Simple Functional Data

COS 326
David Walker
Princeton University

TYPE ERRORS

Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

ocamlbuild says:

```
Error: This expression has type int but an
expression was expected of type string
```

Type Checking Rules

Type errors for if statements can be confusing sometimes. Recall:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

ocamlbuild says:

```
Error: This expression has type int but an
expression was expected of type string
```

merlin inside emacs points to the error above and gives a second error:

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

Type errors for if statements can be confusing sometimes.

Example. We create a string from s, concatenating it n times:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

???

ocamlbuild says:

```
Error: This expression has type int but an
expression was expected of type string
```

merlin inside emacs points to the error above and gives a second error:

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

Type errors for if statements can be confusing sometimes.

Example. We create a string from s, concatenating it n times:

they don't *agree*!

```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

???

ocamlbuild says:

```
Error: This expression has type int but an
expression was expected of type string
```

merlin inside emacs points to the error above and gives a second error:

```
Error: This expression has type string but an
expression was expected of type int
```

Type Checking Rules

Type errors for if statements can be confusing sometimes.
Example. We create a string from s, concatenating it n times:

they don't *agree*!

```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

???

The type checker points to *some* place where there is *disagreement*.

Moral: *Sometimes you need to look in an earlier branch for the error* even though the type checker points to a later branch.
The type checker doesn't know what the user wants.

A Tactic: Add Typing Annotations

```
let rec concatn (s:string) (n:int) : string =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

```
Error: This expression has type int but an
expression was expected of type string
```

ONWARD

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.

It allows *reuse* of ideas, theorems ... functions and programs!

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) * (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) * (2 + 3)
```

- We write this one:

```
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 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:

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

```
let x = 2 + 3

let y = x + 17 / x
```

let … in … is an *expression* that can appear inside any other *expression*

The scope of x (ie: the places x may be used) 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

You can only omit the "in" in a top-level declaration

Binding Variables to Values

During execution, we say an OCaml variable is *bound* to a 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

During execution, we say an OCaml variable is *bound* to a value.

The value to which a variable is bound to never changes!

```
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

During execution, we say an OCaml variable is *bound* to a value.

The value to which a variable is bound to never changes!

a distinct variable that "happens to be spelled the same"

```
let x = 3

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

let x = 4

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

Binding Variables to Values

Since the 2 variables (both happened to be named x) are actually different, unconnected things, we can rename them

rename x to zzz if you want to, replacing its uses

```
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)
```

Binding Variables to Values

A use of a variable always refers to it's *closest* (in terms of syntactic distance) enclosing declaration. Hence, we say OCaml is a *statically scoped* (or *lexically scoped*) language

we can use add_three without worrying about the second definition of x

```
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)
```

How does OCaml execute a let expression?

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

How does OCaml execute a let expression?

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

-->

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

How does OCaml execute a let expression?

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

-->

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

substitute
3 for x

-->

```
          3 * 3
```

How does OCaml execute a let expression?

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

-->

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

substitute
3 for x

-->

```
          3 * 3
```

-->

```
          9
```

How does OCaml execute a let expression?

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

-->

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

substitute
3 for x

-->

```
3 * 3
```

-->

```
9
```

Note: I write
e1 --> e2
when e1 evaluates
to e2 in one step

Meta-comment

OCaml expression

OCaml expression

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

I defined the language in terms of itself:
By reduction of one OCaml expression to another

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 to tell you how OCaml works.

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
```

substitute
2 for x

-->

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

-->

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

substitute
4 for y

-->

```
4 * 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
```

substitute
4 for y

-->

```
4 * 2
```

-->

```
8
```

Moral: Let operates by *substituting* computed values for variables

OCAML BASICS:
TYPE CHECKING AGAIN

Back to Let Expressions ... Typing

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression takes on the type of e2

Back to Let Expressions ... Typing

x granted type of e1 for use in e2

```
let x = e1 in

e2
```

overall expression takes on the type of e2

x has type int for use inside the let body

```
let x = 3 + 4 in

string_of_int x
```

