# Tuples, Records, Algebraic Data Types, Pattern Matching, Lists

most slides due to Dan Grossman



*Feels like cons, but more restricted.*

## Pairs (2-tuples)

Need a way to *build* pairs and a way to *access* the pieces

***Build*:**

- Syntax: `(e1,e2)`
- Evaluation: Evaluate **`e1`** to **`v1`** and **`e2`** to **`v2`**; result is **`(v1,v2)`**
  - A pair of values is a value
- Type-check:
  If **`e1`** has type **`ta`** and **`e2`** has type **`tb`**,
  then the pair expression has type **`ta * tb`**
  - A new kind of type



*Feels like car, cdr.*

## Pairs (2-tuples)

Need a way to *build* pairs and a way to *access* the pieces

***Access***

- Syntax: `#1 e`   `#2 e`
- Type-check: If **`e`** has type **`ta * tb`**,
  then **`#1 e`** has type **`ta`** and **`#2 e`** has type **`tb`**
- Evaluation:
  - Evaluate **`e`** to a pair of values **`v1`** and **`v2`** in the current dynamic environment
  - Return **`v1`** if using **`#1`**; return **`v2`** if using **`#2`**..



## Pairs (2-tuples)

Need a way to *build* pairs and a way to *access* the pieces

***Access** via a new form of binding **(better style)***

- Syntax: `val (x1,x2) = e`
- Type-checking: If **`e`** has type **`ta * tb`**,
  then **`x1`** has type **`ta`** and **`x2`** has type **`tb`**
- Evaluation:
  - Evaluate **`e`** to a pair of values **`v1`** and **`v2`** in the current dynamic environment
  - Extend the current dynamic environment by binding **`x1`** to **`v1`** and **`x2`** to **`v2`**.

## Examples

Functions can take and return pairs

```
fun swap (pr : int*bool) =
  let val (x,y) = pr in (y,x) end

fun sum_two_pairs (pr1 : int*int, pr2 : int*int) =
  let val (x1,y1) = pr1
      val (x2,y2) = pr2
  in x1 + y1 + x2 + y2 end

fun div_mod (x : int, y : int) =
  (x div y, x mod y)

fun sort_pair (pr : int*int) =
  let val (x,y) = pr
  in
    if x < y then pr else (y,x)
  end
```

5

## Tuples

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

- A new feature: a generalization of pairs

- **(e1,e2,…,en)**
- **ta * tb * … * tn**
- **#1 e, #2 e, #3 e, …**
- **val (x1,...,xn) = e**

These really are flat n-tuples, not nested pairs.

6

## Nesting

Pairs and tuples can be nested however you want

- Not a new feature: implied by the syntax and semantics

```
val x1 = (7,(true,9)) (* int * (bool*int) *)

val x2 = #1 (#2 x1)   (* bool *)

val x3 = (#2 x1)      (* bool*int *)

val x4 = ((3,5),((4,8),(0,0)))
         (* (int*int)*((int*int)*(int*int)) *)
```

7

## Lists

Let's try to add lists to ML. Racket does this with pairs, *e.g.:*

**(cons 1 (cons 2 (cons 3 null)))**

ML has a "no value" value written **()**, pronounced "unit," with type **unit**

So let's try: **(1, (2, (3, ())))**

What is the type of this expression?

What is the type of: **(1, (2, (3, (4, ()))))** ?

Why is this a problem?

8

# Lists

Despite nested tuples, the type of an expression still "commits" to a particular fixed "amount" of data.

In contrast, a list:
- Can have any number of elements
- But all list elements have the same type

We need a new tool to build lists in ML.



# How to build bigger types

- Already know:
  - *Base types* like `int bool unit char`
  - Ways to build (nested) *compound types*: tuples
- Today: more interesting **compound types**
- First: 3 most important type building blocks in ***any*** language
  - **Product types ("Each of")**:
    A `t` value contains *values of each of* `t1 t2 … tn`
    A `t` value contains *a* `t1` ***and*** a `t2` ***and*** a … ***and*** a `tn`
  - **Sum types ("One of")**:
    A `t` value contains *values of one of* `t1 t2 … tn`
    A `t` value is `t1` ***xor*** a `t2` ***xor*** a … ***xor*** a `tn`
  - **Recursive types ("Self reference")**: A `t` value can refer to other `t` values
- Remarkable: much data can be described by just these building blocks



Note: versions in "quotes" are not widely used terms.

