Programming Languages and Techniques (CIS120e)

Lecture 2
Sep 10, 2010

OCaml Overview
Lists

Design

Design is the process of translating informal specifications (“word problems”) into running code.

1. Understand the problem
   What are the relevant concepts and how do they relate?
2. Formalize the interface
   How should the program interact with its environment?
3. Write test cases
   How does the program behave on typical inputs? On unusual ones? On erroneous ones?
4. Implement the required behavior
   Often by decomposing the problem into simpler ones and applying the same recipe to each

A Design Problem

Imagine the owner of a movie theater who has complete freedom in setting ticket prices. The more he charges, the fewer people can afford tickets. In a recent experiment the owner determined a precise relationship between the price of a ticket and average attendance. At a price of $5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime ($.10) increases attendance by 15. Unfortunately, the increased attendance also comes at an increased cost. Every performance costs the owner $180. Each attendee costs another four cents ($0.04). The owner would like to know the exact relationship between profit and ticket price so that he can determine the price at which he can make the highest profit.
Solution

```ocaml
let attendees (ticket_price : int) : int =
  120 + ((15 * (500 - ticket_price) ) / 10)

let revenue (ticket_price : int) =
  (attendees ticket_price) * ticket_price

let cost (ticket_price : int) : int =
  18000 + (attendees ticket_price) * 4

let profit (ticket_price : int) : int =
  (revenue ticket_price) - (cost ticket_price)

;;
assert_equal "Attendees at $5.00" (attendees 500) 120;
assert_equal "Attendees at $4.90" (attendees 490) (120 + 15);
assert_equal "Profit at $5.00" (profit 500) 43540;
;;
```

Solution (improved)

```ocaml
let attendees (ticket_price : int) : int =
  120 + ((15 * (500 - ticket_price) ) / 10)

let revenue (ticket_price : int) =
  (attendees ticket_price) * ticket_price

let cost (ticket_price : int) : int =
  18000 + (attendees ticket_price) * 4

let profit (ticket_price : int) : int =
  (revenue ticket_price) - (cost ticket_price)

;;
assert_equal "Attendees at $5.00" (attendees 500) 120
assert_equal "Attendees at $4.90" (attendees 490) (120 + 15)
assert_equal "Profit at $5.00" (profit 500) 43540
```

Step 1: Understand the problem

- In this problem there are four relevant concepts:
  - profit
  - revenue
  - cost
  - attendees
- There are relationships among them:
  - profit = revenue – cost
  - revenue = ticket-price * attendees
  - cost and attendees are functions of the ticket price
    - hence profit and revenue are too

How we got there...

A transcript of the Interactive Interlude from last lecture, with a few more details at the end...
Step 2: Formalize the Interface

• Here, there is only one (mildly) interesting choice:
  
  *How should we represent the ticket price?*

  – option 1: integers
  – option 2: floating point numbers

• Either will work
  – for simplicity, we choose integers

Step 3: Write Test Cases

• By looking at the data from the informal specification, we can calculate* these tests:
  – profit at $5.00 is $415.20
  – profit at $4.90 is $476.10

• Working out tests by hand also helps nail-down corner cases and can help you understand the problem better.

Writing the Test Cases in OCaml

• Record the test cases as assertions in the program:
  
  this is an equality assertion
  
  ;; assert_eq "profit at $5.00" (profit 500) 41520
  ;; assert_eq "profit at $4.90" (profit 490) 47610

*OCaml will let you omit these type annotations, but including them is mandatory for CIS120. Using type annotations is good documentation; they also improve the error messages you get from the compiler. When you get a type error message from the compiler, the first thing you should do is check that your type annotations are there and that they are what you expect.

*The OCaml toplevel can be used as a calculator and to “play around” with definitions while you’re understanding the program and the test cases. You should record the tests you develop as assertions so that they can be run again later when the program changes.
Step 4: Implement the Behavior

profit, revenue, and cost are all easy to define:

```plaintext
let revenue (price:int) : int = price * (attendees price)
let cost (price:int) : int = 18000 + 4 * (attendees price)

(* price is in cents *)
let profit (price:int) : int = (revenue price) - (cost price)
```

Apply the Design Pattern Recursively

attendees* requires a bit of thought:

```plaintext
let attendees (price:int) : int =
120 + (500 – price) / (10 * 15)

;; assert_eq “atts. at $5.00”
(attendees 500) 120

;; assert_eq “atts. at $4.90”
(attendees 490) 135
```

*Note that the definition of attendees must go before the definitions of cost and revenue because the latter make use of the attendees function. Similarly, cost and revenue must be defined before profit.

evaluate: generate the tests from the problem statement first.