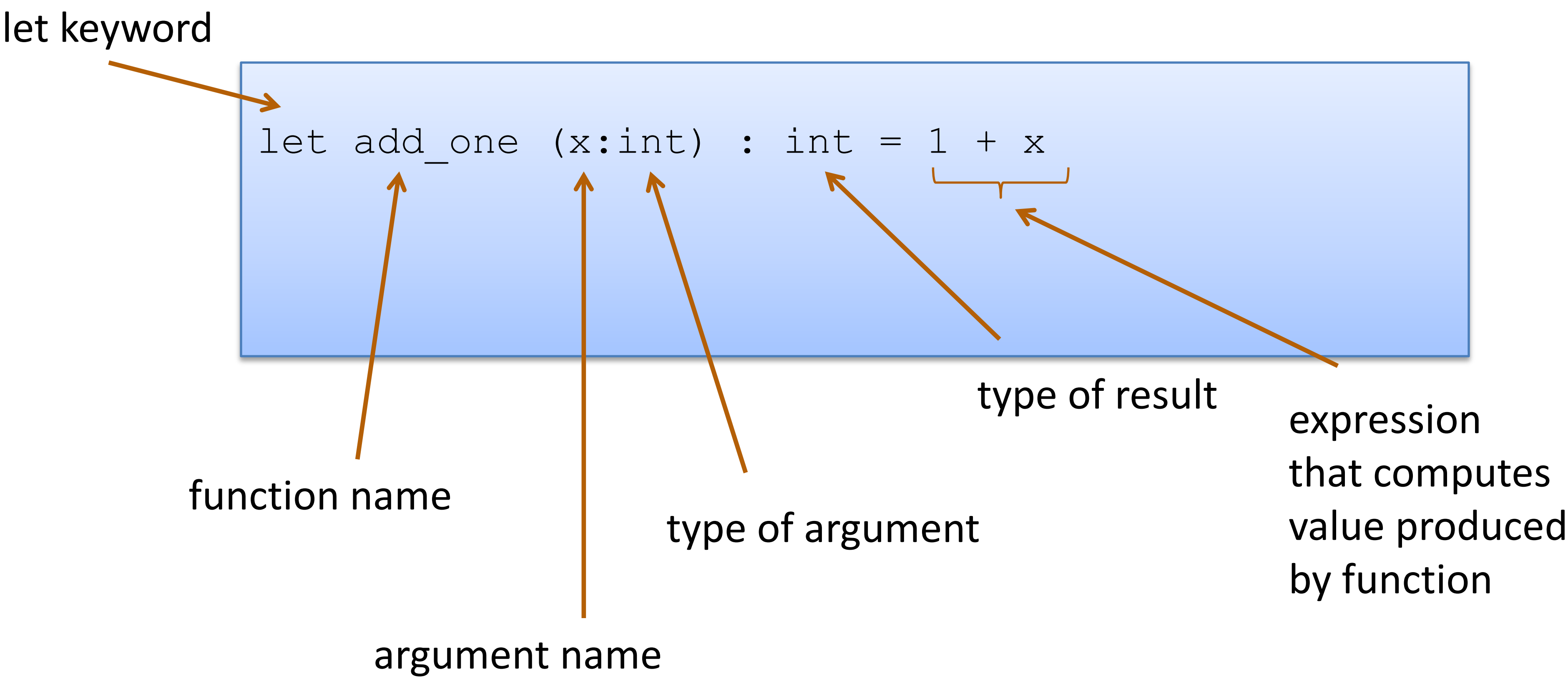
overall expression has type string

OCAML BASICS: FUNCTIONS

Defining functions

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

Defining functions



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

Defining functions

Nonrecursive functions:

```
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

Nonrecursive functions:

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

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

With a local definition:

```
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:

```
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
```

function with two arguments

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 : T1 -> T2
and an argument e : T1
then f e : T2

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:

```
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 : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

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

3 + 4 : int

add (3 + 4) : ???
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) :
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

```
add : int -> (int -> int)

3 + 4 : int

add (3 + 4) : int -> int
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

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

3 + 4 : int

add (3 + 4) : int -> int

(add (3 + 4)) 7 : int
```

Rule for type-checking functions

General Rule:

If a function f : T1 -> T2
and an argument e : T1
then f e : T2

A -> B -> C

same as:

A -> (B -> C)

Example:

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

3 + 4 : int

add (3 + 4) : int -> int

add (3 + 4) 7 : int
```

extra parens not necessary

Rule for type-checking functions

Example:

```
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 : ??
```

Rule for type-checking functions

Example:

```
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 -> B -> C -> D -> E -> 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 -> C -> D -> E -> F (if a1 : A)

f a1 a2 : C -> D -> E -> F (if a2 : B)

f a1 a2 a3 : D -> E -> F (if a3 : C)

f a1 a2 a3 a4 a5 : F (if a4 : D and a5 : E)

OUR FIRST* COMPLEX DATA STRUCTURE! THE TUPLE

* it is really our second complex data structure since functions are data structures too!

Tuples

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:

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

An example:

```
let (x,y) = (2,4) in x + x + y
```

Tuples

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
```

substitute!

