CS 3110

Lecture 3: Functions and data

Prof. Clarkson

Fall 2014

Today's music: Function by E-40 (Clean remix)

Review

Last week:

- Intro to syntax and semantics of OCaml

Today:

- **Functions:** the most important part of functional programming
- **Data:** datatypes, records, tuples

Function declaration

Functions: **the most important building block** in the whole course

- Like Java methods, have arguments and result
- But no classes, **`this`**, **`return`**, etc.

Example *function declaration*:

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

Note: "**`rec`**" is required because the body includes a recursive *function call*: **`pow(x,y-1)`**

Questions

If we want to understand functions in OCaml, what questions do we need to ask?

Syntax?

Type checking?

Evaluation?

Function declaration: 3 questions

- **Syntax**: (for now)

```
let rec f ((x1 : t1), … , (xn : tn)):t = e
```

- **Evaluation**:
  - No evaluation to do, yet; just declaring the function
- **Type-checking:**
  - Conclude that `f : (t1 * … * tn) -> t` if `e : t` under assumptions:
    - `x1 : t1, …, xn : tn` (arguments with their types)
    - `f : (t1 * … * tn) -> t` (for recursion)

Function calls

A new kind of expression: 3 questions

Syntax: (for now)

`e0 (e1,…,en)`

- Parentheses optional if there is exactly one argument
- Space before left paren is optional

Type-checking:

If:

- `e0` has some type `(t1 * … * tn) -> t`
- `e1` has type `t1`, …, `en` has type `tn`

Then:

- `e0 (e1,…,en)` has type `t`

Example: `pow(x,y-1)` in previous example has type `int`

Function calls, continued

`e0 (e1,…,en)`

Evaluation:

1. Evaluate `e0` to a function
   `let rec x0 ((x1 : t1), … , (xn : tn)) = e`
   – Since call type-checked, result is guaranteed to be a function
2. Evaluate arguments to values `v1, …, vn`
3. Substitute `vi` for `xi` in `e` -- again, TRICKY -- producing expression `e'`
4. Evaluate `e'` to a value `v`, which is result

Example functions

```
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)

let cube (x : int) : int =
  pow (x,3)

let sixtyfour = cube 4

let fortytwo = pow(2,4) + pow(4,2) + cube(2) + 2
```

Longer examples in the notes—study them!

Question #1

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> ???
```

Question #1

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> ???
```

A. 9

B. `if y=0 then 1 else x*pow(x,y-1)`

C. `2*pow(3,2)`

D. `if 2=0 then 1 else 3*pow(3,2-1)`

Question #1

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> ???
```

A. 9

B. if y=0 then 1 else x*pow(x,y-1)

C. 2*pow(3,2)

D. if 2=0 then 1 else 3*pow(3,2-1)

Question #1

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> if 2=0 then 1 else 3 * pow(3,2-1)
```

Question #2

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> if 2=0 then 1 else 3 * pow(3,2-1)
  --> ???
```

Question #2

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> if 2=0 then 1 else 3 * pow(3,2-1)
  --> ???
```

A. `false`

B. `if false then 1 else 3*pow(3,2-1)`

C. `3*pow(3,2-1)`

D. `3*3`

Question #2

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> if 2=0 then 1 else 3 * pow(3,2-1)
  --> ???
```

A. false
B. if false then 1 else 3*pow(3,2-1)
C. 3*pow(3,2-1)
D. 3*3

Question #2

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> if 2=0 then 1 else 3 * pow(3,2-1)
  --> if false then 1 else 3*pow(3,2-1)
```

Example function evaluation

```
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)
```

```
pow(3,2)
  --> if 2=0 then 1 else 3 * pow(3,2-1)
  --> if false then 1 else 3 * pow(3,2-1)
  --> 3 * pow(3,2-1)
  --> 3 * pow(3,1)
  --> 3 * (if 1=0 then 1 else 3 * pow(3,1-1))
  --> 3 * (if false then 1 else 3 * pow(3,1-1))
  --> 3 * (3 * pow(3,1-1))
  --> 3 * (3 * pow(3,0))
  --> 3 * (3 * (if 0=0 then 1 else 3 * pow(3,0-1)))
  --> 3 * (3 * (if true then 1 else 3 * pow(3,0-1)))
  --> 3 * (3 * 1)
  --> 3 * 3
  --> 9
```

