
let <name> = fun <formal parameters> -> <body>

# let square = fun x -> x * x;;
val square : int -> int = <fun>
```

- Anonymous recursive function definition

```
let rec <name> = fun <formal parameters> -> <body>

# let rec fact = fun x ->
    if x = 0 then 1 else x * fact (x - 1);;
val fact : int -> int = <fun>
```

Lambda

- Examples

```
let sum_cubes a b =
    let cube x = x ** 3. in
    let inc x = x +. 1. in
    sum cube a inc b

let sum_cubes2 a b =
    sum (fun x -> x ** 3.) a (fun x -> x +. 1.) b


let sum_ints a b =
    let identity x = x in
    let inc x = x +. 1. in
    sum identity a inc b

let sum_ints2 a b =
    sum (fun x -> x) a (fun x -> x +. 1.) b
```

Lambda

- let and lambda
  - let bindings can be rewritten using lambda
  - The following two expressions are equivalent

```
let <name_1> = <expr_1> in
let <name_2> = <expr_2> in
…
let <name_n> = <expr_n> in
<body>
```

```
(fun <name_1>
     <name_2> …
     <name_n> -> <body>)
     <expr_1>
     <expr_2> …
     <expr_n>
```

```
# let x = 3 in
  let y = 4 in
  x + y;;
- : int = 7
```

```
# (fun x y -> x + y) 3 4;;
- : int = 7
```

Example: Bisection Method

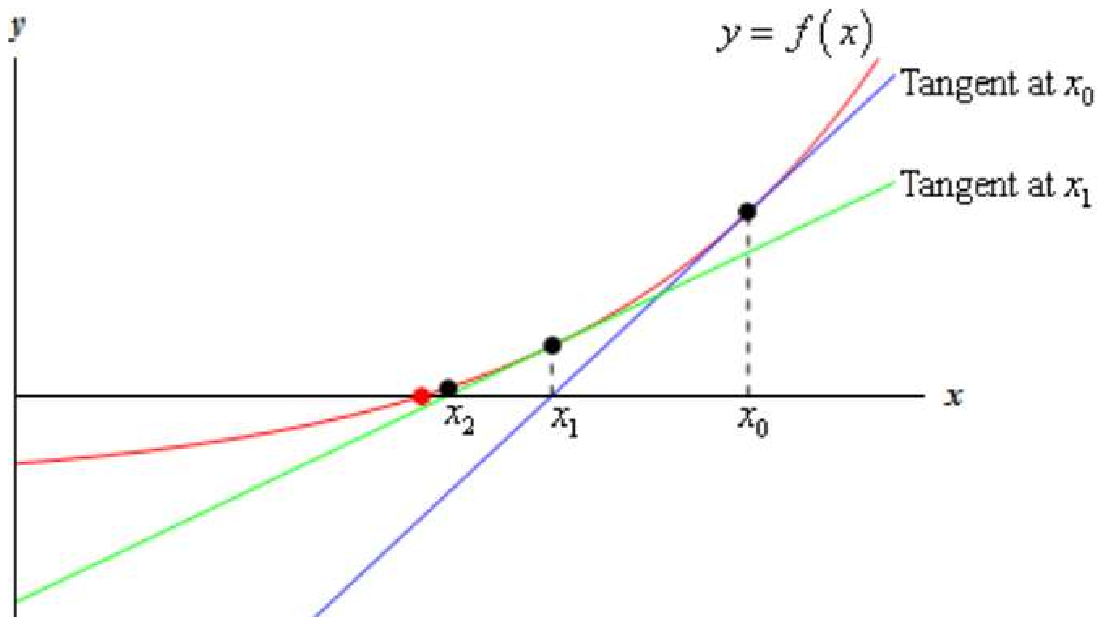
```
let bisection f a b =
    let eps = 1e-10 in
    let abs x = if x < 0. then -. x else x in
    let rec iter a b fa fb =
        let m = (a +. b) /. 2. in
        let fm = f m in
        if abs (a -. b) < eps then
            m
        else if fa *. fm < 0. then
            iter a m fa fm
        else
            iter m b fm fb in
    iter a b (f a) (f b)

let sqrt x =
    bisection (fun y -> y *. y -. x) 0. 10.

let sqrt2 = sqrt 2.