# Records

*Record values* have fields (any name) holding values

`{f1 = v1, …, fn = vn}`

*Record types* have fields (any name) holding types

`{f1 : t1, …, fn : tn}`

The order of fields in a record value or type never matters
- REPL alphabetizes fields just for consistency

Building records: `{f1 = e1, …, fn = en}`

Accessing components: `#myfieldname e`

(Evaluation rules and type-checking as expected)



# Example

`{name = "Wendy", id = 41123 - 12}`

Has type

`{id : int, name : string}`

And evaluates to

`{id = 41111, name = "Wendy"}`

If some expression such as a variable **`x`** has this type, then get fields with: `#id x` `#name x`

Note we did not have to declare any record types
- The same program could also make a `{id=true,ego=false}` of type `{id:bool,ego:bool}`

## By position vs. by name

*(structural/positional)* *(nominal)*

`(4,7,9)` `{f=4,g=7,h=9}`

Common syntax decision:
- parts *by position* (as in tuples) or *by name* (as with records)
- Concise vs. clear.
- Taste, practicality, etc.

Common hybrid: function/method arguments:
- Caller: *positional*
- Callee: *nominal*
- Could totally do it differently; some languages have

15

## Tuples are sugar

`(e1,…,en)` desugars to `{1=e1,…,n=en}`

`t1*…*tn` desugars to `{1:t1,…,n:tn}`

Records with contiguous fields 1…n printed like tuples

Can write `{1=4,2=7,3=9}`, bad style

16

## Datatype bindings

Sum/one-of types:

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

**Algebraic Data Type**
- Adds new type `mytype` to environment
- Adds *constructors* to environment: `TwoInts`, `Str`, `Pizza`
- Constructor: function that makes values of new type (or is a value of new type):
  - `TwoInts : int * int -> mytype`
  - `Str : string -> mytype`
  - `Pizza : mytype`

17

## Constructing values

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

- Each value of type `mytype` came from *one of* the constructors
- Value contains:
  - Tag: which constructor (*e.g.,* `TwoInts`)
  - Carried data (*e.g.,* `(7,9)`)
- Examples:
  - `TwoInts (3+4,5+4)` evaluates to `TwoInts (7,9)`
  - `Str if true then "hi" else "bye"` evaluates to `Str "hi"`
  - `Pizza` is a value

18

## Using values

*Two* aspects to accessing a datatype value

1. Check what *variant* it is (what constructor made it)
2. Extract carried *data* (if that variant has any)

ML *could* create functions to get parts of datatype values

- Like to *pair?* or *cdr* in Racket
- Instead it does something better... ***totally awesomely better.***



## Pattern matching


***Case expression*** and ***pattern-matching***

```
fun f x = (* f has type mytype -> int *)
    case x of
        Pizza => 3
      | TwoInts(i1,i2) => i1+i2
      | Str s => String.size s
```

All-in-one:

- Multi-branch conditional, picks branch based on variant.
- Extracts data and binds to branch-local variables.
- Type-check: all branches must have same type.

- Gets even better later.



## Pattern matching

Syntax:

```
case e0 of
     p1 => e1
   | p2 => e2
     …
   | pn => en
```

- (For now), each pattern ***pi*** is:
  - a constructor name followed by the right number of variables:
  - **C** or **D x** or **E (x,y)** or ...
- Syntactically, many patterns look like expressions, but **patterns are not expressions.**
  - **We do not evaluate them.**
  - **We match e0 against their structure.**

- Precise type-checking/evaluation rules later...



## Why pattern-matching rocks


1. Cannot forget a case (inexhaustive pattern-match warning)
2. Cannot duplicate a case (redundant pattern type-checking error)
3. Cannot forget to test the variant correctly and get an error (**(car null)** in Racket)
4. It's much more general. Supports elegant, concise code.