Run the program!

```plaintext
Running atts at $5.00...
  -> success
Running atts at $4.90...
  -> failure
...stopping on first failure
Process ended with exit value 1
```

• One of our test cases for attendees failed...
• Debugging reveals that we mis-parenthesized the arithmetic
• Here is the fixed version:

```plaintext
let attendees (price:int) : int = 120 + (500 – price) / 10 * 15
```

Success!

```plaintext
Running atts at $5.00...
  -> success
Running atts at $4.90...
  -> success
Running profit at $5.00...
  -> success
Running profit at $4.90...
  -> success
Process ended with exit value 0
```
How not to Solve this Problem

let profit price = (120 + (15 * (500 - price) / 10)) * price - (18000 + 4 * (120 + (15 * (500 - price) / 10)))

This program is bad because it
– hides the structure and abstractions of the problem
– duplicates code that could be shared
– doesn’t document the interface via types and comments

*Note that this program still passes all the tests!*

Evolving/Refactoring Code

• For this simple problem, this design methodology may seem like overkill.
  – The real benefits are to be had in bigger programs
  – But, even small programs evolve over time

• Suppose that, based on the problem description, you decided to define cost in terms of the number of attendees directly, rather than calling the attendees from within cost.
  – How do our tools and this design methodology help?

Refactoring 1: Change ‘cost’

cost is simplified:

(* atts is the number of attendees *)
let cost (atts:int) : int = 18000 + 4 * atts

... but suppose we forget to change profit, which calls cost. (As might easily happen in a big program.)

Test Case for Profit Fails

• We need to fix profit like this:

(* price is in cents *)
let profit (price:int) : int = (revenue price) - (cost (attendees price))
Any questions?

Essential OCaml Cheatsheet

The next few slides summarize the part of OCaml that we’ve seen so far. (We’ll add a few more things as we go along.)

Caveat

Many people find programming in OCaml a little disorienting at first. The syntax is unfamiliar, but more importantly OCaml embodies a value-oriented programming style that takes a little while to get used to.

For the moment, we ask you to trust that this is all going to feel much more natural in a couple of weeks and enjoy the challenge of learning to think about programming a little differently.

In particular, please stick to just the features we’re showing you, when you’re working on the homework assignment...

The Big Picture

For the moment, we can think of an OCaml program as a sequence of declarations and tests.

When the program is executed, all the tests get run. If they are all successful, the program simply terminates. If any are unsuccessful, a failure message is printed.

Yes, we are saying that a correct program does nothing at all!
The Big Picture

A declaration can be either a variable declaration or a function declaration.

```
let x = 100
let f (k:int) : int = k * 5 + x
let y = f 42
```

The scope of each declaration is the remainder of the program after the point where it occurs.

Unlike many other languages, variables and functions can only be used after they are declared.

Using Tests

Modern approaches to software engineering advocate test-driven development, where tests are written very early in the programming process and used to drive the rest of the process.

We are big believers in this philosophy, and we’ll be using it throughout the course.

In the homework template, we’ve provided one or more tests for each of the problems. You should start each problem by making up some more tests.

Tests

Tests are calls to the assert_eq function:

```
;; assert_eq "Attendees at $5.00" (attendees 500) 120
;; assert_eq "Attendees at $4.90" (attendees 490) 135
```

The arguments are:
- a string describing the test
- an expression to be tested
- the expected result

Each test is introduced by a double-semicolon.

These are the only places that semicolons should appear in your programs so far. (We’ll see one more place in a little while.)

Primitive Types and Constants

OCaml’s built-in primitive types include...

- int  
  0, 1, 42, -1, 999
- string  
  “hello world”
- bool  
  true, false
Expressions

Numeric expressions:

- `1 + 2` addition
- `1 - 2` subtraction
- `2 * 3` multiplication
- `10 / 3` integer division
- `10 mod 3` modulus (remainder)

Standard rules of operator precedence:

- `1 + (2 * 3) = 7`
- `1 + 2 * 3 = 7`
- `(1 + 2) * 3 = 9`

Comparisons:

- `=` equality
- `<>` inequality
- `<` less than
- `>=` greater than or equal

Boolean (logical) operators:

- `not` logical negation
- `& &` and
- `||` or

Function Declarations

```
let total_secs (hours:int) (minutes:int) (seconds:int) : int =
  (hours * 60 + minutes) * 60 + seconds
```

Function Calls

Once a function has been declared, it can be invoked by writing the function name followed by a list of arguments.

```
total_secs 5 30 22
```

(Note that the list of arguments is not parenthesized.)
### Conditional Expressions

- **if** `s = “positive” then 1 else -1
- **if** `day >= 6 && day <= 7` then “weekend” else “weekday”

OCaml conditionals are *expressions*: they can be used inside of other expressions:

- `(if x > 0 then x else –x) * 100
- **if** `x > y` then “x is bigger” else if `x < y` then “y is bigger” else “same”

### Variable Declarations

A variable declaration gives a name to the result of some expression.