Tuples

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
-->  8
```

Rules for Typing Tuples

if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2

Rules for Typing Tuples

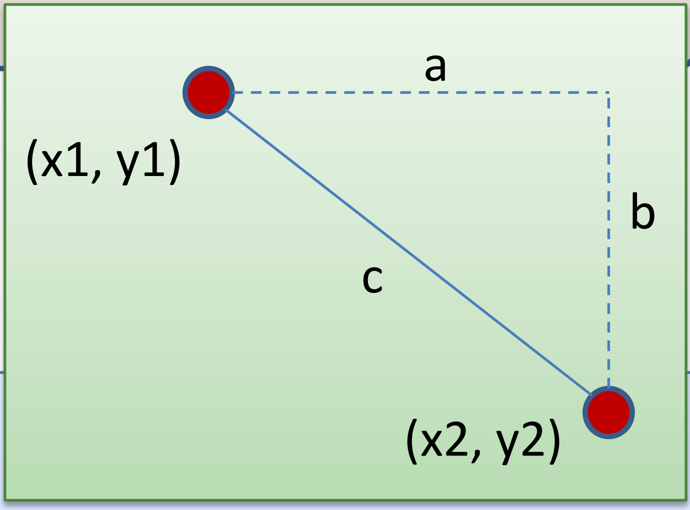
if e1 : t1 and e2 : t2
then (e1, e2) : t1 * t2

if e1 : t1 * t2 then
x1 : t1 and x2 : t2
inside the expression e2

```
let (x1,x2) = e1 in

e2
```

overall expression
takes on the type of e2

Distance between two points



$$c^2 = a^2 + b^2$$

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)

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*

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

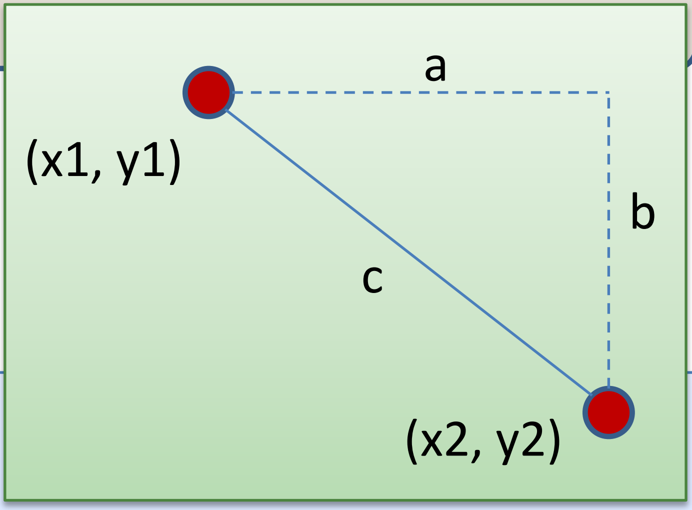
Writing Functions Over Typed Data

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.

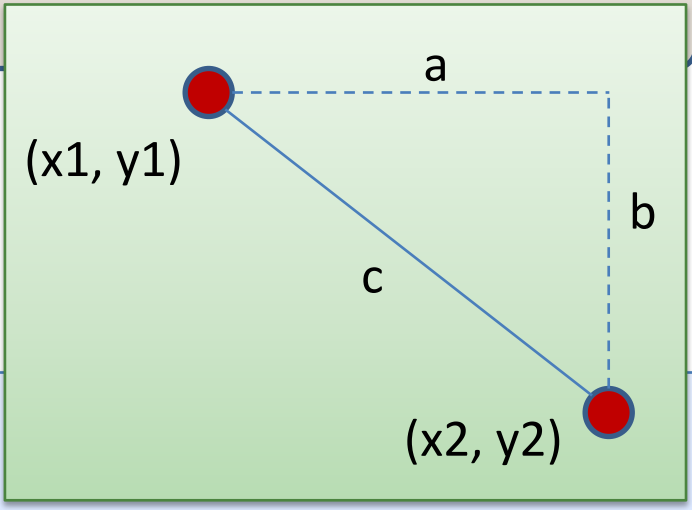
Distance between two points



a type abbreviation

