Programming Languages and Techniques (CIS120)

Lecture 3

Jan 13, 2013

Value-Oriented Programming
Lists and Recursion

Announcements

- Homework 1: OCaml Finger Exercises
  - Due: Tuesday, Jan 22nd at 11:59:59pm (midnight)
- Please *read* Chapter 1-3 of the course notes, which are available from the course web pages.
- Lab topic this week: *Debugging OCaml programs*

- TA office hours: on webpage (calendar) and on Piazza
- Questions?
  - Post to Piazza, privately if you need to include code
  - My drop-by office hours: 3:30-5PM today

Value-Oriented Programming in OCaml

See also Chapter 2 of the CIS 120 lecture notes available from the web pages.

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.

Value-Oriented Programming

We run programs by *calculating* expressions to values:

```
3 ⇒ 3                values compute to themselves
3 + 4 ⇒ 7
2 * (4 + 5) ⇒ 18
true && (false || true) ⇒ true
```

The notation <exp> ⇒ <val> means that the expression <exp> computes to the value <val>.

Note: the symbol '⇒' is *not* OCaml syntax. It's a convenient way to *talk* about OCaml syntax.

Primitive Values

OCaml's built-in primitive types of values include…

- int

  0, 1, 42, -1, 999

- float

  3.14159, 0.123

- string

  "hello world"

- bool

  true, false

- In the next few weeks, we will introduce many more value forms, built from structured data.

Expressions

Numeric expressions (ints):

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

From constants and operations, we can build bigger expressions:

```
(1 + 2 * (10 mod 4)) / 4
```

*These operators can only be used with ints. Floating point operators have a . after them.

Step-wise Calculation

- We can understand ⇒ in terms of single step calculations written '$\longmapsto$'
  - Single step calculations do "the expected thing" for primitive operations
- For example:

`(2+3) * (5-2)`	
$\longmapsto$ `5 * (5-2)`	because 2+3 $\longmapsto$ 5
$\longmapsto$ `5 * 3`	because 5-2 $\longmapsto$ 3
$\longmapsto$ `15`	because 5*3 $\longmapsto$15

Operators

Comparisons:

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

(these can be used with any type of data – numbers, strings, characters, etc.)

Boolean (logical) operators:

not	logical negation
&&	and
||	or

(These can only be used with boolean values. Most operators in OCaml only work for a single type of argument.)

String operators:

^	string concatenation

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 3 > 0 then 2 else –1) * 100
```

```
if x > y then "x is bigger"
else if x < y then "y is bigger"
else "same"
```

Running Conditional Expressions

- A conditional expression yields the value of either its 'then'-branch expression or its 'else'-branch expression, depending on whether the test is 'true' or 'false'.
- For example:

```
    (if 3 > 0 then 2 else –1) * 100
⟼  (if true then 2 else -1) * 100
⟼  2 * 100
⟼  200
```

- Note: this means that it's not sensible to leave out the 'else' branch. (What would be the result if the test was 'false'?)

(Top-level) Let Declarations

A let declaration gives a *name* (a.k.a. an *identifier*) 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 an identifier after it is declared.

*We might sometimes call these identifiers *variables*, but the terminology is a bit confusing because in languages like Java and C a variable is something that can be modified over the course of a program. In OCaml, like in mathematics, once a variable's value is determined, it can never be modified… As a reminder of this difference, for the purposes of OCaml we'll try to use the word "identifier" when talking about the name bound by a let.

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.

```
let x = 1
let y = x + 1
let x = 1000
let z = x + 2
let test () : bool =
    z = 1002
;; run_test "x shadowed" test
```

scope of x

scope of y

scope of x (shadows earlier x)

scope of z

Evaluating Let Declarations

To calculate the value of a let declaration, first calculate the value of the right hand side and then substitute that value for the identifier in its scope:

```
let x = 1
let y = x + 1
let x = 1000
let z = x + 2
let test () : bool =
    z = 1002
;; run_test “x shadowed” test
```

Evaluating Let Declarations

To calculate the value of a let declaration, first calculate the value of the right hand side and then substitute that value for the identifier in its scope:

```
let x = 1
let y = 1 + 1
let x = 1000
let z = x + 2
let test () : bool =
    z = 1002
;; run_test "x shadowed" test
```

1 ⇒ 1, so substitute 1 for x in x's scope

note that this occurrence doesn't change

Evaluating Let Declarations

To calculate the value of a let declaration, first calculate the value of the right hand side and then substitute that value for the identifier in its scope:

```
let x = 1
let y = 2
let x = 1000
let z = x + 2
let test () : bool =
    z = 1002
;; run_test "x shadowed" test
```

1+1 ⇒ 2, so substitute 2 for y in y's scope (there are no occurrences of y)

Evaluating Let Declarations

To calculate the value of a let declaration, first calculate the value of the right hand side and then substitute that value for the identifier in its scope:

```
let x = 1
let y = 2
let x = 1000
let z = 1000 + 2
let test () : bool =
    z = 1002
;; run_test "x shadowed" test
```

1000⇒1000, so substitute 1000 for x in this x's scope

This 'x' is part of the string…it doesn't change.

Evaluating Let Declarations

To calculate the value of a let declaration, first calculate the value of the right hand side and then substitute that value for the identifier in its scope:

```
let x = 1
let y = 2
let x = 1000
let z = 1002
let test () : bool =
    1002 = 1002
;; run_test "x shadowed" test
```

1000+2⇒1002, so substitute 1002 for z in its scope

Local Let Declarations

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

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

let test () : bool =
   (f 3) = 900
;; run_test "test f" test
```

scope of x is the body of f

scope of y is nested within the body of f

scope of f is the rest of the program

Nested let declarations are followed by "in".

