Programming Languages and Techniques (CIS120)

Lecture 3

Jan 18, 2012

Lists and Recursion

Announcements

- Homework 1: OCaml Finger Exercises
  - Due: Monday, Jan. 23rd at 11:59:59pm (midnight)
- Please *read* Chapter 1-3 of the course notes, which is 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 need to include code
  - Drop by office hours
    (Weirich, 3:20-5PM today and 4-5PM Friday)

Let Declarations

A let declaration gives a *name* (a.k.a. an *identifier*) to the result of some expression*.

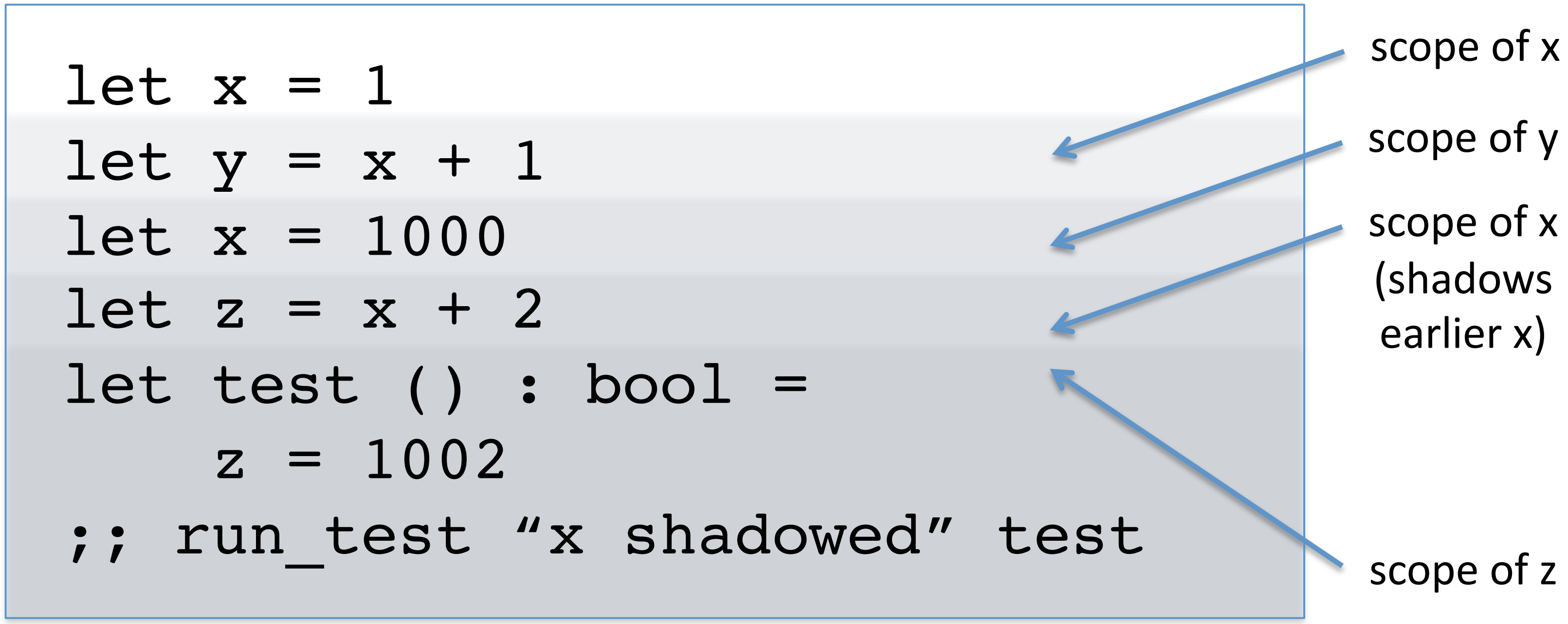
```
let pi : float = 3.14159
let seconds_per_day : int = 60 * 60 * 24
```