```
type point = float * float
```

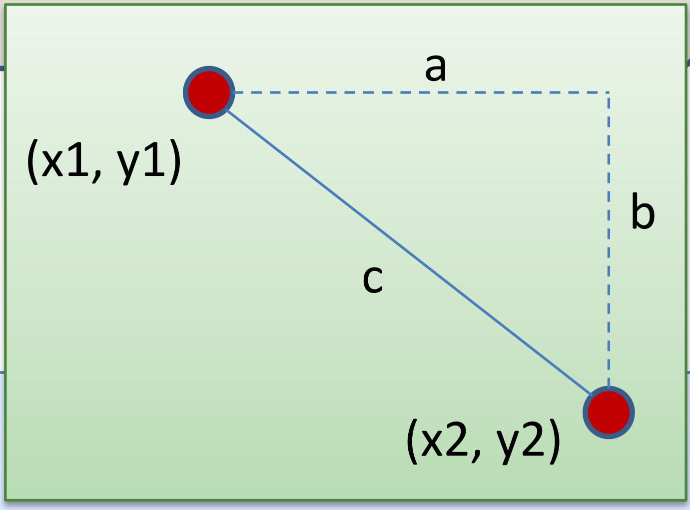
Distance between two points




```
type point = float * float

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

write down function name
argument names and types

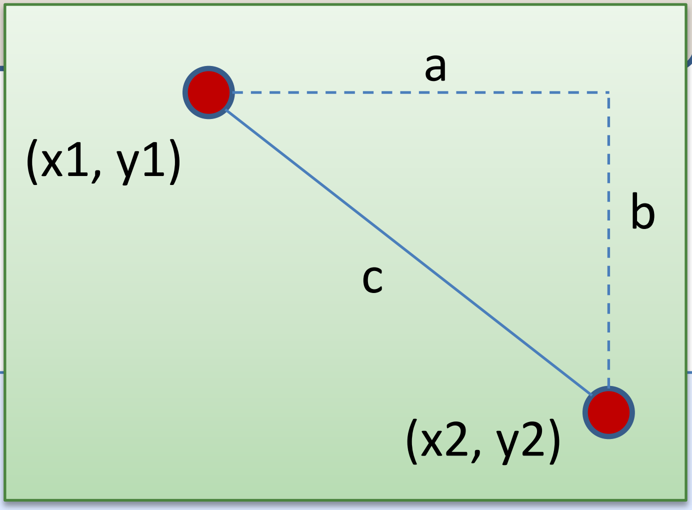
Distance between two points



examples

```
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 =
```

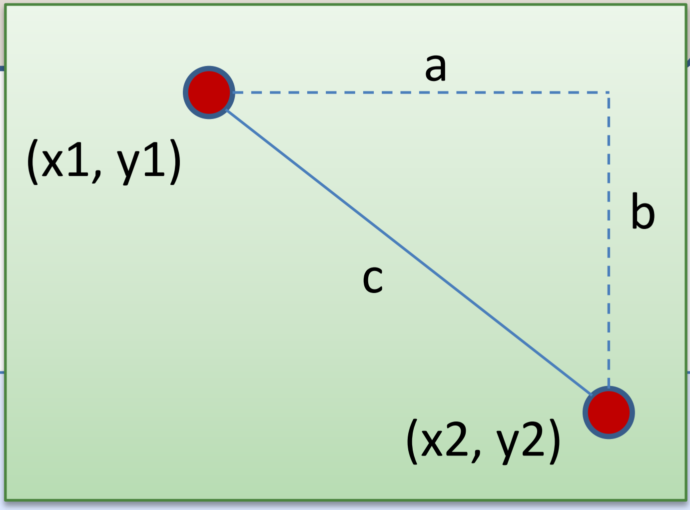
Distance between two points




```
type point = float * float

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

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

deconstruct
function inputs

Distance between two points

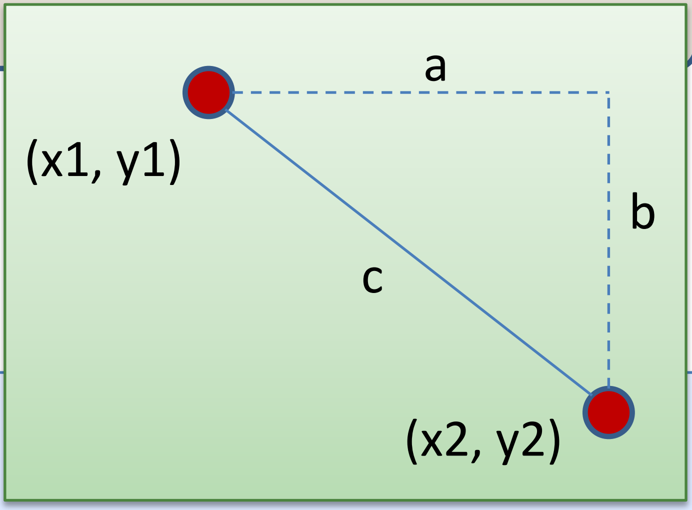