Alternative function syntax

All three are equivalent:

```
let abs (x : int) : int =
  if x<0 then -x else x

let abs : int -> int =
  function x -> if x<0 then -x else x

let abs : int -> int =
  fun x -> if x<0 then -x else x
```

(and you could leave out the types, too)

Omitting argument types

When argument type omitted, so are extra parens:

```
let rec pow ((x : int), (y : int)) : int =
  if y=0 then 1
  else x * pow(x,y-1)

let rec pow’ (x , y) : int =
  if y=0 then 1
  else x * pow(x,y-1)

let cube (x : int) : int =
  pow (x,3)

let cube’ x : int =
  pow (x,3)
```

Some gotchas

Three common "gotchas":

- The use of `*` in type syntax is not multiplication
  - Example: `int * int -> int`
  - In expressions, `*` is multiplication: `x * pow(x,y-1)`
- Order matters: cannot refer to later function bindings from earlier
  - So helper functions must come before their uses
  - Need `and` construct for *mutual recursion*
- Inscrutable error messages if you mess up function-argument syntax

Function specifications

Specification: contract between function and rest of the code about how function will behave

```
(* requires: precondition *)
(* returns:  postcondition *)
let f(x) = ...
```

Function specifications

Postcondition: Predicate that is guaranteed to hold when function returns

- Responsibility: Function must ensure that postcondition holds
- Function names usually give a clue as to postcondition

```
(* requires: ... *)
(* returns:  the lowercase character
   corresponding to c *)
let lowercase (c : char) : char = ...
```

Function specifications

Precondition: Predicate that is assumed to hold when function is called

– Responsibility: Programmer who calls function must ensure that precondition holds

```
(* requires: c is an uppercase letter *)
(* returns:  the lowercase character
   corresponding to c *)
let lowercase (c : char) : char = ...
```

- If precondition doesn't hold, function is allowed to behave arbitrarily
- Robust implementations try to do something sane though
  - e.g., check precondition and immediately fail if it doesn't hold
- If function has no particular precondition, omit `requires` comment

Question #3

Given this code, which are permissible behaviors for **`lowercase('?')`**?

```
(* requires: c is an uppercase letter *)
(* returns:  the lowercase character
   corresponding to c *)
let lowercase (c : char) : char = …
```

A. It can throw an exception
B. Return **`'?'`**
C. Return "**`zardoz`**"
D. A and B
E. A, B, and C

Question #3

Given this code, which are permissible behaviors for **`lowercase('?')`**?

```
(* requires: c is an uppercase letter *)
(* returns:  the lowercase character
   corresponding to c *)
let lowercase (c : char) : char = …
```

A. It can throw an exception

B. Return **`'?'`**

C. Return "**`zardoz`**"

D. A and B

E. A, B, and C

FUNCTIONS…DATA

Two new kinds of data

- **Datatypes** (*one-of* types)
- **Records, tuples** (*each-of* types)

Datatype declaration

- New sort of declaration (variable declaration, function declaration): *type declaration*

```
type mybool = Myfalse | Mytrue
```

  - Creates a *one-of type* named **`mybool`**
  - Creates two *constructors* named **`Mytrue`** and **`Myfalse`**
    - Those are also values of type **`mybool`**

- In fact, that's effectively how Booleans are defined in OCaml:

```
type bool = false | true
```

Datatype for Days

```
(* similar to an enum in Java or C *)

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

(* returns: the day of the week for d *)
let day_to_int (d : day) =
    if d=Sun then 1
    else if d=Mon then 2
    else if d=Tue then 3
    else if d=Wed then 4
    else if d=Thu then 5
    else if d=Fri then 6
    else (* d=Sat *) 7
```

But there's a much more idiomatic way of expressing this in OCaml…

Datatype for Days

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

These are patterns

Pattern matching: more beautiful idiom than nested if expressions

Datatype semantics

- We've seen **syntax** for datatype declarations, pattern matching
- What about **type checking**, **evaluation**?

 …hold that thought!

