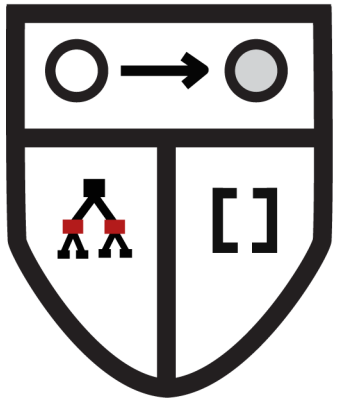

CS 3110

User-defined Data Types

Prof. Clarkson

Fall 2015

Today's music: *Pokémon Theme* by Jason Paige

Question

How much progress have you made on A1?

A. I'm still figuring out how Enigma works.
B. My code can cipher single letters.
C. My code can cipher multiple letters, but stepping is still iffy.
D. I'm done with **`cipher`** and **`simulate`**.
E. I've finished the scavenger hunt, too.

Submission of A1

- Please **have fun** and enjoy building the Engima

- Do **use the automatic extension** from the soft deadline to the hard deadline, if it will help you
- Do **submit earlier than the deadline** (11:59 pm)
- Do be aware that there is **no CMS grace period**

- Please don't try to submit by email

Review

Previously in 3110:

- **Functions**:
  - writing them, binding variables in them,
  - recursive, anonymous, higher-order
  - map and fold

Today:

- Turn attention to **data**
- Ways to define your own data types: records, tuples, variants

RECORDS

Record definition

- A **record** contains several named **fields**
- Before you can use a record, must **define** a record type:

```
type time = {hour: int; min: int; ampm: string}
```

- To *build* a record:
  - Write a record expression:
    `{hour=10; min=10; ampm="am"}`
  - Order of fields doesn't matter:
    `{min=10; hour=10; ampm="am"}` is equivalent

- To *access* record's field: `r.hour`

Record expressions

- **Syntax:** `{f1 = e1; …; fn = en}`

- **Evaluation:**
  - If `e1` evaluates to `v1`, and … `en` evaluates to `vn`
  - Then `{f1 = e1; …; fn = en}` evaluates to `{f1 = v1, …, fn = vn}`
  - Result is a *record value*

- **Type-checking:**
  - If `e1 : t1` and `e2 : t2` and … `en : tn`,
  - and if `t` is a defined type of the form `{f1:t1, …, fn:tn}`
  - then `{f1 = e1; …; fn = en} : t`

Record field access

- **Syntax:** `e.f`
- **Evaluation:**
  - If `e` evaluates to `{f = v, ...}`
  - Then `e.f` evaluates to `v`
- **Type-checking:**
  - If `e : t1`
  - and if `t1` is a defined type of the form `{f : t2, ...}`
  - then `e.f : t2`

Evaluation notation

We keep writing statements like:
If `e` evaluates to `{f = v, ...}` then `e.f` evaluates to `v`

Let's introduce a shorthand notation:

- Instead of "`e` evaluates to `v`"
- write "`e ==> v`"

So we can now write:
If `e ==> {f = v, ...}` then `e.f ==> v`

By name vs. by position

- Fields of record are identified **by name**
  - order we write fields in expression is irrelevant
- Opposite choice: identify **by position**
  - e.g., "Would the student named NN. step forward?" vs. "Would the student in seat *n* step forward?"
- You're accustomed to both:
  - Java object fields accessed by name
  - Java method arguments passed by position (but accessed in method body by name)
- OCaml has something you might not have seen:
  - A kind of heterogeneous data accessed by position

PAIRS AND TUPLES

Pairs

A **pair** of data: two pieces of data glued together

e.g.,

- `(1,2)`
- `(true, "Hello")`
- `([1;2;3], 0.5)`

We need language constructs to *build* pairs and to *access* the pieces...

Pairs: building

- Syntax: `(e1,e2)`

- Evaluation:
  - If `e1 ==> v1` and `e2 ==> v2`
  - Then `(e1,e2) ==> (v1,v2)`
  - A pair of values is itself a value

- Type-checking:
  - If `e1 : t1` and `e2 : t2`,
  - then `(e1,e2) : t1*t2`
  - A new kind of type, the **product type**

Pairs: accessing

- **Syntax**: `fst e` and `snd e`
  Projection functions

- **Evaluation**:
  - If `e ==> (v1,v2)`
  - then `fst e ==> v1`
  - and `snd e ==> v2`

- **Type-checking**:
  - If `e: ta*tb`,
  - then `fst e` has type `ta`
  - and `snd e` has type `tb`

Tuples

Actually, you can have *tuples* with more than two parts

- A new feature: a generalization of pairs
- Syntax, semantics are straightforward, except for projection...

- `(e1,e2,…,en)`
- `t1 * t2 * … * tn`
- `fst e, snd e, ???`

Instead of generalizing projection functions,
use pattern matching...

