

Programming Languages

Records, Datatypes, Case Expressions and more

Jiwon Seo

Five different things

1. **Syntax:** How do you write language constructs?
2. **Semantics:** What do programs mean? (Evaluation rules)
3. **Idioms:** What are typical patterns for using language features to express your computation?
4. **Libraries:** What facilities does the language (or a well-known project) provide “standard”? (E.g., file access, data structures)
5. **Tools:** What do language implementations provide to make your job easier? (E.g., REPL, debugger, code formatter, ...)
 - Not actually part of the language

These are 5 separate issues

- In practice, all are essential for good programmers
- Many people confuse them, but shouldn't

Our Focus

This course focuses on semantics and idioms

- Syntax is usually uninteresting
 - A fact to learn, like “The American Civil War ended in 1865”
 - People obsess over subjective preferences
- Libraries and tools crucial, but often learn new ones “on the job”
 - We are learning semantics and how to use that knowledge to understand all software and employ appropriate idioms
 - By avoiding most libraries/tools, our languages may look “silly” but so would *any* language used this way

Related Sections in Elements of ML Programming

Section 7.1 (Records), 6.1, 6.2 (Datatypes),
5.1 (Pattern Matching)

How to build bigger types

- Already know:
 - Have various *base types* like `int bool unit char`
 - Ways to build (nested) *compound types*: tuples, lists, options
- Today: more ways to build compound types
- First: 3 most important type building blocks in *any* language
 - “Each of”: A `t` value contains *values of each of* `t1 t2 ... tn`
 - “One of”: A `t` value contains *values of one of* `t1 t2 ... tn`
 - “Self reference”: A `t` value can refer to other `t` values

Remarkable: A lot of data can be described with just these building blocks

Note: These are not the common names for these concepts
-- Product, sum, recursive type

Examples

- Tuples build each-of types
 - `int * bool` contains an `int` *and* a `bool`
- Options build one-of types
 - `int option` contains an `int` *or* it contains no data
- Lists use all three building blocks
 - `int list` contains an `int` *and* another `int list` *or* it contains no data
- And of course we can nest compound types
 - `((int * int) option) * (int list list) option`

Rest of this Lecture

- Another way to build each-of types in ML
 - *Records*: have named *fields*
 - Connection to tuples and idea of *syntactic sugar*
- A way to build and use our own one-of types in ML
 - For example, a type that contains an **int** or a **string**
 - Will lead to *pattern-matching*, one of ML's coolest and strangest-to-Java-programmers features
- Later in course: How OOP does one-of types
 - Key contrast with procedural and functional programming

Records

Record values have fields (any name) holding values

```
{f1 = v1, ..., fn = vn}
```

Record types have fields (and 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 = "Amelia", id = 41123 - 12}
```

Evaluates to

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

And has type

```
{id : int, name : string}
```

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 name vs. by position

- Little difference between `(4, 7, 9)` and `{f=4, g=7, h=9}`
 - Tuples a little shorter
 - Records a little easier to remember “what is where”
 - Generally a matter of taste, but for many (6? 8? 12?) fields, a record is usually a better choice
- A common decision for a construct’s syntax is whether to refer to things *by position* (as in tuples) or *by some (field) name* (as with records)
 - A common hybrid is like with Java method arguments (and ML functions as used so far):
 - Caller uses *position*
 - Callee uses *variables*
 - Could totally do it differently; some languages have

The truth about tuples

Previous lecture gave tuples syntax, type-checking rules, and evaluation rules

But we could have done this instead:

- Tuple syntax is just a different way to write certain records
- (e_1, \dots, e_n) is another way of writing $\{1=e_1, \dots, n=e_n\}$
- $t_1 * \dots * t_n$ is another way of writing $\{1:t_1, \dots, n:t_n\}$
- In other words, records with field names 1, 2, ...

In fact, this is how ML actually defines tuples

- Other than special syntax in programs and printing, they don't exist
- You really can write $\{1=4, 2=7, 3=9\}$, but it's bad style

Syntactic sugar

“Tuples are just **syntactic sugar** for records with fields named 1, 2, ... n”

- *Syntactic*: Can describe the semantics entirely by the corresponding record syntax
- *Sugar*: They make the language sweeter 😊

Will see many more examples of syntactic sugar

- They simplify *understanding* the language
- They simplify *implementing* the language

Why? Because there are fewer semantics to worry about even though we have the syntactic convenience of tuples

Another example we saw: **andalso** and **orelse** vs. **if then else**

Boolean Operations

Operation	Syntax	Type-checking	Evaluation
andalso	e1 andalso e2	e1 and e2 must have type bool	Same as Java's e1 && e2
orelse	e1 orelse e2	e1 and e2 must have type bool	Same as Java's e1 e2
not	not e1	e1 must have type bool	Same as Java's !e1

- **not** is just a pre-defined function, but **andalso** and **orelse** must be built-in operations since they cannot be implemented as a function in ML.
 - Why? Because **andalso** and **orelse** “short-circuit” their evaluation and may not evaluate *both* **e1** and **e2**.
- Be careful to always use **andalso** instead of **and**.
- **and** is completely different. We will get back to it later.

Style with Booleans

Language does not *need* `andalso` , `orelse` , or `not`

