CS 3110

Lecture 4: Lists and more data

Prof. Clarkson

Fall 2014

Today's music: "Everything is AWESOME!!!" from *The Lego Movie*

Review

Features so far: variables, operators, let expressions, if expressions, functions, datatypes, records

Today:

- Review **tuples**
- **Lists**, options, algebraic datatypes

Question #1

A **tuple** contains…

A. A fixed number of components all of which must have the same type
B. Exactly two components which may have different types
C. A fixed number of components each of which may have a different type
D. Exactly two components which must have the same type
E. I forgot to study tuples

Question #1

A **tuple** contains…

A. A fixed number of components all of which must have the same type
B. Exactly two components which may have different types
C. **A fixed number of components each of which may have a different type**
D. Exactly two components which must have the same type
E. I forgot to study tuples

Question #2

To access the first component of a pair, I can use…

A. The `fst` projection function

B. Pattern matching with a `let` expression

C. The `unit` expression

D. A and B

E. A and C

Question #2

To access the first component of a pair, I can use…

A. The `fst` projection function

B. Pattern matching with a `let` expression

C. The `unit` expression

D. A and B

E. A and C

Question #3

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 #3

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)`

Hmm...

Q: What is the type of `(1,2,3)`?

A: `int*int*int`

Q: What is the type of `sum_triple` in:

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

A: `int*int*int->int`

Hmm...

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

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

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

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

See the difference? (Me neither.) ☺ *More next week…*

PS1 is out today

- Due in 7 days: Thursday, Sept. 11, 11:59 pm
- Covers everything through today
  - In lecture and in notes
  - A couple very small things to learn on your own:
    - E.g., (+) is prefix version of + operator
  - Might (not) find some library modules useful (**`List`**, **`Char`**, …)
- Must be done with a partner
  - Find a partner on Piazza
  - Form a partnership on CMS well before due day
  - Right way vs. wrong way…
  - Everything is AWESOME when you're part of a team!!!

Problem set grading

- **Automated grading** for correctness
  - Critical for you to program to the specification we give you
  - No-compile grace period: we notify you Thursday night, you get till Saturday 11:59 pm to fix it
  - If you submit a small patch (2-3 lines) that gets code to compile, just a minor penalty
  - If your code still can't be compiled, you get a zero
- Manual grading for written problems, code style
- You get two *late passes* for use in semester
  - Automatic 48-hour extension: assignment becomes due Saturday at 11:59 pm
  - No-compile grace period does not apply
  - Both partners must relinquish a pass
  - To use: email Course Administrator
- In case of true emergency (medical, family) contact Instructor ASAP

LISTS…ARE AWESOME!!!

Lists

- So far, the type of a variable commits to a particular "amount" of data
  - e.g., pair has two components, exactly
- In contrast, a *list* can have any number of elements
- But unlike tuples, all elements have the same type

Need ways to *build* lists and *access* the pieces…

Building Lists

Syntax:

- A list of values is a value; elements separated by semi-colons:

```
[v1;v2;…;vn]
```

- The empty list is a value:

```
[] (* :: pronounced “nil” *)
```

- Prepend an element to beginning of list:

```
e1::e2 (* :: pronounced “cons” *)
```

Evaluation:

- If `e1-->v1` and…and `en-->vn`
  then `[e1;…;en]-->[v1;…vn]`
- If `e1-->v` and `e2-->[v1,…,vn]`
  then `e1::e2-->[v,v1,…,vn]`
    - v is the *head* of new list; rest is *tail*

Type-checking list builders

New types:

For any type `t`, the type `t list` describes lists where all elements have type `t`

- `[1;2;3] : int list`
- `[true] : bool list`
- `[[1+1;2-3];[3*7]] : int list list`
- `[(1,2);(2,4)] : (int * int) list`
- `[([0;1],2);([3;4],5)] : (int list * int) list`

Caution: semi-colons in lists, commas in tuples

Cons:

If `e1 : t` and `e2 : t list` then `e1::e2 : t list`

With parens for clarity:

If `e1 : t` and `e2 : (t list)` then `(e1::e2) : (t list)`

Nil:

`[]:t list` for *any* type `t`

- OCaml uses type `'a list` to indicate this ("quote a" or "alpha")

Accessing lists

A list is either:

- nil
- or a head “cons-ed” onto a tail

Use **pattern matching** to access list in one of those ways:

```
let empty lst =
  match lst with
     []   -> true
   | h::t -> false
```

Your brain is probably exploding with AWESOME questions about pattern matching now...

Example list functions

```
let rec sum_list (lst : int list) : int =
  match lst with
    []    -> 0
  | h::t -> h + sum_list(t)

let rec length (lst : int list) : int =
  match lst with
    []     -> 0
  | x::xs -> 1 + length(xs)

let rec append ((lst1:'a list),(lst2:'a list))
    : 'a list =
  match lst1 with
    []    -> lst2
  | h::t -> h::append(t,lst2)
(* append is available as built-in operator @ *)
```

Lists are immutable

- No way to *mutate* an element of a list
- Instead, build up new lists out of old
  - e.g., **`append`**

Question #4

What is the type of `31::[10]`?