```
type point = float * float

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

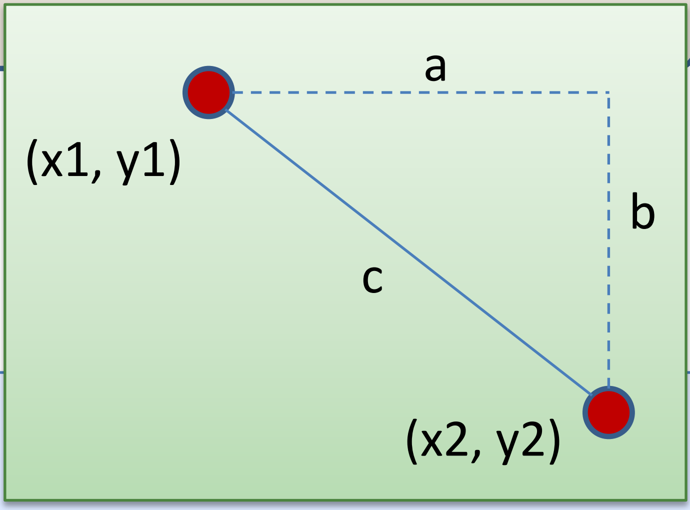
Distance between two points




```
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

Distance between two points




```
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
```

testing

MORE TUPLES

Tuples

Here's a tuple with 2 fields:

(4.0, 5.0) : float * 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

Here's a tuple with 0 fields:

() : unit

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

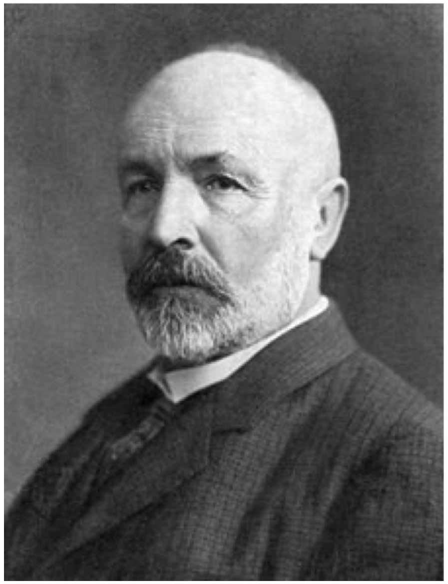
For tuple types:

- when the input has type t1 * t2
  - use let (x,y) = … to deconstruct
- when the output has type t1 * t2
  - use (e1, e2) to construct

We will see this paradigm repeat itself over and over

WHERE DID TYPE SYSTEMS COME FROM?

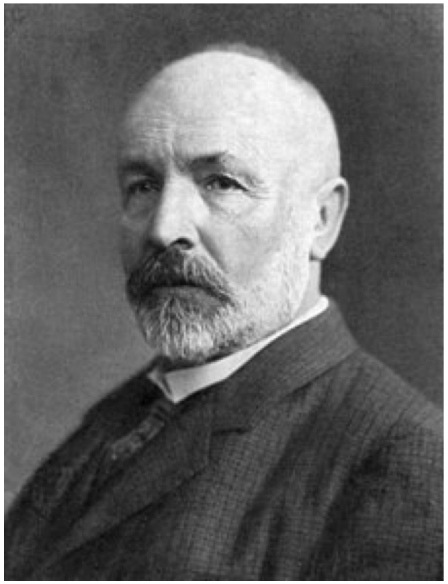
Origins of Type Theory



Georg Cantor

* http://www.math.ups.edu/~bryans/Current/Journal_Spring_1999/JEarly_232_S99.html

Origins of Type Theory



Georg Cantor

Über eine Eigenshaft des Inbegriffes aller reellen algebraischen Zahlen. 1874

(On a Property of the System of all the Real Algebraic Numbers)

"Considered the first purely theoretical paper on set theory." *

* http://www.math.ups.edu/~bryans/Current/Journal_Spring_1999/JEarly_232_S99.html

Origins of Type Theory

Bertrand Russell

Origins of Type Theory

Bertrand Russell

He noticed that Cantor's set theory allows the definition of this set S:

$\{ A \mid A \text{ is a set and } A \notin A \}$

Origins of Type Theory

Bertrand Russell

He noticed that Cantor's set theory allows the definition of this set S:

{ A | A is a set and A ∉ A }

If we assume S is not in the set S, then by definition, it must belong to that set.

If we assume S is in the set S, then it contradicts the definition of S.

Russell's paradox

Origins of Type Theory

Bertrand Russell

He noticed that Cantor's set theory allows the definition of this set S:

$\{ A \mid A \text{ is a set and } A \notin A \}$

Russell's solution:

Each set has a distinct type:
type 1, 2, 3, 4, 5, ...

A set of type i+1 can only have elements of type i so it can't include itself.

Aside

Ernst Zermelo

Abraham Fraenkel

Developers of Fraekel-Zermelo set theory.
An alternative solution to Russell's paradox.

Fast Forward to the 70s

Robin Milner

In 1978, developed ML
and coined the phrase

"*well-typed programs don't go wrong*"

to describe a key property
of type-safe languages