```
(* e1 andalso e2 *)  
if e1  
then e2  
else false
```

```
(* e1 orelse e2 *)  
if e1  
then true  
else e2
```

```
(* not e1 *)  
if e1  
then false  
else true
```

Using more concise forms generally much better style

And definitely please do not do this:

```
(* just say e (!!!) *)  
if e  
then true  
else false
```

Datatype bindings

A “strange” (?) and totally awesome (!) way to make one-of types:

- A `datatype` binding

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

- Adds a new type `mytype` to the environment
- Adds *constructors* to the environment: `TwoInts`, `Str`, and `Pizza`
- A constructor is (among other things), a function that makes values of the new type (or is a value of the new type):
 - `TwoInts : int * int -> mytype`
 - `Str : string -> mytype`
 - `Pizza : mytype`

The values we make

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

- Any value of type **mytype** is made from *one of* the constructors
- The value contains:
 - A “tag” for “which constructor” (e.g., **TwoInts**)
 - The corresponding 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

Using them

So we know how to *build* datatype values; need to *access* them

There are *two* aspects to accessing a datatype value

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

Notice how our other one-of types used functions for this:

- `null` and `isSome` check variants
- `hd`, `tl`, and `valOf` extract data (raise exception on wrong variant)

ML *could* have done the same for datatype bindings

- For example, functions like “isStr” and “getStrData”
- Instead it did something better

Case

ML combines the two aspects of accessing a one-of value with a *case expression* and *pattern-matching*

- Pattern-matching much more general/powerful

Example:

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

- A multi-branch conditional to pick branch based on variant
- Extracts data and binds to variables local to that branch
- Type-checking: all branches must have same type
- Evaluation: evaluate between **case ... of** and the right branch

Patterns

In general the syntax is:

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

For today, each *pattern* is a constructor name followed by the right number of variables (i.e., `C` or `C x` or `C (x,y)` or ...)

- Syntactically most patterns (all today) look like expressions
- But patterns are not expressions
 - We do not evaluate them
 - We see if the result of `e0` *matches* them

Why this way is better

0. You can use pattern-matching to write your own testing and data-extractions functions if you must

– But do not do that on your homework

1. You cannot forget a case (inexhaustive pattern-match warning)
2. You cannot duplicate a case (a type-checking error)
3. You will not forget to test the variant correctly and get an exception (like `hd []`)
4. Pattern-matching can be generalized and made more powerful, leading to elegant and concise code

Useful examples

Let's fix the fact that our only example datatype so far was silly...

- 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

Unfortunately, bad training and languages that make one-of types inconvenient lead to common *bad style* where each-of types are used where one-of types are the right tool

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

- Approach gives up all the benefits of the language enforcing every value is one variant, you don't forget branches, etc.
- And makes it less clear what you are doing

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 }
```

