Simple Data

COS 326
David Walker
Princeton University
What is the single most important mathematical concept ever developed in human history?
What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable
Why is the mathematical variable so important?

The mathematician says:

“Let x be some integer, we define a polynomial over x ...”
Why is the mathematical variable so important?

The mathematician says:

“Let x be some integer, we define a polynomial over x ...”

What is going on here? The mathematician has separated a definition (of x) from its use (in the polynomial). This is the most primitive kind of abstraction.

Abstraction is the key to controlling complexity and without it, modern mathematics, science, and computation would not exists.
O’CAML BASICS:
LET DECLARATIONS
• Good programmers identify repeated patterns in their code and factor out the repetition into meaning components

• In O’Caml, the most basic technique for factoring your code is to use let expressions

• Instead of writing this expression:

\[(2 + 3) \times (2 + 3)\]
Abstraction & Abbreviation

• Good programmers identify repeated patterns in their code and factor out the repetition into meaning components.

• In O’Caml, the most basic technique for factoring your code is to use let expressions.

• Instead of writing this expression:

\[(2 + 3) \times (2 + 3)\]

• We write this one:

```ocaml
let x = 2 + 3 in
x * x
```
A Few More Let Expressions

let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed
A Few More Let Expressions

let a = "a" in
let b = "b" in
let as = a ^ a ^ a in
let bs = b ^ b ^ b in
as ^ bs

let x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed

let two = 2.0 in
let zero = 0.0 in
two *. zero
How do let expressions operate?

```
let x = 2 + 1 in x * x
```
How do let expressions operate?

```plaintext
let x = 2 + 1 in x * x
```

--> 

```plaintext
let x = 3 in x * x
```
How do let expressions operate?

```
let x = 2 + 1 in x * x
```

--> 

```
let x = 3 in x * x
```

--> 

```
3 * 3
```

substitute 3 for x
How do let expressions operate?

\[
\text{let } x = 2 + 1 \text{ in } x \times x
\]

\[\rightarrow\]

\[
\text{let } x = 3 \text{ in } x \times x
\]

\[\rightarrow\]

\[
3 \times 3
\]

\[\rightarrow\]

\[
9
\]

substitute 3 for x
How do let expressions operate?

`let x = 2 + 1 in x * x`  

-->  

`let x = 3 in x * x`  

-->  

`3 * 3`  

-->  

`9`  

**Note:** I write $e_1 \rightarrow e_2$ when $e_1$ evaluates to $e_2$ in one step.
let x = 2 in
let y = x + x in
y * x
Another Example

let x = 2 in
let y = x + x in
y * x

-->

let y = 2 + 2 in
y * 2

substitute
2 for x
Another Example

let x = 2 in
let y = x + x in
y * x

substitute 2 for x

-->
let y = 2 + 2 in
y * 2

-->
let y = 4 in
y * 2
Another Example

```
let x = 2 in
let y = x + x in
y * x
```

- Substitute 2 for x:
  
  ```
  let y = 2 + 2 in
  y * 2
  ```

- Substitute 4 for y:
  
  ```
  let y = 4     in
  y * 2
  ```

```
4 * 2
```
Another Example

let \( x = 2 \) in
let \( y = x + x \) in
\( y \times x \)

\[
\begin{align*}
\text{substitute} & \quad 2 \text{ for } x \\
\text{let } y & \text{ = } 2 + 2 \text{ in} \\
& y \times 2 \\
\text{substitute} & \quad 4 \text{ for } y \\
\text{let } y & \text{ = } 4 \text{ in} \\
& y \times 2 \\
\text{--->} & \\
& 4 \times 2 \\
\text{--->} & \\
& 8
\end{align*}
\]

Moral: Let operates by substituting computed values for variables.
Abstraction & Abbreviation

• Two kinds of let:

```plaintext
if tuesday() then
  let x = 2 + 3 in
  x + x
else
  0
;;
```

`let ... in ...` is an expression that declares a local variable for temporary use and produce a value
Abstraction & Abbreviation

