Programming Languages and Techniques (CIS120)

Lecture 4

Jan 16, 2013

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

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 =
```

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.

Interactive Interlude

gaga.ml

Lists

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*.

Pattern Matching

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

```
let mylist : int list = [1; 2; 3; 5]

let y =
  begin match mylist with
  | [] -> 42
  | first::rest -> first+10
  end
```

match expression syntax is:

```
begin match … with
  | … -> …
  | … -> …
end
```

case branches

This case analysis is justified because there are only *two* shapes that a list can have.

Note that `first` and `rest` are identifiers that are bound in the body of the branch.

Calculating with Matches

- Consider how to run a match expression:

```
begin match [1;2;3] with
   | [] -> 42
   | first::rest -> first + 10
end
```

⟼

1 + 10

⟼

11

Note: `[1;2;3]` equals `1::(2::(3::[]))`

It doesn't match the pattern [] so the first branch is skipped, but it *does* match the pattern `first::rest` when `first` is 1 and `rest` is `(2::(3::[]))`.
So, substitute 1 for `first` in the second branch

Using Recursion Over Lists

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.

Patterns specify the **structure** of the value and (optionally) give **names** to parts of it.

Calculating with Recursion

```
number_of_songs ["Monster";"Teeth"]
```

⟼ *(substitute the list for pl in the function body)*

```
begin match "Monster"::("Teeth"::[]) with
  | [] -> 0
  | (song :: rest) -> 1 + (number_of_songs rest)
end
```

⟼ *(second case matches with rest = "Teeth"::[]*

```
1 + (number_of_songs "Teeth"::[])
```

⟼ *(substitute the list for pl in the function body)*

```
1 + (begin match "Teeth"::[] with
       | [] -> 0
       | (song :: rest) -> 1 + (number_of_songs rest)
     end
```

⟼ *(second case matches again, with rest = [])*

```
1 + (1 + number_of_songs [])
```

⟼ *(substitute [] for pl in the function body)*

…

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