Expression Trees

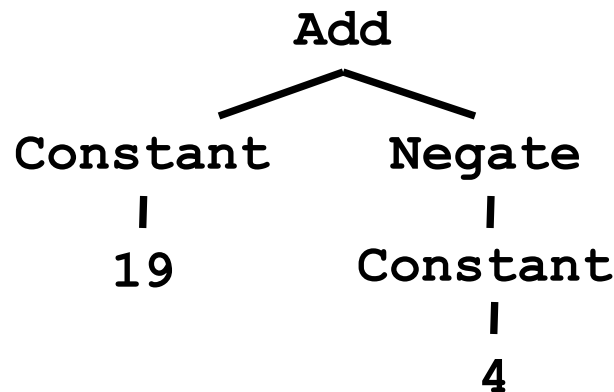
A more exciting example of a datatype, using self-reference

```
datatype exp = Constant of int
              | Negate   of exp
              | Add      of exp * exp
              | Multiply of exp * exp
```

An expression in ML of type **exp**:

```
Add (Constant (10+9), Negate (Constant 4))
```

How to picture the resulting value in your head:



Recursion

Not surprising:

Functions over recursive datatypes are usually recursive

```
fun eval e =  
  case e of  
    Constant i      => i  
  | Negate e2        => ~ (eval e2)  
  | Add(e1,e2)       => (eval e1) + (eval e2)  
  | Multiply(e1,e2)  => (eval e1) * (eval e2)
```

Putting it together

```
datatype exp = Constant of int
             | Negate    of exp
             | Add       of exp * exp
             | Multiply  of exp * exp
```

Let's define `max_constant : exp -> int`

Good example of combining several topics as we program:

- Case expressions
- Local helper functions
- Avoiding repeated recursion
- Simpler solution by using library functions

See the `.sm1` file...

Careful definitions

When a language construct is “new and strange,” there is *more* reason to define the evaluation rules precisely...

- ... so let's review datatype bindings and case expressions “so far”
 - *Extensions* to come but won't invalidate the “so far”

Datatype bindings

```
datatype t = C1 of t1 | C2 of t2 | ... | Cn of tn
```

Adds type t and constructors C_i of type $t_i \rightarrow t$

- $C_i \ v$ is a value, i.e., the result “includes the tag”

Omit “of t ” for constructors that are just tags, no underlying data

- Such a C_i is a value of type t

Given an expression of type t , use *case expressions* to:

- See which variant (tag) it has
- Extract underlying data once you know which variant

Datatype bindings

```
case e of p1 => e1 | p2 => e2 | ... | pn => en
```

- As usual, can use a case expressions anywhere an expression goes
 - Does not need to be whole function body, but often is
- Evaluate **e** to a value, call it **v**
- If **p_i** is the first *pattern* to *match* **v**, then result is evaluation of **e_i** in environment “extended by the match”
- Pattern **C_i (x₁ , ... , x_n)** matches value **C_i (v₁ , ... , v_n)** and extends the environment with **x₁** to **v₁** ... **x_n** to **v_n**
 - For “no data” constructors, pattern **C_i** matches value **C_i**

Datatype bindings

```
case e of p1 => e1 | p2 => e2 | ... | pn => en
```

```
p ::= _ | c | x | (p1, ..., pn) | {x1= p1, ..., xn= pn} | [] | p1::p2 | X | X(p)
```

- Evaluate **e** to a value, call it **v**
- If **p_i** is the first *pattern* to *match* **v**, then result is evaluation of **e_i** in environment “extended by the match”
- Pattern **C_i (x₁, ..., x_n)** matches value **C_i (v₁, ..., v_n)** and extends the environment with **x₁** to **v₁** ... **x_n** to **v_n**
 - For “no data” constructors, pattern **C_i** matches value **C_i**

Recursive datatypes

Datatype bindings can describe recursive structures

- Have seen arithmetic expressions
- Now, linked lists:

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

val x = Cons (4, Cons (23, Cons (2008, Empty)))

fun append_my_list (xs, ys) =
  case xs of
    Empty => ys
  | Cons (x, xs') => Cons (x, append_my_list (xs', ys))
```

Options are datatypes

Options are just a predefined datatype binding

- **NONE** and **SOME** are *constructors*, not just functions
- So use pattern-matching not **isSome** and **valOf**