Top-level let declarations are not.

Top-level Declarations

A top-level declaration can be either an *identifier declaration* or a *function declaration.*

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

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

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

Function Declarations

function name parameter names parameter types

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

result type

function body (an expression)

Function Calls

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

```
total_secs 5 30 22
```

(Note that the list of arguments is *not* parenthesized.)

Calculating With Functions

- To calculate the value of a function application, first calculate values for its arguments and then *substitute* them for the parameters in the body of the functions.

```
   total_secs (2 + 3) 12 17
⟼ total_secs 5 12 17          because 2+3⟼5
⟼ (5*60 + 12) * 60 + 17      subst. the args. in the body
⟼ (300 + 12) * 60 + 17
⟼ 312 * 60 + 17
⟼ 18720 + 17
⟼ 18737
```

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

Test Commands

Tests *always* follow the same pattern:

```
let test () : bool =
    (attendees 500) = 120
;; run_test "Attendees at $5.00" test

let test () : bool =
    (attendees 490) = 135
;; run_test "Attendees at $4.90" test
```

The arguments are:

- an expression to be tested
- the expected result
- a string describing the test

The run_test command (like all commands) is prefixed by a double-semicolon.

Such commands are the *only* places that semicolons should appear in your programs (so far).

Structured Data

A Design Problem / Situation

Suppose we have a friend who has a lot of digital music, and she wants some help with her playlists.

She wants to be able to do things like check how many songs are in a playlist, check whether a particular song is in a playlist, check how many Lady Gaga songs are in a playlist, and see all of the Lady Gaga songs in a playlist, etc.

She might want to *remove* all the Lady Gaga songs from her collection.

Design Pattern

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 behavior

    Often by decomposing the problem into simpler ones and applying the same recipe to each

1. Understand the problem

How do we store and query information about songs?

Important concepts are:

1. A playlist (a collection of songs)
2. A fixed collection of *gaga_songs*
3. Counting the *number_of_songs* in a playlist
4. Determining whether a playlist *contains* a particular song
5. Counting the *number_of_gaga_songs* in a playlist
6. Calculating *all_gaga_songs* in a playlist
7. Calculating *all_non_gaga_songs* in a playlist

2. Formalize the interface

- Represent a song by a *string* (which is its name)
- Represent a playlist using an *immutable list of strings*
- Represent the collection of Lady Gaga Songs using a *toplevel definition*
- Define the interface to the functions:

```
let number_of_songs (pl : string list) : int =
let contains (pl : string list) (song : string) : bool =
let number_of_gaga_songs (pl : string list) : int =
let all_gaga_songs (pl : string list) : string list =
let all_non_gaga_songs (pl : string list) : string list =
```

List Types*

The type of lists of integers is written

```
int list
```

The type of lists of strings is written

```
string list
```

The type of lists of booleans is written

```
bool list
```

The type of lists of lists of strings is written

```
(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”; 3; true] you will get a type error.

What is a list?

- A list is either:

`[]`	the *empty* list, sometimes called *nil*
or	
`v::tail`	a *head* value v, followed by a list of the remaining elements, the *tail*

- Here, the '`::`' operator *constructs* a new list from a head element and a shorter list.
  - This operator is pronounced "cons" (for "construct")

- Importantly, *there are no other kinds of lists*.

Example Lists

To build a list, cons together elements, ending with the empty list:

`1::2::3::4::[]`	a list of four numbers
`“abc”::”xyz”::[]`	a list of two strings
`true::[]`	a list of one boolean
`[]`	the empty list

Explicitly parenthesized

'`::`' is an ordinary operator like + or ^, except it takes an element and a *list* of elements as inputs:

`1::(2::(3::(4::[])))`	a list of four numbers
`"abc"::("xyz"::[])`	a list of two strings
`true::[]`	a list of one boolean
`[]`	the empty list

Convenient List Syntax

Much simpler notation: enclose a list of elements in `[` and `]` separated by `;`

`[1;2;3;4]`	a list of four numbers
`[“abc”;”xyz”]`	a list of two strings
`[true]`	a list of one boolean
`[]`	the empty list

Calculating With Lists

- Calculating with lists is just as easy as calculating with arithmetic expressions:

$(2+3)::(12\ /\ 5)::[]$

$\longmapsto 5::(12\ /\ 5)::[]$ because $2+3 \Rightarrow 5$

$\longmapsto 5::2::[]$ because $12/5 \Rightarrow 2$

A list is a value whenever all of its elements are values.

3. Write test cases

```
let pl1 : string list = [ "Bad Romance"; "Nightswimming";
          "Telephone"; "Everybody Hurts" ]
let pl2 : string list = [ "Losing My Religion";
          "Man on the Moon"; "Belong" ]
let pl3 : string list = []

let test () : bool =
  (number_of_songs pl1) = 4
;; run_test "number_of_songs pl1" test

let test () : bool =
  (number_of_songs pl2) = 3
;; run_test "number_of_songs pl2" test

let test () : bool =
  (number_of_songs pl3) = 0
;; run_test "number_of_songs pl3" test
```

Define playlists for testing. Include some with and without Gaga songs as well as an empty list.