## Useful examples

- Enumerations, including carrying other data

```
datatype suit = Club | Diamond | Heart | Spade
datatype card_value = Jack | Queen | King
                    | Ace | Num of int
```

- Alternate ways of identifying real-world things/people

```
datatype id = StudentNum of int
            | Name of string
                    * (string option)
                    * string
```



## Don't do this!

Languages lacking convenient sum/one-of types foster *bad style* where product/each-of types are misused in place of sum/one-of types:

```
(* use the student_num and ignore other
   fields unless the student_num is ~1 *)
{ student_num : int,
  first       : string,
  middle      : string option,
  last        : string }
```

- Unclear. No help from the language managing/remembering variants.



## That said...

But if instead the point is that every "person" in your program has a name and maybe a student number, then each-of is the way to go:

```
{ student_num : int option,
  first       : string,
  middle      : string option,
  last        : string }
```



## Lists!

A list is either:

- The empty list; or
- A pair of a list element and a list that holds the rest of the list.

Algebraic data types are ***just*** what we need for lists!

```
datatype mylist = Empty | Cons of int * mylist
```

datatypes can be recursive

```
val some_ints = Cons (1, Cons (2, Cons (3, Empty)))
```

## Accessing Lists

```
val some_ints = Cons (1, Cons (2, Cons (3, Empty)))
```

```
fun length (xs : mylist) =
  case xs of
    | Empty => 0
    | Cons (x,xs') => 1 + length xs'
```

```
fun sum (xs : mylist) =
  case xs of
    | Empty => 0
    | Cons (x,xs') => x + sum xs'
```

27

## Syntactic sugar for lists: build

Lists are important enough for their own syntax.

- The empty list is a value: `[]`
- A list of expressions/values is an expression/value; elements separated by commas:

`[e1,e2,…,en]` `[v1,v2,…,vn]`

- If `e1` evaluates to `v` and `e2` evaluates to a list `[v1,…,vn]`, then `e1::e2` evaluates to `[v,…,vn]`

```
e1::e2 (* pronounced "cons" *)
```

28

## Syntactic sugar for lists: access

- With pattern-matching, of course.

```
val some_ints = [1,2,3]
```

note the space between `int` and `list`

```
fun length (xs : int list) =
  case xs of
      [] => 0
    | x::xs' => 1 + length xs'
```

```
fun sum (xs : int list) =
  case xs of
      [] => 0
    | x::xs' => x + sum xs'
```

29

## Type-checking list operations

For any type `t`, type `t list` describes lists with all elements of type `t`
- `int list` `bool list` `int list list` `(int * int) list` `(int list * int) list` ...

- `[]` can have type `t list` list for *any* type
  - SML uses type `'a list` to indicate this ("quote a" or "alpha")

- `e1::e2` type-checks with type `t list` if and only if:
  - `e1` has type `t`; and
  - `e2` has type `t list`

**More on 'a soon!** *(Nothing to do with 'a in Racket.)*

30

## Example list functions

(types?)

```
fun countdown (x : int) =
  if x=0
  then []
  else x :: countdown (x-1)

fun append (xs : int list, ys : int list) =
  case xs of
      [] => ys
    | x::xs' => x :: append (xs', ys)

fun rev (xs : int list) =
  let fun revtail (acc : int list, xs : int list) =
        case xs of
            [] => acc
          | x::xs' => revtail (x :: acc, xs')
  in
    revtail ([], xs)
  end
```

## Example higher-order list functions

(type?)

```
fun map (f : int -> int, xs : int list) =
  case xs of
      [] => []
    | x::xs' => f x :: map (f, xs')
```

- But these examples only work on lists of ints.
- They should be more general: work on any list
  - and any function for map...