• Two kinds of let:

```plaintext
let ... in ... is an expression that can appear inside any other expression

The scope of x does not extend outside the enclosing “in”

let x = 2 + 3 ;;
let y = x + 17 / x ;;

let ... ;; is a top-level declaration that appears at the top-level only.

Variables x and y may be exported; used by other modules
```
Typing Simple Let Expressions

let x = e1 in e2

x granted type of e1 for use in e2

overall expression takes on the type of e2
Typing Simple Let Expressions

- The variable `x` is granted the type of `e1` for use in `e2`.
- The overall expression takes on the type of `e2`.

```
let x = e1 in
```

- `x` has type `int` for use inside the `let` body.
- The overall expression has type `string`.

```
let x = 3 + 4 in
string_of_int x
```
Defining functions

• Non-recursive functions:

```plaintext
let add_one (x:int) : int = 1 + x ;;
```
Defining functions

• Non-recursive functions:

let add_one (x:int) : int = 1 + x ;;

let keyword

expression that computes value produced by function

let add_one (x:int) : int = 1 + x ;;

function name

type of argument

argument name

type of result

let keyword

non-recursive functions:

function name

argument name

type of argument

type of result

expression that computes value produced by function
Defining functions

• Non-recursive functions:

```plaintext
let add_one (x:int) : int = 1 + x ;
```

Note: recursive functions begin with "let rec"
Defining functions

• Non-recursive functions:

```plaintext
let add_one (x:int) : int = 1 + x ;;
let add_two (x:int) : int = add_one (add_one x) ;;
```
Defining functions

• Non-recursive functions:

```ocaml
let add_one (x:int) : int = 1 + x ;;

let add_two (x:int) : int = add_one (add_one x) ;;
```

• With a local definition:

```ocaml
let add_two' (x:int) : int =
  let add_one x = 1 + x in
  add_one (add_one x)
;;
```

- local function definition
  - hidden from clients

I left off the types. O'Caml figures them out

Good style: types on top-level definitions
Types for Functions

• Some functions:

```plaintext
let add_one (x:int) : int = 1 + x ;;
let add_two (x:int) : int = add_one (add_one x) ;;
let add (x:int) (y:int) : int = x + y ;;
```

• Types for functions:

```
add_one : int -> int
add_two : int -> int
add : int -> int -> int
```
Rule for type-checking functions

General Rule:

If a function $f : T_1 \rightarrow T_2$ and an argument $e : T_1$ then $f e : T_2$

Example:

add_one : int -> int
3 + 4 : int
add_one (3 + 4) : int
Rule for type-checking functions

- Recall the type of `add`:

  **Definition:**

  ```
  let add (x:int) (y:int) : int = x + y
  ;;
  ```

  **Type:**

  ```
  add : int -> int -> int
  ```
Rule for type-checking functions

• Recall the type of add:

Definition:

```ml
let add (x:int) (y:int) : int =
  x + y
;;
```

Type:

```ml
add : int -> int -> int
```

Same as:

```ml
add : int -> (int -> int)
```
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f e : T_2 \)

Remember:

\( A \rightarrow B \rightarrow C \) is the same as \( A \rightarrow (B \rightarrow C) \)

Example:

\( add : \text{int} \rightarrow \text{int} \rightarrow \text{int} \)

\( 3 + 4 : \text{int} \)

\( add (3 + 4) : \text{???} \)
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f\ e : T_2 \)

Example:

\[
\text{add} : \text{int} \rightarrow (\text{int} \rightarrow \text{int})
\]

\[
3 + 4 : \text{int}
\]

\[
\text{add} \ (3 + 4) :
\]

Remember:

\[
A \rightarrow B \rightarrow C \\
is the same as \\
A \rightarrow (B \rightarrow C)
\]
Rule for type-checking functions

General Rule:

If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Remember:

\( A \rightarrow B \rightarrow C \)

is the same as

\( A \rightarrow (B \rightarrow C) \)

Example:

\( \text{add} : \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \)

\( 3 + 4 : \text{int} \)

\( \text{add} \ (3 + 4) : \text{int} \rightarrow \text{int} \)
Rule for type-checking functions

General Rule:
If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \ e : T_2 \)

Example:
\[
\text{add} : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 : \text{int} \\
\text{add} (3 + 4) : \text{int} \rightarrow \text{int} \\
(\text{add} (3 + 4)) 7 : \text{int}
\]

Remember:
A \( \rightarrow \) B \( \rightarrow \) C is the same as A \( \rightarrow \) (B \( \rightarrow \) C)
Rule for type-checking functions