- `let pi = 3.14159`  
  `let seconds_per_day = 60 * 60 * 24`

Note that there is no way of *assigning* a new value to a variable after it is declared.

### Scope

The scope of each top level declaration is the rest of the program after the point where it occurs.

Unlike some other languages, variables and functions in OCaml can only be used after they are declared.

### Variable Declarations

Variable declarations can appear both at top-level and *nested* within other expressions.

- `let f (x:int) : int =`  
  `let y = x * 10 in`  
  `y * y`

  ;; assert_eq “” (f 3) 900

Nested variable declarations are followed by “in”. Top-level variable declarations are not.
Scope

Multiple declarations of the same variable or function name are allowed. The later declaration *shadows* the earlier one for the rest of the program.

```ocaml
let x = 1
let y = x + 1
let x = 1000
let z = x + 2
;; assert_eq "" z 1002
;; assert_eq "" y 2
```

Modules

An OCaml program is a set of *modules*. Each module lives in its own file. A module named `foo` lives in a file named `foo.ml`.

Each module consists of a sequence of *declarations*. A declaration can be either a *variable declaration* or a *function declaration*.

```ocaml
let x = 100
let s = "hello world"
let f (k:int):int = k * 5 + x
```

Modules

If you are writing a module M and you need to refer to a name x from another module N, either prefix it with the module name and a dot (N.x) or else write "open N" at the top of m.ml and then write just x to refer to N.x.

Value-Oriented Programming

OCaml embodies a value-oriented (or declarative) style of programming.

- In the fragment of the language we’re using so far, the only construct that actually *does* anything is the `assert_eq` command, which runs a test and, if it fails, prints something.
Structured Data

A New Twist

- The design problem we looked at last time involved relationships among atomic values — simple numbers.
- Some real-world programs live in this simple world — e.g., the first one ever run in this building!
- But most interesting programs need to work with collections of data — sets, lists, tables, databases, ...

Tuples

- The simplest form of structured data in OCaml is the tuple.
- A tuple is a way of grouping together two or more data values

Varieties of Collections

- There are many ways of representing collections, depending on what operations we need.
  - create new collections
    - in particular, new empty ones
  - add to a collection
    - at the front
    - at the back
    - at an arbitrary position
  - take a collection apart and see what’s in it
    - get elements off the front, one by one
    - get elements off the back
    - get elements by position (random access)
    - get elements in any order
    - get elements in ascending order
  - change elements in place
  - etc.
A Simple Collection: Immutable Lists

Let’s begin with one of the very simplest forms of collection: immutable lists.

Operations:
- build a list
- add an element to the front of an existing list
- divide a list into front element and rest of the list

Building Lists

To build a list, simply write down its elements, enclosed in square brackets and separated by semicolons:

- \([1; 2; 3; 5]\) a list of four numbers
- \(["hello";"world"]\) a list of two strings
- \([true;false;true]\) a list of three booleans
- \([\ ]\) the empty list

List Types*

The type of lists of integers is written

\[\text{int \ list}\]

The type of lists of strings is written

\[\text{string \ list}\]

The type of lists of booleans is written

\[\text{bool \ list}\]

The type of lists of lists of strings is written

\[\text{string \ list \ list}\]

etc.

*Note that lists in OCaml are homogeneous—all of the list elements must be of the same type. If you try to create a list like \([1;"hello";true]\) you will get a type error.

Adding Elements

If mylist is a list of integers and \(x\) is an integer, then

\[x :: \text{mylist}\]

(pronounced “\(x\) cons mylist”) is a list whose first element is \(x\) and the rest of whose elements come from mylist.

\[
\text{let mylist} = [1; 2; 3] \\
\text{let anotherlist} = 0 :: \text{mylist} \\
\text{;;; assert_eq "" another_list [0;1;2;3]}
\]
Pattern Matching

OCaml provides a single expression for destructing lists, called pattern matching.

```ocaml
let mylist = [1; 2; 3; 5]
let y = begin match mylist with
  | [] -> 42
  | first::rest -> first+10
end
;; assert_eq "" y 11
```

A Design Problem / Situation

Suppose we have a friend who is opening a toy-store, and she wants some help computerizing her inventory. She wants to be able to do things like check how much total inventory she has, remove items from the inventory or replace one item with another, make lists of different kinds of items (dolls, games, etc.), etc.

To keep things simple, let’s start by working with just the names of the toys...