val sqrt2 : float = 1.41421356237868
```

Assignment 2

- Implement Newton's method for complex functions
  - Download newton.zip
  - Implement all TODOs in complex.ml, complex_arith.ml and newton.ml
  - Zip the three modified files and upload the single zip file to Brightspace
- Due date: 3/21/2024

Newton's Method

- Newton's method is a numerical method that can find a root of an equation as below
  - $x_{n+1} = x_n - f(x_n) / f'(x_n)$
  - i.e. $x_{n+1} = next(x_n)$: $x_1 = next(x_0)$, $x_2 = next(x_1)$, $x_3 = next(x_2)$, …



Newton's Method

- Fixed point of a function f: $\mathbb{R}\rightarrow\mathbb{R}$ is x such that f(x) = x
  - fixedPoint: $(\mathbb{R}\rightarrow\mathbb{R})\rightarrow\mathbb{R}$ is a function that returns the fixed point of f
  - Apply f to x_n until the difference between x_{n+1} and x_n becomes less than ε, where $x_{n+1} = f(x_n)$

- Given a function f: $\mathbb{R}\rightarrow\mathbb{R}$, next: $(\mathbb{R}\rightarrow\mathbb{R}) \rightarrow (\mathbb{R}\rightarrow\mathbb{R})$ is a function such that
  - (next f) x = x - (f x) / (f' x)

- Given a function f: $\mathbb{R}\rightarrow\mathbb{R}$, Newton's method finds a fixed point of next f
  - fixed_point (next f)

SUNY Korea

Program Overview

- App.ml will run the unit test cases

```
(*app.ml*)
#use "complex.ml"
#use "complex_arith.ml"
#use "newton.ml"

(*run the test cases*)
let _ = test_complex ()
let _ = test_polar ()

let _ = test_arith_complex ()
let _ = test_arith_polar ()

let _ = test_sqrt ()
let _ = test_poly ()
```

```
Expected output

> ocaml
…
# #use "app.ml";;
…
testing complex...
success.
- : unit = ()
…
testing arith...
success.
- : unit = ()
testing newton (sqrt -2)...
sa: 0.000000 + i 1.414214
sb: 1.414214 \_ 1.000000
success.
- : unit = ()
testing newton (solve x^2 + 1)...
ans: 0.000000 + i 1.000000
success.
- : unit = ()#
```

```ocaml
(*complex.ml*)

(*complex number in rectangular form*)
(*sel is one of "real", "imag", "mag", and "ang"*)
let complex r i = (*TODO: implement this function*)
    fun sel ->


(*complex number in polar form*)
(*sel is one of "real", "imag", "mag", and "ang"*)
let polar m a = (*TODO: implement this function*)
    fun sel ->


(*test*)
…
let test_complex () =
    Printf.printf "testing complex...\n";
…
let test_polar () =
    Printf.printf "testing polar...\n";
```

```
(*complex_arith.ml*)
#use "complex.ml"

(*arithmetic operations on complex numbers*)
(*opr is one of "add", "sub", "mul", and "div"*)
let rec complex_arith opr =
    let add a b = (*TODO: implement add in rectangular form: using real and imag*)

    let sub a b = (*TODO: implement sub in rectangular form: using real and imag*)

    let mul a b = (*TODO: implement mul in polar form: using mag and ang*)

    let div a b = (*TODO: implement div in polar form: using mag and ang*)

    (*TODO: return add, sub, mul or div depending on opr*)


(*test*)
…
let test_arith a b =
    Printf.printf "testing arith...\n";
…
let test_arith_complex () =
…
let test_arith_polar () =
```

```
(*newton.ml*)
#use "complex_arith.ml"

(*TODO: implement newton's method*)
let newton f x0 =
    let ( + ) = complex_arith "add" in
    let ( - ) = complex_arith "sub" in
    let ( / ) = complex_arith "div" in
    let eps   = 1e-8 in  (*epsilon: a small number*)
    let delta = complex eps eps in

    (*difference*)
    let diff a b =
        (a - b) "mag" in


    (*the derivative of f: (f(x + delta) - f(x)) / delta*)
    (*TODO: implement derivative*)
    let derivative f =
```

```
(*return a function that finds the next guess from the current guess*)
(*TODO: implement next*)
let next f =
    let dfdx = derivative f in (*f'(x)*)


(*fixed point of f is x such that x = f(x) *)
(*TODO: recursively apply f(x) to f until the difference
        between x and f(x) is less than eps*)
let rec fixed_point f x =


(*return the solution*)
(*TODO: find a fixed point of next f starting from x0*)
```

```
let complex_sqrt x =
    let ini = complex 1. 1. in
    let ( - ) = complex_arith "sub" in
    let ( * ) = complex_arith "mul" in
    newton (fun y -> y * y - x) ini

(*test*)
…
let test_sqrt () =
    Printf.printf "testing newton (sqrt -2)...\n";

let test_poly () =
    Printf.printf "testing newton (solve x^2 + 1)...\n";
```