Programming Languages and Techniques (CIS120)

Lecture 4
Jan 16, 2013

Lists and recursion
Announcements

• Homework 1: OCaml Finger Exercises
  – Due: Tuesday, Jan 22\textsuperscript{nd} at 11:59:59pm (midnight)
• Please \textit{read} Chapter 1-3 of the course notes, which are available from the course web pages.
• Lab topic this week: \textit{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 \textit{gaga\_songs}
3. Counting the \textit{number\_of\_songs} in a playlist
4. Determining whether a playlist \textit{contains} a particular song
5. Counting the \textit{number\_of\_gaga\_songs} in a playlist
6. Calculating \textit{all\_gaga\_songs} in a playlist
7. Calculating \textit{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:

```plaintext
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

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

```ocaml
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
```
Interactive Interlude

gaga.ml
Lists
What is a list?

• A list is either:
  ```
  []
  
  or
  v::tail
  ```

  the *empty* list, sometimes called *nil*

• 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.*
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
```

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:

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

= 1 + 10

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 \textit{recursively} so the function declaration must be marked with \texttt{rec}.

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

Lists are either empty or nonempty. \textit{Pattern matching} determines which.

Patterns specify the \texttt{structure} of the value and (optionally) give \texttt{names} to parts of it.
Calculating with Recursion

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

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

```
(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"::[])
```

```
(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)
```

```
...