A. int

B. int list

C. int*(int list)

D. (int*int) list

E. Not well-typed

Question #4

What is the type of `31::[10]`?

A. int

B. int list

C. int*(int list)

D. (int*int) list

E. Not well-typed

Question #5

```
match ["zar";"doz"] with
   []    -> "kitteh"
 | h::t -> h
```

To what value does the above expression evaluate?

A. "zar"
B. "doz"
C. "kitteh"
D. []
E. h

Question #5

```
match ["zar";"doz"] with
   []    -> "kitteh"
| h::t -> h
```

To what value does the above expression evaluate?

A. "zar"

B. "doz"

C. "kitteh"

D. []

E. h

Recursion!

Functions over lists are usually recursive: only way to "get to" all the elements

- What should the answer be for the empty list?
- What should the answer be for a non-empty list?
  – Typically in terms of the answer for the tail of the list

Accessing lists, with poor style

- Two library functions that return head and tail
  - `List.hd, List.tl`

- **They are usually poor style when directly applied to a list**
  - Why? Because they throw exceptions; you can easily write buggy code
  - Whereas pattern matching guarantees no exceptions when destructing list; it's hard to write buggy code!

OPTIONS

What is max of empty list?

```
let max (x, y) =
  if x>y then x else y

let rec max_list (lst : int list) : int =
  match lst with
    [] -> ???
  | h::t -> max(h,max_list(t))
```

negative infinity would be a reasonable choice…
or could raise an exception…
or might return a null Integer in Java…
but OCaml gives us another AWESOME option!

Options

Options:

- `t option` is a type for any type `t`
  - (much like `t list` is a type for any type `t`)

Building and Type Checking and Evaluation:

- `None` has type `'a option`
  - much like `[]` has type `'a list`
  - `None` is a value
- `Some e :t option` if `e:t`
  - much like `e::[]` has type `t list` if `e:t`
  - If `e-->v` then `Some e-->Some v`

Accessing:

```
match e with
      None -> ...
    | Some x -> ...
```

Again: What is max of empty list?

```
let max (x, y) =
  if x>y then x else y

let rec max_list (lst : int list) : int option =
  match lst with
     []    -> None
   | h::t -> match max_list(t) with
               None    -> Some h
             | Some x -> Some (max(h,x))
```

Very stylish!
…no possibility of exceptions
…no chance of programmer ignoring a "null return"

ALGEBRAIC DATATYPES

Recall: datatype for days

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

One-of type
Each “branch” is a *constructor*

But wait, there’s more…

Algebraic datatypes

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

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

- Each constructor can *carry* data along with it
- A constructor behaves like 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`

Algebraic datatypes

```
type 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 of evaluation:
  - **`TwoInts(3+4,5+4)-->TwoInts(7,9)`**
  - **`Str(if true then "hi" else "bye") -->Str("hi")`**
  - **`Pizza`** is a value

Algebraic datatypes

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 carries any)

Pattern matching alg. datatypes

OCaml combines the two aspects of accessing an algebraic datatype into (once again) pattern matching:

```
let f (x:mytype) : int =
  match x with
    Pizza -> 3
  | TwoInts(i1,i2) -> i1+i2
  | Str s -> String.length s
```

- One branch per variant
- Each branch
  - extracts the carried data and
  - binds data to variables local to that branch

Patterns for alg. datatypes

Syntax:

```
match e0 with
   p1 -> e1
 | p2 -> e2
 | ...
 | pn -> en
```

For now, 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 patterns might look like expressions
- But patterns are not expressions
  - OCaml does not evaluate patterns
  - OCaml does determine whether result of **`e0`** *matches* patterns

Type checking and evaluation will take us till next week…

Why pattern matching is AWESOME

1. You can't forget a case
   (inexhaustive pattern-match warning)
2. You can't duplicate a case
   (unused match case warning)
3. You can't get an exception
   from forgetting to test the variant
   (e.g., `hd []`)
4. Pattern matching leads to elegant, concise, beautiful code

Useful datatypes

That last datatype was silly…

- Enumerations, including containing other data

```
type suit = Club | Diamond | Heart | Spade
type rank = Jack | Queen | King
          | Ace | Num of int
```

- Alternative ways of representing data

```
(* Every student either has an id number
 * or (temporarily) is identified by name. *)
type student_id =
  IdNum of int
| FullName of string
```

Please hold still for 1 more minute

WRAP-UP FOR TODAY

Registration

- If you put yourself on the Waiting Set, you should have received an email

In honor of the 50th Anniversary of the Department of Computer Science, Cornell Cinema Presents

You, Robot

A timely film series* that is guaranteed to get you thinking about the growing autonomy of machines.

The Day the Earth Stood Still (1951)

Thursday, Sep 4
7:00 pm
Willard Straight Theatre

Introduced by Professor Charles Van Loan (CS)

"Gort, Klaatu barada nikto."

*The Day the Earth Stood Still / 2001: A Space Odyssey / Robocop / Ghost in the Shell / Metropolis / Robot and Frank

Upcoming events

- **PS1 is out today, due one week from today**
- Clarkson office hours this week: TR 1:30-2:30
- TA office hours and consulting start tonight; times and places on course website

Everything is AWESOME!!!

THIS IS 3110