## Polymorphic types and type inference

The identity function: `fun id (x : int) = x`

`val id : int -> int`

It should work on anything! Omit the type: `fun id x = x`

`val id : 'a -> 'a`

**General!**

- `'a` is a ***polymorphic type variable*** that stands in for *any type*.
- *"id takes an argument of any type and returns a result of that same type."*



## Polymorphic types and type inference

```
fun swap pr = let val (x,y) = pr in (y,x) end
```

`val swap : ('a * 'b) -> ('b * 'a)`

Works on ***any*** type of pair!

```
val pair = swap (4,"hello")
```

`('a * 'b)` is ***more general than*** `(int * string)`.

Here, `int` ***instantiates*** `'a` and `string` ***instantiates*** `'b`.

## Polymorphic datatypes

- Let's make lists that can hold elements of any one type.

```
datatype 'a mylist = Empty | Cons of 'a * 'a mylist
```

- A list of "alphas" is either:
  - the empty list; or
  - a pair of an "alpha" and a list of "alphas"

```
datatype 'a list = [] | :: of 'a * 'a list
```

- The type **int list** is an *instantiation* of the type **'a list**, where the type variable **'a** is *instantiated* with **int**.

35

## Polymorphic list functions

(types?)

```
fun append (xs, ys) =
  case xs of
      [] => ys
    | x::xs' => x :: append (xs', ys)

fun rev (xs) =
  let fun revtail (acc : int list, xs : int list) =
        case xs of
            [] => acc
          | x::xs' => revtail (x :: acc, xs')
  in revtail [] xs end

fun map (f, xs) =
  case xs of
      [] => []
    | x::xs' => f x :: map (f, xs')
```

36

## Polymorphic list functions

(type?)

```
fun map (f, xs) =
  case xs of
      [] => []
    | x::xs' => f x :: map (f, xs')
```

- Type inference system (more later) chooses most general type.
- Polymorphic types show up commonly with higher-order functions.
- Polymorphic function types often give you a good idea of what the function does.

37

## Exceptions

An exception binding introduces a new kind of exception

```
exception MyFirstException
exception MySecondException of int * int
```

The **raise** primitive raises (a.k.a. throws) an exception

```
raise MyFirstException
raise (MySecondException (7,9))
```

A handle expression can handle (a.k.a. catch) an exception
- If doesn't match, exception continues to propagate

```
e1 handle MyFirstException => e2
e3 handle MyFirstException => e4
     | MySecondException (x,y) => e5
```

## Actually…

Exceptions are a lot like datatype constructors…

- Declaring an exception adds a constructor for type **`exn`**
- Can pass values of **`exn`** anywhere (e.g., function arguments)
  - Not too common to do this but can be useful
- **`handle`** can have multiple branches with patterns for type **`exn`**, just like a **`case`** expression.
- **See examples in exnopt.sml**

39

## Options

```
datatype 'a option = NONE | SOME of 'a
```

- **`t option`** is a type for any type **`t`**
  - (much like **`t list`**, but a different type, not a list)

Building:

- **`NONE`** has type **`'a option`** (much like **`[]`** has type **`'a list`**)
- **`SOME e`** has type **`t option`** if **`e`** has type **`t`** (much like **`e::[]`**)

Accessing:

- Pattern matching with case expression

Good style for functions that don't always have a meaningful result.
**See examples in exnopt.sml**

41

## Parametric Polymorphism (again) and the power of what you cannot do.

- Type 'a means "some type, but don't know what type"
- There is ***no*** way to "figure out" what type it actually is.
- No operation can distinguish between values of unknown type 'a.
- Example: *What can a function of type* ***`'a list -> int`*** *do?*

```
fun f (xs : 'a list) : int = ...
```

- *`'a -> 'a ?`*

```
fun g (x : 'a) : 'a = ...
```

43

Special case of what should be more general feature…

## Equality Types

So if we cannot inspect values of type 'a in any way, how do we write a general **`contains`** function?

```
fun contains (xs : 'a list, x : 'a) : bool = ...
```

***eqtypes (equality types):***
Special category of types that support comparison.
Accompanying eqtype variables with double quotes

Mostly accurate:
**`= : (''a * ''b) -> bool`**

```
fun contains (xs : ''a list, x : ''a) : bool = ...
```