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
What is the single most important mathematical concept ever developed in human history?

An answer: The mathematical variable

(runner up: natural numbers induction)
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 (x is some integer)

Abstraction is the key to controlling complexity and without it, modern mathematics, science, and computation would not exist.
OCAML BASICS:
LET DECLARATIONS
Abstraction

- Good programmers identify repeated patterns in their code and factor out the repetition into meaningful 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 \times 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 x = 2 in
let squared = x * x in
let cubed = x * squared in
squared * cubed

let a = "a" in
let b = "b" in
let as = a ^ a ^ a in
let bs = b ^ b ^ b in
as ^ bs
Abstraction & Abbreviation

- Two kinds of let:

```plaintext
let x = 2 + 3 in
x + x
```

```plaintext
let y = x + 17 / x ;;
```

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

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

- `let ... ;;` without “in” is a top-level *declaration*

- Variables x and y may be exported; used by other modules

(Don’t need ;; if another let comes next; do need it if expression next)
• Each OCaml 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 ;;
```
• Each OCaml variable is \textit{bound} to 1 value
• \textit{The value to which a variable is bound to never changes!}

\begin{verbatim}
let x = 3 ;;
let add_three (y:int) : int = y + x ;;
\end{verbatim}

\textit{It does not matter what I write next. add_three will always add 3!}
Binding Variables to Values

• Each OCaml 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 ;;
```

a distinct variable that "happens to be spelled the same"
• Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them

```ocaml
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) ;;
```

rename x to zzz if you want to, replacing its uses
Binding Variables to Values

- Each OCaml variable is bound to 1 value
- OCaml 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)
;;
```

we can use `add_three` without worrying about the second definition of `x`
How do let expressions operate?

```latex
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?

let x = 2 + 1 in x * x

-->

let x = 3 in x * x

-->

3 * 3

-->

9

substitute 3 for x
How do let expressions operate?

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

-->

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

-->

```plaintext
3 * 3
```

-->

```plaintext
9
```

Note: I write e₁ → e₂ when e₁ evaluates to e₂ in one step.
Did you see what I did there?
Did you see what I did there?

I defined the language in terms of itself:

```
let x = 2 in x + 3  --\>  2 + 3
```

I’m trying to train you to think at a high level of abstraction.

*I didn’t have to mention low-level abstractions like assembly code or registers or memory layout*
Another Example

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

\[ \Rightarrow \]

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

\[ \Rightarrow \]

```
let y = 4 in
y * 2
```

\[ \Rightarrow \]

```
4 * 2
```

substitute 2 for \( x \)

substitute 4 for \( y \)
Another Example

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

\(\Rightarrow\)

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

\(\Rightarrow\)

```
let y = 4 in
y * 2
```

\(\Rightarrow\)

```
4 * 2
```

\(\Rightarrow\)

```
8
```

Moral: Let operates by substituting computed values for variables.
OCAML BASICS: TYPE CHECKING AGAIN
There are simple rules that tell you what the type of an expression is.

Those rules compute a type for an expression based on the *types* of its subexpressions (and the types of the variables that are in scope).

You don’t have to know the details of how a subexpression is implemented to do type checking. You just need to know its type.

That’s what makes OCaml type checking *modular*.

We write “e : t” to say that expression e has type t
if e : int
then string_of_int e : string
if e : int
then string_of_int e : string

if e1 : bool
and e2 : t and e3 : t (the same type t, for some type t)
then if e1 then e2 else e3 : t (that same type t)
Type Checking Rules

• Violating the rules:

```ocaml
# "hello" + 1;;
Error: This expression has type string but an expression was expected of type int
```

• The type error message tells you the type that was expected and the type that it inferred for your subexpression.

• By the way, this was one of the nonsensical expressions that did not evaluate to a value.

• I consider it a good thing that this expression does not type check.
Type Checking Rules

• Violating the rules:

```plaintext
# "hello" + 1;;
Error: This expression has type string but an expression was expected of type int
```

• A possible fix:

```plaintext
# "hello" ^ (string_of_int 1);;
- : string = "hello1"
```

• One of the keys to becoming a good ML programmer is to understand type error messages.
Typing Simple Let Expressions

- **x granted type of e1 for use in e2**
- **let x = e1 in e2**
- **overall expression takes on the type of e2**
Typing Simple Let Expressions

x granted type of e1 for use in e2

\[
\text{let } x = e_1 \text{ in } e_2
\]

overall expression takes on the type of e2

x has type int for use inside the let body

\[
\text{let } x = 3 + 4 \text{ in } \text{string_of_int } x
\]

overall expression has type string
let add_one (x:int) : int = 1 + x ;;
let

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

**Defining functions**

- `let` keyword
- Function name: `add_one`
- Argument name: `x`
- Type of argument: `int`
- Type of result: `int`
- Expression that computes value produced by function: `1 + x`

**Note:** recursive functions with `begin` begin with "let rec"
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) ;;
```

definition of add_one must come before use
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) 
  ;;
  ```
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:

```plaintext
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:

\[
\begin{align*}
\text{add\_one} : \text{int} & \rightarrow \text{int} \\
3 + 4 & : \text{int} \\
\text{add\_one} \ (3 + 4) & : \text{int}
\end{align*}
\]
• Recall the type of add:

**Definition:**

```ml
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:**

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

  **Type:**

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

  **Same as:**

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

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

Example:
\[
\text{add} : \text{int} \to \text{int} \to \text{int}
\]
\[
3 + 4 : \text{int}
\]
\[
\text{add} \, (3 + 4) : ???
\]

