Discussion 03

Pattern Matching

Kenneth Fang (kwf37), Newton Ni (cn279)

Feb. 3, 2019

Agenda

1. Review concept of pattern matching
2. Explore pattern matching with three different data structures:
   - Lists v1::v2:: ... ::vn::[]
   - Records {label1=v1, label2=v2, ..., labeln = vn}
   - Tuples (v1,v2,...,vn)
3. Recitation 4

What is Pattern Matching?

What is Pattern Matching?

- How do we access data in arrays in an object-oriented language?

What is Pattern Matching?

- How do we access data in arrays in an object-oriented language?
- How do we access data in objects in an object-oriented language?

What is Pattern Matching?

- How do we access data in arrays in an object-oriented language?
- How do we access data in objects in an object-oriented language?
- What if we extracted data by leveraging the *structure* of the data?

What is Pattern Matching?

- How do we access data in arrays in an object-oriented language?
- How do we access data in objects in an object-oriented language?
- What if we extracted data by leveraging the *structure* of the data?
- When pattern matching, we can ensure that our data accesses are exhaustive and that every branch in the pattern match is being used

Lists

Lists in OCaml are

- Singly-linked lists
- Immutable
- “First-class” data structures

Lists

Lists are defined to be either

- Nil `[]`
- Cons `h::t`

Lists

Lists are defined to be either

- Nil `[]`
- Cons `h::t`

Lists can be constructed using the following syntax:

- `[]`
- `e1::e2::e3::[]`
- `[e1;e2;e3]` (syntactic sugar for above syntax)

Lists in OCaml are

List Static Semantics

- The empty list Nil has type `'a list`

List Static Semantics

- The empty list Nil has type `'a list`
- The list `e1::[]` has type `t list` if `e1 : t`

List Static Semantics

- The empty list Nil has type `'a list`
- The list `e1::[]` has type `t list` if `e1 : t`
- All elements in a list must have the same type

List Static Semantics

- The empty list Nil has type `'a list`
- The list `e1::[]` has type `t list` if `e1 : t`
- All elements in a list must have the same type
- For the cons operator `h::t`, if `h:typ`, then it must be true that `t:typ list`

List Static Semantics

- The empty list Nil has type `'a list`
- The list `e1::[]` has type `t list` if `e1 : t`
- All elements in a list must have the same type
- For the cons operator `h::t`, if `h:typ`, then it must be true that `t:typ list`
- What is the type of the cons operator (::)?

List Pattern Matching

Typically a list can be broken down as follows:

```
match lst with
| [] -> (*Do something when list is empty*)
| h::t -> (*Do something with head or tail*)
```

List Length

```
let rec length lst =
```

List Length

```
let rec length lst =
   match lst with
   | [] -> 0
   | h::t -> 1 + (length t)
```

List Length With Syntactic Sugar and Wildcard

```
let rec length = function
      | [] -> 0
      | _::t -> 1 + (length t)
```

Sum Last Two Elements of (Int) List

```
let rec sum_last_two = function
```

Sum Last Two Elements of (Int) List

```
let rec sum_last_two = function
        | x1::x2::[] -> x1 + x2
        | _::t -> sum_last_two t
        | _ -> raise LengthException
```

Records (By Name)

We can define record *types* that have multiple fields and then create record *expressions* that have that type. Data fields are structured by name.

Records (By Name)

We can define record *types* that have multiple fields and then create record *expressions* that have that type. Data fields are structured by name.

- Record type definition:

```
type student = {name:string; age:int; is_sleepy:bool}
```

Records (By Name)

We can define record *types* that have multiple fields and then create record *expressions* that have that type. Data fields are structured by name.

- Record type definition:

  ```
  type student = {name:string; age:int; is_sleepy:bool}
  ```

- Record expression:

  ```
  let kenneth = {name=kenneth; age=20; is_sleepy=true}
  ```

Records (By Name)

We can define record *types* that have multiple fields and then create record *expressions* that have that type. Data fields are structured by name.

- Record type definition:

  ```
  type student = {name:string; age:int; is_sleepy:bool}
  ```

- Record expression:

  ```
  let kenneth = {name=kenneth; age=20; is_sleepy=true}
  ```

- Record expression using `with` keyword:

  ```
  {let newton = kenneth with name=newton; age=21}
  ```

Records (By Name)

How do we access data?

- Method 1: Dot Notation

  ```
  kenneth.name
  ```

- Method 2: Pattern Matching

  ```
  match kenneth with
  | {name=n;age=x;is_sleepy=s} -> n
  ```

Tuples (By Position)

Tuples are also data structures that have multiple fields, but they are not labelled. Instead, data is structured based on the *position*.

- Type definition:
  - Tuple type definition:

    ```
    type student = string * int * bool
    ```

Tuples (By Position)

Tuples are also data structures that have multiple fields, but they are not labelled. Instead, data is structured based on the *position*.

- Type definition:
  - Tuple type definition:

    ```
    type student = string * int * bool
    ```
  - Tuple expression:

    ```
    let kenneth = (kenneth, 20, true)
    ```

Tuples (By Position)

How do we access data?

- Method 1: Pattern Matching

```
match kenneth with
| (name, age, is_sleepy_boi) -> name
```

- The standard library comes with the `fst` and `snd` functions, which can be used to extract the first and second fields of a tuple, respectively.

List Equality

```
let rec list_equals l1 l2 =
```

List Equality

```
let rec list_equals l1 l2 =
        match (l1,l2) with
        | ([],[]) -> true
        | (h1::t1, h2::t2) when h1 = h2 -> list_equals t1 t
        | _ -> false
```

Recitation Questions