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
```

Evaluating Let Expressions

To calculate the value of a let expression, 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 Expressions

To calculate the value of a let expression, 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 Expressions

To calculate the value of a let expression, 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 Expressions

To calculate the value of a let expression, 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 Expressions

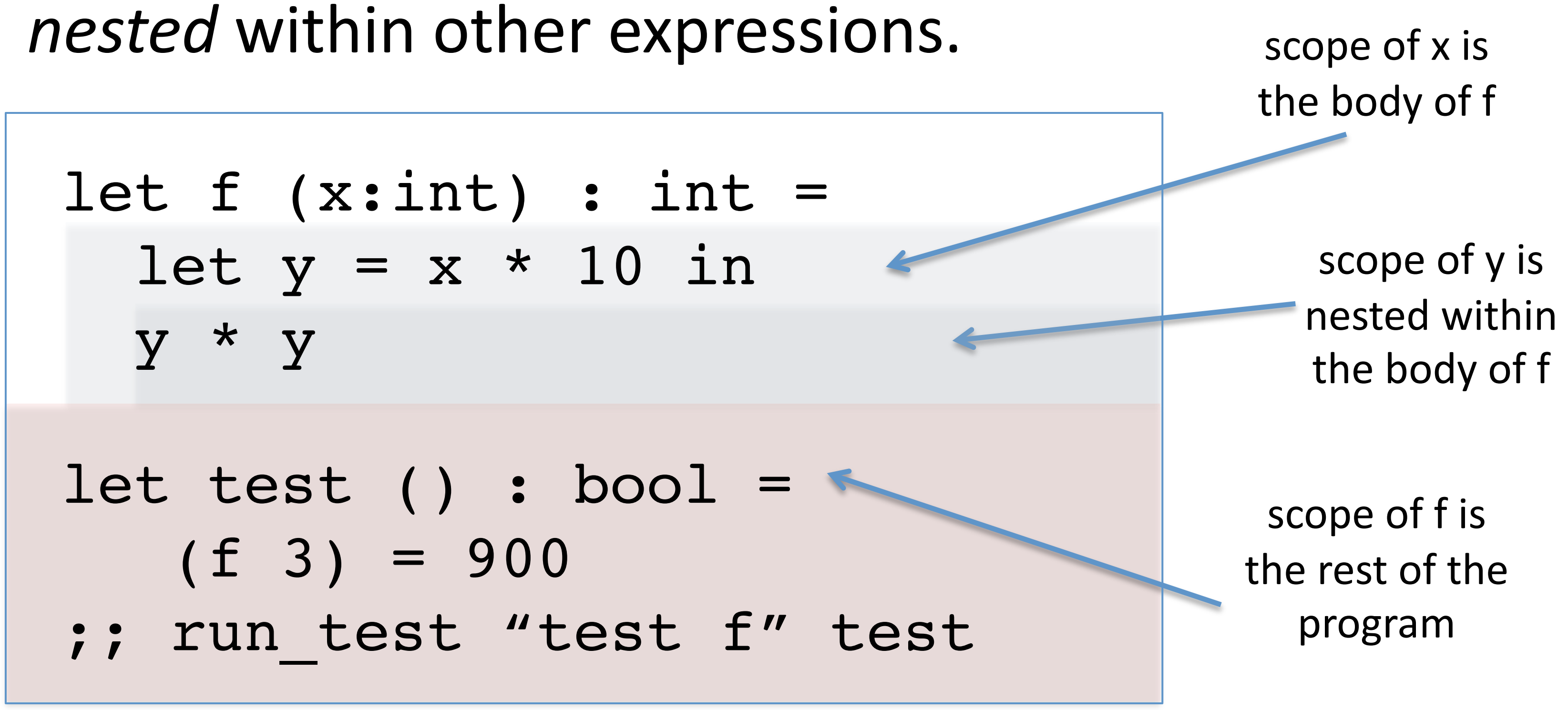
To calculate the value of a let expression, first calculate the value of the right hand side and then substitute that value for the identifier in its scope:

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

$1000+2 \Rightarrow 1002$, so substitute 1002 for z in its scope

Local Let Declarations

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



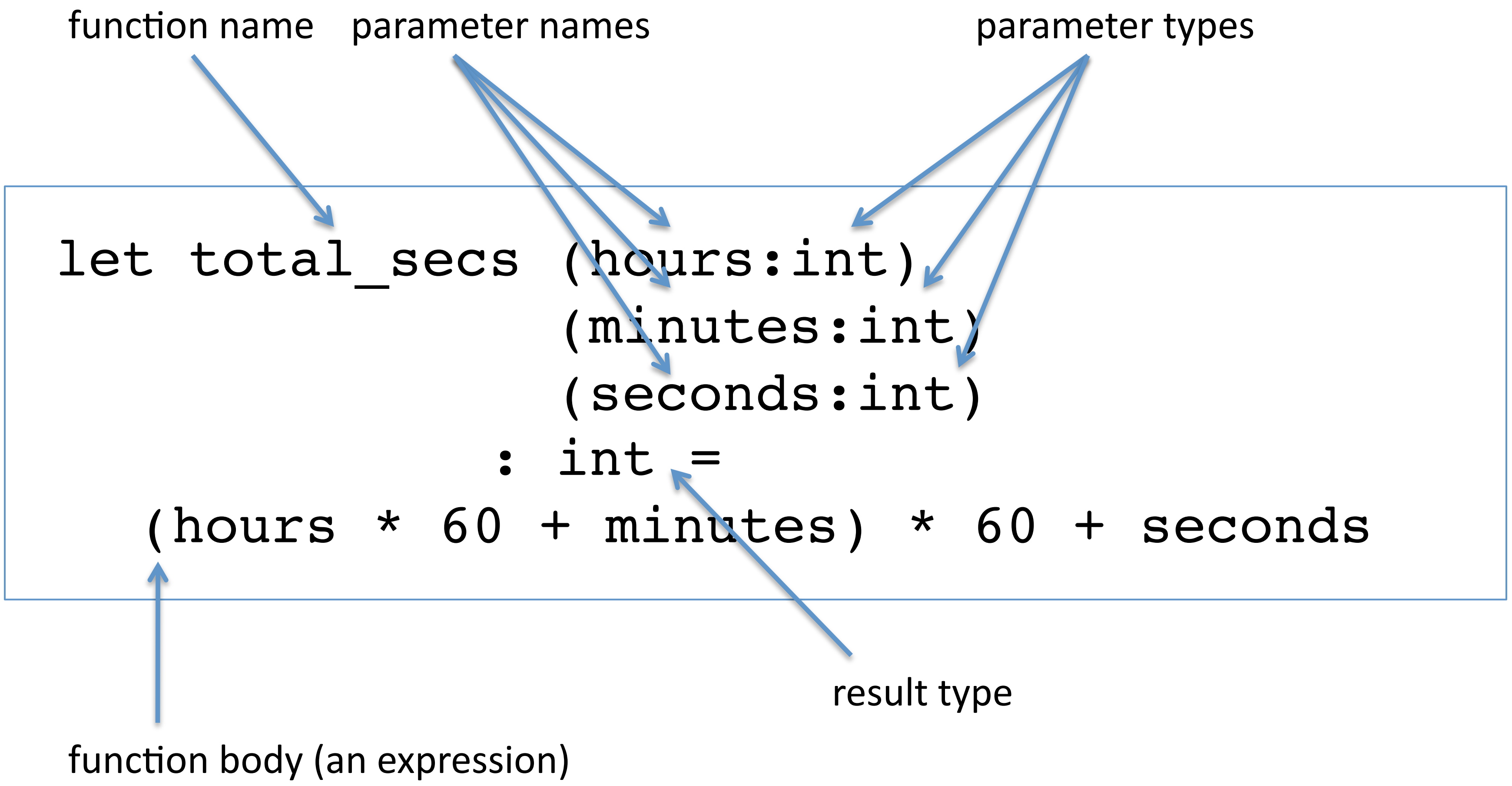
```
let f (x:int) : int =
  let y = x * 10 in
  y * y

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.

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. 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
⟼ (5*60 + 12) * 60 + 17
⟼ (300 + 12) * 60 + 17
⟼ 312 * 60 + 17
⟼ 18720 + 17
⟼ 18737
```

because 2+3⟼5

subst. the args. in the body

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

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, …

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 of all of the Lady Gaga songs in a playlist, etc. She might want to *remove* all the Lady Gaga songs from here 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

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 of all of the Lady Gaga songs in a playlist, etc.

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. 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

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*

```
let gaga_songs : string list = [ "Bad Romance"; … ]
```

- 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 =
```

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.

4. Implement the behavior

The function calls itself *recursively* so the function declaration must be marked with rec.

Lists are either empty or nonempty. *Pattern matching* determines which.

```
let rec number_of_songs (pl : string list) : int =
  begin match pl with
  | [] -> 0
  | ( song :: rest ) -> 1 + number_of_songs rest
  end
```

If the lists is non-empty, then “song” is the first song of the list and “rest” is the remainder of the list.