```
fun inc_or_zero intoption =  
  case intoption of  
    NONE => 0  
  | SOME i => i+1
```


Lists are datatypes

Do not use `hd`, `tl`, or `null` either

- `[]` and `::` are constructors too
- (strange syntax, particularly *infix*)

```
fun sum_list xs =  
  case xs of  
    [] => 0  
  | x::xs' => x + sum_list xs'  
  
fun append (xs,ys) =  
  case xs of  
    [] => ys  
  | x::xs' => x :: append(xs',ys)
```

Why pattern-matching

- Pattern-matching is better for options and lists for the same reasons as for all datatypes
 - No missing cases, no exceptions for wrong variant, etc.
- So why are `null`, `t1`, etc. predefined?
 - For passing as arguments to other functions (next lecture)
 - Because sometimes they are convenient
 - But not a big deal: could define them yourself

Excitement ahead...

Learn some deep truths about “what is really going on”

- Using much more syntactic sugar than we realized

- Every val-binding and function-binding uses pattern-matching
- Every function in ML takes exactly one argument

First need to extend our definition of pattern-matching...

Each-of types

So far have used pattern-matching for one of types because we *needed* a way to access the values

Pattern matching also works for records and tuples:

- The pattern **(*x*₁ , ... , *x*_n)**
matches the tuple value **(*v*₁ , ... , *v*_n)**
- The pattern **{*f*₁=*x*₁ , ... , *f*_n=*x*_n}**
matches the record value **{*f*₁=*v*₁ , ... , *f*_n=*v*_n}**
(and fields can be reordered)

Example

This is poor style, but based on what I told you so far, the only way to use patterns

- Works but poor style to have one-branch cases

```
fun sum_triple triple =  
  case triple of  
    (x, y, z) => x + y + z  
  
fun full_name r =  
  case r of  
    {first=x, middle=y, last=z} =>  
      x ^ " " ^ y ^ " " ^ z
```

Example

This is poor style, but based on what I told you so far, the only way to use patterns

- Works but poor style to have one-branch cases

```
fun sum_triple triple =  
  case triple of  
    (x, y, z) => x + y + z  
  
fun full_name r =  
  case r of  
    {first=x, middle=y, last=z} =>  
      x ^ " " ^ y ^ " " ^ z
```

Val-binding patterns

- New feature: A val-binding can use a pattern, not just a variable
 - (Turns out variables are just one kind of pattern, so we just told you a half-truth in Lecture 1)

```
val p = e
```

- Great for getting (all) pieces out of an each-of type
 - Can also get only parts out (not shown here)
- Usually poor style to put a constructor pattern in a val-binding
 - Tests for the one variant and raises an exception if a different one is there (like `hd`, `tl`, and `valOf`)

Better example

This is okay style

- Though we will improve it again next
- Semantically identical to one-branch case expressions

```
fun sum_triple triple =  
  let val (x, y, z) = triple  
  in  
    x + y + z  
  end
```

```
fun full_name r =  
  let val {first=x, middle=y, last=z} = r  
  in  
    x ^ " " ^ y ^ " " ^ z  
  end
```


Function-argument patterns

A function argument can also be a pattern

- Match against the argument in a function call

```
fun f p = e
```

Examples (great style!):

```
fun sum_triple (x, y, z) =  
  x + y + z
```

```
fun full_name {first=x, middle=y, last=z} =  
  x ^ " " ^ y ^ " " ^ z
```

Hmm

A function that takes one triple of type `int*int*int` and returns an `int` that is their sum:

```
fun sum_triple (x, y, z) =  
    x + y + z
```

A function that takes three `int` arguments and returns an `int` that is their sum

```
fun sum_triple (x, y, z) =  
    x + y + z
```

See the difference? (Me neither.) ☺

The truth about functions

- In ML, every function takes exactly one argument (*)
- What we call multi-argument functions are just functions taking one tuple argument, implemented with a tuple pattern in the function binding
 - Elegant and flexible language design
- Enables cute and useful things you cannot do in Java, e.g.,

```
fun rotate_left (x, y, z) = (y, z, x)
fun rotate_right t = rotate_left(rotate_left t)
```

* “Zero arguments” is the unit pattern `()` matching the unit value `()`