General Rule:
If a function \( f : T_1 \rightarrow T_2 \) and an argument \( e : T_1 \) then \( f \; e : T_2 \)

Remember:
\( A \rightarrow B \rightarrow C \) is the same as \( A \rightarrow (B \rightarrow C) \)

Example:
\[
\text{add} : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 : \text{int} \\
\text{add} \; (3 + 4) : \text{int} \rightarrow \text{int} \\
\text{add} \; (3 + 4) \; 7 : \text{int}
\]
Example:

```ocaml
let munge (b:bool) (x:int) : ?? =
  if not b then
    string_of_int x
  else
    "hello"
;;

let y = 17;;

munge (y > 17) : ??

munge true (f (munge false 3)) : ??
  f : ??

munge true (g munge) : ??
  g : ??
```
Example:

```ocaml
let munge (b:bool) (x:int) : ?? =
  if not b then
    string_of_int x
  else
    "hello"
;;

let y = 17;;

munge (y > 17) : ??

munge true (f (munge false 3)) : ??
  f : string -> int

munge true (g munge) : ??
  g : (bool -> int -> string) -> int
```
One key thing to remember

• If you have a function \( f \) with a type like this:

\[
A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow F
\]

• Then each time you add an argument, you can get the type of the result by knocking off the first type in the series

\[
f a1 : B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a1 : A)\]
\[
f a1 a2 : C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a2 : B)\]
\[
f a1 a2 a3 : D \rightarrow E \rightarrow F \quad \text{(if } a3 : C)\]
\[
f a1 a2 a3 a4 a5 : F \quad \text{(if } a4 : D \text{ and } a5 : E)\]
Binding Variables to Values

- Each O'Caml variable is *bound* to 1 value
- The value to which a variable is bound to never changes!

```
let x = 3 ;;

let add_three (y:int) : int = y + x ;;
```
Binding Variables to Values

- Each O'Caml variable is *bound* to 1 value
- The value to which a variable is bound to never changes!

```ocaml
let x = 3 ;;
let add_three (y:int) : int = y + x ;;
```

*It does not matter what I write next. add_three will always add 3!*
Binding Variables to Values

• Each O'Caml variable is bound to 1 value
• The value a variable is bound to never changes!

```ocaml
let x = 3 ;;
let add_three (y:int) : int = y + x ;;
let x = 4 ;;
let add_four (y:int) : int = y + x ;;
```
• Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename one of them

```
let x = 3 ;;
let add_three (y:int) : int = y + x ;;
let zzz = 4 ;;
let add_four (y:int) : int = y + zzz ;;
let add_seven (y:int) : int =
    add_three (add_four y)
;;
```
• Each O'Caml variable is bound to 1 value
• O'Caml is a **statically scoped** language

```ocaml
let x = 3 ;;

let add_three (y:int) : int = y + x ;;

let x = 4 ;;

let add_four (y:int) : int = y + x ;;

let add_seven (y:int) : int =
    add_three (add_four y) ;;
```
OUR FIRST* COMPLEX DATA STRUCTURE!
THE TUPLE

* it is really our second complex data structure since functions are data structures too!
A tuple is a fixed, finite, ordered collection of values

Some examples with their types:

(1, 2) : int * int

("hello", 7 + 3, true) : string * int * bool

('a', ("hello", "goodbye")) : char * (string * string)
Tuples

- To use a tuple, we extract its components
- General case:

```latex
let (id1, id2, ..., idn) = e1 in e2
```

- An example:

```latex
let (x, y) = (2, 4) in x + x + y
```
• To use a tuple, we extract its components
• General case:

\[
\text{let } (id_1, id_2, \ldots, id_n) = e_1 \text{ in } e_2
\]

• An example:

\[
\begin{align*}
\text{let } (x, y) &= (2, 4) \text{ in } x + x + y \\
\text{ substitute!} \rightarrow \quad 2 + 2 + 4
\end{align*}
\]
• To use a tuple, we extract its components

• General case:

\[
\text{let } (id1, id2, \ldots, idn) = e1 \text{ in } e2
\]