Well-typed Programs Don’t Go Wrong

Some ML programs do not have a well-defined semantics:

```
“hello” + 3
```

Such programs do not type check.

Well-typed Programs Don't Go Wrong

Some ML programs do not have a well-defined semantics:

```
"hello" + 3
```

Such programs do not type check.

Moreover, when we execute a well-typed program, *we are guaranteed to never, ever run into such a program during execution.*

```
let x = "hello" in
let y = 3 in
x + y
```

Well-typed Programs Don't Go Wrong

Some ML programs do not have a well-defined semantics:

```
"hello" + 3
```

Such programs do not type check.

Moreover, when we execute a well-typed program, *we are guaranteed to never, ever run into such a program during execution.*

```
let x = "hello" in
let y = 3 in
x + y
```

-- >*

```
"hello" + 3
```

well-typed programs don't reduce to programs like "hello" + 3, which go wrong

Well-type programs don't go wrong

What about this expression:

```
3 / 0
```

Well-type programs don’t go wrong

What about this expression:

```
3 / 0
```

It type checks. When executed, ML will supply this message:

```
Exception: Division_by_zero.
```

Did the expression “go wrong”?

Did it violate our credo “well-typed expressions don’t go wrong?”

Well-type programs don’t go wrong

What about this expression:

```
3 / 0
```

It type checks. When executed, ML will supply this message:

```
Exception: Division_by_zero.
```

Did the expression “go wrong”?

Did it violate our credo “well-typed expressions don’t go wrong?”

No and No. Exceptions are a well-defined result of a computation.

ie: you can look up what happens to 3 / 0 in the OCaml manual.

Discussion Topics

What's the difference between raising an exception and "going wrong"?

Why distinguish between these things?

Does one have to treat "hello" + 3 as "going wrong"?

Why does OCaml make such choices?

Is it reasonable for other languages to choose differently?

Type Soundness

"Well typed programs do not go wrong"

Programming languages with this property have *sound* type systems. They are called *safe* languages.

Safe languages are generally *immune* to buffer overrun vulnerabilities, uninitialized pointer vulnerabilities, etc., etc. (but not immune to all bugs!)

Safe languages: ML, Java, Python, …

Unsafe languages: C, C++, Pascal

Well typed programs do not go wrong

Robin Milner

Turing Award, 1991

"For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics."

"Well typed programs do not go wrong"

Robin Milner, 1978