Note:
\( A \to B \to C \) is the same as \( A \to (B \to C) \)
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:

\begin{align*}
\text{add} &: \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \\
3 + 4 &: \text{int} \\
\text{add} (3 + 4) &: 
\end{align*}
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:

\[
\begin{align*}
\text{add} &: \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \\
3 + 4 &: \text{int} \\
\text{add} (3 + 4) &: \text{int} \rightarrow \text{int}
\end{align*}
\]

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:

add : int $\rightarrow$ int $\rightarrow$ int

$3 + 4 : \text{int}$

add $(3 + 4) : \text{int} \rightarrow \text{int}$

$(\text{add} \, (3 + 4)) \, 7 : \text{int}$
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:

\[
\begin{align*}
\text{add} & : \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
3 + 4 & : \text{int} \\
\text{add} \ (3 + 4) & : \text{int} \rightarrow \text{int} \\
\text{add} \ (3 + 4) \ 7 & : \text{int}
\end{align*}
\]

Remember:

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

\[
f \ a1 : B \rightarrow C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a1 : A) \\
\]

\[
f \ a1 \ a2 : C \rightarrow D \rightarrow E \rightarrow F \quad \text{(if } a2 : B) \\
\]

\[
f \ a1 \ a2 \ a3 : D \rightarrow E \rightarrow F \quad \text{(if } a3 : C) \\
\]

\[
f \ a1 \ a2 \ a3 \ a4 \ a5 : F \quad \text{(if } a4 : D \text{ and } a5 : E) \\
\]
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)
• To use a tuple, we extract its components
• General case:

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

• An example:

```haskell
let (x, y) = (2, 4) in x + x + y
```
To use a tuple, we extract its components

General case:

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

An example:

let (x, y) = (2, 4) in x + x + y
--> 2 + 2 + 4
Tuples

- 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:

\[
\text{let } (x,y) = (2,4) \text{ in } x + x + y \\
\rightarrow 2 + 2 + 4 \\
\rightarrow 8
\]
if $e_1 : t_1$ and $e_2 : t_2$
then $(e_1, e_2) : t_1 \ast t_2$
Rules for Typing Tuples

- If \( e_1 : t_1 \) and \( e_2 : t_2 \)
  then \( (e_1, e_2) : t_1 \times t_2 \)

- If \( e_1 : t_1 \times t_2 \) then
  - \( x_1 : t_1 \) and \( x_2 : t_2 \)
  - inside the expression \( e_2 \)

- Overall expression takes on the type of \( e_2 \)

let \( (x_1, x_2) = e_1 \) in
\[ e_2 \]
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)
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 suggest how to do it*
5. **Build** new output values
   - *the result type suggests how you do it*
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 suggests how to do it
5. Build new output values
   - the result type suggests 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
   - *the argument types suggests how to do it*
5. **Build** new output values
   - *the result type suggests 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

*Types help structure your thinking about how to write programs.*
Distance between two points

A type abbreviation

type point = float * float

Diagram showing two points, (x1, y1) and (x2, y2), with the distance between them labeled as c.
type point = float * float

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

write down function name
argument names and types
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));

;:

notice operators on floats have a "." in them

compute function results

Distance between two points

(x1, y1)

a

(x2, y2)
b

c
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;;
MORE TUPLES
Here's a tuple with 2 fields:

\[(4.0, 5.0) : \text{float} \times \text{float}\]
Tuples

• Here's a tuple with 2 fields:

  (4.0, 5.0) : float * float

• Here's a tuple with 3 fields:

  (4.0, 5, "hello") : float * int * string
Tuples

• Here's a tuple with 2 fields:

  (4.0, 5.0) : float * float

• Here's a tuple with 3 fields:

  (4.0, 5, "hello") : float * int * string

• Here's a tuple with 4 fields:

  (4.0, 5, "hello", 55) : float * int * string * int
Tuples

• Here's a tuple with 2 fields:

  (4.0, 5.0) : float * float

• Here's a tuple with 3 fields:

  (4.0, 5, "hello") : float * int * string

• Here's a tuple with 4 fields:

  (4.0, 5, "hello", 55) : float * int * string * int

• Have you ever thought about what a tuple with 0 fields might look like?
Unit

• **Unit** is the tuple with zero fields!

\[ () : \text{unit} \]

• the unit value is written with an pair of parens
• there are no other values with this type!
• **Unit** is the tuple with zero fields!

```
() : unit
```

• the unit value is written with an pair of parens
• there are no other values with this type!

• Why is the unit type and value useful?
• Every expression has a type:

```
(print_string "hello world\n") : ???
```
Unit

- **Unit** is the tuple with zero fields!
  
  ```
  () : unit
  ```

  - the unit value is written with an pair of parens
  - there are no other values with this type!

- Why is the unit type and value useful?
- Every expression has a type:
  
  ```
  (print_string "hello world\n") : unit
  ```

- Expressions executed for their **effect** return the unit value
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
  5. **Build** new output values
  6. Clean up by identifying repeated patterns

• For unit type:
  – when the **input** has type unit
    • use `let () = ... in ...` to deconstruct
    • or better use `e1; ...` to deconstruct if `e1` has type unit
  – when the **output** has type unit
    • use `()` to **construct**
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* suggest how to do it
5. **Build** new output values
   - the *result type* 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 tuple types:

- 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
END