New kind of pattern, the **tuple pattern**: `(p1, ..., pn)`

Pattern matching tuples

```
match (1,2,3) with
| (x,y,z) -> x+y+z

(* ==> 6 *)

let thrd t =
  match t with
  | (x,y,z) -> z

(* thrd : 'a*'b*'c -> 'c *)
```

Note: we never needed more than one branch in the match expression...

Pattern matching without match

```
(* OK *)
let thrd t =
  match t with
  | (x,y,z) -> z

(* good *)
let thrd t =
  let (x,y,z) = t in z

(* better *)
let thrd t =
  let (_,_,z) = t in z

(* best *)
let thrd (_,_,z) = z
```

Extended syntax for let

- Previously we had this syntax:
  - `let x = e1 in e2`
  - `let [rec] f x1 ... xn = e1 in e2`

- Everywhere we had a variable identifier x, we can really use a pattern!
  - `let p = e1 in e2`
  - `let [rec] f p1 ... pn = e1 in e2`

- Old syntax is just a special case of new syntax, since a variable identifier is a pattern

Pattern matching arguments

```
(* OK *)
let sum_triple t =
   let (x,y,z) = t
   in x+y+z

(* better *)
let sum_triple (x,y,z) = x+y+z
```

Note how that last version looks syntactically like a function in C/Java!

Question

What is the type of this expression?

```
let (x,y) = snd("zar",("doz",42))
in (42,y)
```

A. {x:string; y:int}

B. int*int

C. string*int

D. int*string

E. string*(string*int)

Question

What is the type of this expression?

```
let (x,y) = snd("zar",("doz",42))
in (42,y)
```

A. `{x:string; y:int}`

B. `int*int`

C. `string*int`

D. `int*string`

E. `string*(string*int)`

Unit

- Can actually have a tuple `()` with no components whatsoever
  - Think of it as a degenerate tuple
  - Or, like a Boolean that can only have one value
- "Unit" is
  - a value written `()`
  - and a type written **`unit`**
- We've seen this already with printing functions

Pattern matching records

```
(* OK *)
let get_hour t =
  match t with
   | {hour=h; min=m; ampm=s} -> h

(* better *)
let get_hour t =
  match t with
   | {hour=h; min=_; ampm=_} -> h

(* better *)
let get_hour t =
  match t with
   | {hour; min; ampm} -> hour
```

```
(* better *)
let get_hour t =
  match t with
   | {hour} -> hour

(* better *)
let get_hour t =
  let {hour} = t in hour

(* better *)
let get_hour {hour} = hour

(* best *)
let get_hour t = t.hour
```

New kind of pattern, the **record pattern**:

`{f1[=p1]; ...; fn[=pn]}`

By name vs. by position, again

How to choose between coding `(4,7,9)` and `{f=4;g=7;h=9}`?

- Tuples are syntactically shorter
- Records are self-documenting
- For many (3? 4? 5?) fields, a record is usually a better choice

VARIANTS

Variant

```
type day = Sun | Mon | Tue | Wed
         | Thu | Fri | Sat

let day_to_int d =
    match d with
     | Sun -> 1
     | Mon -> 2
     | Tue -> 3
     | Wed -> 4
     | Thu -> 5
     | Fri -> 6
     | Sat -> 7
```

Building and accessing variants

Syntax: `type t = C1 | ... | Cn`

the `Ci` are called *constructors*

Evaluation: a constructor is already a value

Type checking: `Ci : t`

Accessing: use pattern matching; constructor name is a pattern

Pokémon variant



DEFENSE → ATTACK ↓	NOR	FIR	WAT	E
NORMAL				
FIRE		½	½	
WATER		2	½	

Pokémon variant



```
type ptype = TNormal | TFire | TWater

type peff = ENormal | ENotVery | ESuper

let eff_to_float = function
   | ENormal  -> 1.0
   | ENotVery -> 0.5
   | ESuper   -> 2.0

let eff_att_vs_def : ptype*ptype -> peff = function
   | (TFire,TFire)   -> ENotVery
   | (TWater,TWater) -> ENotVery
   | (TFire,TWater)  -> ENotVery
   | (TWater,TFire)  -> ESuper
   | _ -> ENormal
```

Argument order: records

If you are worried about clients of function forgetting which order to pass arguments in tuple, use a record:

```
type att_def = {att:ptype; def:ptype}

let eff_att_vs_def : att_def -> peff = function
   | {att=TFire;def=TFire}   -> ENotVery
   | {att=TWater;def=TWater} -> ENotVery
   | {att=TFire;def=TWater}  -> ENotVery
   | {att=TWater;def=TFire}  -> ESuper
   | _ -> ENormal
```

Upcoming events

- [today] A1 soft deadline
- [Saturday] A1 hard deadline
- [Tuesday?] A2 out

This is user defined.

THIS IS 3110