Record declaration

- Also declared with *type declaration*:

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

  – Creates a *each-of type* named **`time`**

- To *build* a record:

```
{hour=10; min=10; ampm="am"}
```

  – order of *fields* doesn't matter; could write

```
{min=10; ampm="am"; hour=10}
```

- To *access* fields of record variable **`t`**:

```
t.min
```

Record expressions

- **Syntax:** `{f1 = e1; …; fn = en}`

- **Evaluation:**
  - If `e1-->v1`, and `e2-->v2`, and … `en-->vn`
  - Then `{f1 = e1; …; fn = en}-->{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 declared type of the form `{f1:t1, …, fn:tn}`
  - then `{f1 = e1; …; fn = en}: t`

Record field access

- **Syntax:** `e.f`

- **Evaluation:**
  - If `e-->{f = v, …}`
  - Then `e.f-->v`

- **Type-checking:**
  - If `e:t1`
  - and if `t1` is a declared type of the form `{f:t2, …}`
  - then `e.f:t2`

Datatypes vs. records

	Declare	Build/ construct	Access/ destruct
Datatype	`type`	Constructor name	Pattern matching with `match`
Record	`type`	Record expression with `{…}`	Field selection with dot operator `.`

Question 4

Which of the following would be better represented with records rather than datatypes?

A. *Coins*, which can be pennies, nickels, dimes, or quarters

B. *Students*, who have names and NetIDs

C. A *plated dessert*, which has a sauce, a creamy component, and a crunchy component

D. A and C

E. B and C

Question 4

Which of the following would be better represented with records rather than datatypes?

A. Coins, which can be pennies, nickels, dimes, or quarters
B. Students, who have names and NetIDs
C. A *plated dessert*, which has a sauce, a creamy component, and a crunchy component
D. A and C
E. B and C

Your turn!

- Part of your development as a programmer is learning new language features on your own
- Here's your chance to practice:
  - Learn pairs, tuples, and unit
  - The remaining slides will help you
  - We'll start the next lecture with some clicker questions about them!

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 data accessed by position

Pairs

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

e.g.,

- **`(1,2)`**
- **`(true, "Hello")`**
- **`("cs", 3110)`**

Note: looks a lot like the arguments passed to a 2-argument function

We need a way to *build* pairs and a way 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 **pair type**
  - *(Though we've seen* **`*`** *before in function types…)*

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…

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

Instead of generalizing projection functions,
use pattern matching…

Pattern-matching tuples

```
let sum_triple (triple:int*int*int)  =
    let (x, y, z) = triple
    in x + y + z
```

- `(x, y, z)` is a *pattern*
  - because it's on the LHS of equals in `let`
- Evaluation (intuitively):
  - Value on RHS of equals is "matched" against pattern
  - Each variable in pattern is bound to "matching" part of value

Pattern-matching records

The same syntax works for records:

```
type stooges = {larry:int; moe:int; curly:int}

let sum_stooges (s:stooges)  =
   let {larry=x; moe=y; curly=z} = s
   in x + y + z
```

By name vs. by position, again

- 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"
    - Names are self-documenting
  - Generally a matter of taste, but for many (4? 8? 12?) fields, a record is usually a better choice

Datatypes vs. records vs. tuples

	Declare	Build/construct	Access/destruct
Datatype	`type`	Constructor name	Pattern matching with `match`
Record	`type`	Record expression with `{…}`	Pattern matching with `let` OR field selection with dot operator `.`
Tuple	N/A	Tuple expression with `(…)`	Pattern matching with `let` OR `fst` or `snd`

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`**
- Might seem dumb now; will be useful later!

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Registration

If you (still) want in:

- Keep attending and doing problem sets
- Email Course Administrator with your full name and NetID by the **end of today**
- You will be placed in "Waiting Set". NO PROMISES.
- Tomorrow, I begin working through the Waiting Set

Upcoming events

- **PS 0 is out now, PS1 comes out Thursday**
- No recitations on Tuesday (today), there are recitations Wednesday and Thursday
- Clarkson office hours this week: TR 1:30-2:30
- TA office hours and consulting start soon; times and places TBA

We trynna function.

THIS IS 3110