• An example:

\[
\begin{align*}
\text{let } (x,y) &= (2,4) \text{ in } x + x + y \\
\rightarrow &\quad 2 + 2 + 4 \\
\rightarrow &\quad 8
\end{align*}
\]
Rules for Typing Tuples

if \( e_1 : t_1 \) and \( e_2 : t_2 \)
then \( (e_1, e_2) : t_1 \times t_2 \)
Rules for Typing Tuples

\[
\text{let } (x_1, x_2) = e_1 \text{ in } e_2
\]

if \(e_1 : t_1 \text{ and } e_2 : t_2\) then \((e_1, e_2) : t_1 \times t_2\)

if \(e_1 : t_1 \times t_2\) then
\(x_1 : t_1 \text{ and } x_2 : t_2\)
inside the expression \(e_2\)

overall expression takes on the type of \(e_2\)
Problem:
• A point is represented as a pair of floating point values.
• Write a function that takes in two points as arguments and returns the distance between them as a floating point number
Writing Functions Over Typed Data

• Steps to writing functions over typed data:
  1. Write down the function and argument names
  2. Write down argument and result types
  3. Write down some examples (in a comment)
  4. Deconstruct input data structures
     • the argument types may suggest how to do it
  5. Build new output values
     • the result type may suggest how you do it
  6. Clean up by identifying repeated patterns
     • define and reuse helper functions
     • your code should be elegant and easy to read
Distance between two points

a type abbreviation

type point = float * float

(x1, y1)

(x2, y2)
Distance between two points

```ocaml
type point = float * float

let distance (p1:point) (p2:point) : float =
    ;;
```

Write down function name, argument names, and types:

- **Function Name:** distance
- **Argument 1:** p1 : point
- **Argument 2:** p2 : point
- **Return Type:** float

Vector notation method for calculating the distance between two points:

Distance formula: \( \sqrt{(x_2-x_1)^2 + (y_2-y_1)^2} \)
Distance between two points

type point = float * float

(*) distance (0.0,0.0) (0.0,1.0) == 1.0
* distance (0.0,0.0) (1.0,1.0) == sqrt(1.0 + 1.0)
* from the picture:
* distance (x1,y1) (x2,y2) == sqrt(a^2 + b^2)
*)

let distance (p1:point) (p2:point) : float =
type point = float * float

let distance (p1:point) (p2:point) : float =

   let (x1,y1) = p1 in
   let (x2,y2) = p2 in
   ...

;;

deconstruct function inputs
type point = float * float

let distance (p1:point) (p2:point) : float =

let (x1,y1) = p1 in
let (x2,y2) = p2 in
sqrt ((x2 -. x1) *. (x2 -. x1) +. (y2 -. y1) *. (y2 -. y1))
;;

compute function results

notice operators on floats have a "." in them
type point = float * float

let distance (p1:point) (p2:point) : float =
    let square x = x *. x in
    let (x1,y1) = p1 in
    let (x2,y2) = p2 in
    sqrt (square (x2 -. x1)) +. square (y2 -. y1)

;;

define helper functions to avoid repeated code
type point = float * float

let distance (p1:point) (p2:point) : float =
  let square x = x *. x in
  let (x1,y1) = p1 in
  let (x2,y2) = p2 in
  sqrt (square (x2 -. x1) +. square (y2 -. y1)) ;;

let pt1 = (2.0,3.0);;
let pt2 = (0.0,1.0);;
let dist12 = distance pt1 pt2;;
SUMMARY:
BASIC FUNCTIONAL PROGRAMMING
Writing Functions Over Typed Data

• Steps to writing functions over typed data:
  1. Write down the function and argument names
  2. Write down argument and result types
  3. Write down some examples (in a comment)
  4. Deconstruct input data structures
     • the argument types may suggest how to do it
  5. Build new output values
     • the result type may suggest how you do it
  6. Clean up by identifying repeated patterns
     • define and reuse helper functions
     • your code should be elegant and easy to read
Writing Functions Over Typed Data

• Steps to writing functions over typed data:
  1. Write down the function and argument names
  2. Write down argument and result types
  3. Write down some examples (in a comment)
  4. **Deconstruct** input data structures
  5. **Build** new output values
  6. Clean up by identifying repeated patterns

• For tuples:
  – when the **input** has type $t_1 \times t_2$
    • use `let (x, y) = ...` to **deconstruct**
  – when the **output** has type $t_1 \times t_2$
    • use `(e1, e2)` to **construct**

• We will see this paradigm repeat itself over and over