COMP 302: Programming Languages and Paradigms
Prof. B. Pientka, McGill University

Week 2 (Part 1): Data Types and Pattern Matching

Often it is useful to define a collection of elements or objects and not encode all data we are working with using our base types of integers, floats or strings. For example, we might want to model a simple card game.

As a first step, we define the *suit* in the game. In OCaml, we define a finite, unordered collection of elements using a (non-recursive) data type definition.

Example

```
type suit = Clubs | Spades | Hearts | Diamonds
```

all possible form of that type

Note:

- `Clubs`, `Spades`, `Hearts`, and `Diamonds` are constructors.
- Order in which we specify the elements does not matter.
- In OCaml, each element begins with a capital letter.

Write a function `dom`**.** It takes in a pair `s1` and `s2` of `suit`. If `s` beats of is equal to suit `s2` we return true – otherwise we return false. We will use the following ordering on suits:

$$\texttt{Spades} > \texttt{Hearts} > \texttt{Diamonds} > \texttt{Clubs}$$

What is the type of the function `dom`?

The type of the function `dom` is `suit * suit -> bool`.

How do we write programs about elements of type `suit`?

The answer is *pattern matching* using a match-expression.

```
match <expression> with
| <pattern> -> <expression>
| <pattern> -> <expression>
 ...
| <pattern> -> <expression>
```

Last Step: Writing the `dom` function using pattern matching

```
(* dom : suit * suit -> bool *)
let rec dom (s1, s2) = match (s1, s2) with
  | (Spades, _)         -> true
  | (Hearts, Diamonds)  -> true
  | (Hearts, Clubs)     -> true
  | (Diamonds, Clubs)   -> true
  | (s1, s2)            -> s1 = s2
```

order

What are patterns?

A pattern `pat` of type `T` can be characterized as follows:

- A pattern variable `x` (or `y`, `s1`, `s2`, etc.) is a pattern of type `T`.
- A wild card `_` is a pattern of type `T`.
- A constant `c` of type `T` is a pattern of type `T`.
- A pair `(pat1 , patt2)` is a pattern of type `T * S`, if `pat1` is a pattern of type `T` and `pat2` is a pattern of type `S`.

What is the operational behavior of writing a program using pattern matching?

An even compacter way of writing `dom`

order ?

```
let rec dom (s1, s2) = match (s1, s2) with
  | (Spades, _)     | (Hearts, Diamonds)
  | (Hearts, Clubs) | (Diamonds, Clubs)  -> true
  | (s1, s2)                            -> s1 = s2
```

More on patterns Patterns are a flexible and compact way to compare data. In particular, we can write deep patterns. For example, let's build a tuple of type `(suit * int) * (suit * int)` where we pair every suit with an integer to describe its value and we now want to add up the values of matching suits and return `0` otherwise.

```
let rec add c1 c2 = match c1, c2 with
| (Hearts,   v1), (Hearts , v2)
| (Diamonds, v1), (Diamonds,v2)
| (Spades,   v1), (Spades,  v2)
| (Clubs, v1)   , (Clubs,   v2) -> v1 + v2
| _             , _             -> 0
```

Imagine writing this program with if-expressions in the traditional way! This will be long, cumbersome and difficult to read.

Let's continue with our example. We define the rank of cards similarly to the suit of cards introducing a new type `cards`.

```
type rank =
 Two | Three | Four | Five | Six | Seven | Eight | Nine | Ten |
 Jack | Queen | King | Ace
```

To describe a card, we introduce a *type definition* or *abbreviation*.

```
type card = rank * suit
```

For now this allows us to write in our type specification simply `card` instead of `rank * suit` and makes type specifications easier to read.

Recursive Data Types To actually play a simple card game, we also need to be able to describe a collection of cards we may hold in our hand. This is not a finite collection – we may hold no cards, one card, or any number of cards! In our setting, we do not want to restrict the number of cards we can hold, as this may also differ from game to game.

Defining a `hand` We can say that a hand (i.e. the collection of cards we hold in a given hand) is either empty or it consists of a card followed by the rest of the hand.

Turning this inductive definition into a recursive data type We will now define a recursive data type `hand` that characterizes the collection of cards we hold more precisely:

- The constant `Empty` is of type `hand`. It describes the empty collection of cards one holds in a given hand.
- If `c` is a `card` and `h` is of type `hand` then `Hand (c, h)` is of type `hand`.
  constructor
- Nothing else is of type hand.

In OCaml, we can define the recursive data types `hand` as follows:

```
type hand = Empty | Hand of card * hand
```

recursion

Examples Here are some examples of hands of type `hand` we can construct.

```
let hand0:hand = Empty
let hand1:hand = Hand((Ace, Hearts), Empty)
let hand2:hand = Hand((Queen, Diamonds), hand1)
let hand5:hand = Hand((Ace, Spades),
                      Hand((Ten, Diamonds),
                           Hand((Seven, Clubs),
                                Hand((Queen, Spades),
                                     Hand((Eight, Clubs), Empty)))))
```

How can we write functions that take a `hand` as an input?

Let's see how we use pattern matching on elements of type `hand`.

We want to write a function `extract` of type `suit -> hand -> hand` which given a `suit` and a `hand` it extracts from the input `hand` all those cards that match the given suit and returns a new `hand` containing only those cards.

To illustrate here is the expected behaviour when we execute the function `extract`

```
# extract Spades hand5;;
- : hand = Hand ((Ace, Spades), Hand ((Queen, Spades), Empty))
# extract Diamonds hand5;;
- : hand = Hand ((Ten, Diamonds), Empty)
# extract Hearts hand5;;
- : hand = Empty
```

What is the type of the function `extract`?

We define it as

```
extract : suit -> hand -> hand
```

not as

```
extract : suit * hand -> hand
```

Let's write the function `extract`

We need to recursively analyze all possible `hands`. General idea:

- If we have an `Empty` hand, there is nothing to extract and we simply return `Empty`.
- When we have `Hand(c,h)`, then we must compare whether the card `c` has the appropriate suit. If it does not, we recursively analyze the remaining hand `h`. If it does, we want to keep the card `c` and recursively analyze the remaining hand `h`.

This recipe translates directly into a recursive program in OCaml.

```
let rec extract (s:suit) (h:hand) = match h with
  | Empty -> Empty
  | Hand ((_, s') as c, h') ->
    if s = s' then Hand(c, extract s h')
     else extract s h'
```

card hand

new Hand

Note that we do not modify the input hand `h`. We are simply analyzing it and returning a new hand copying some cards from the input hand.

Find the first card in a hand of a given rank

Intuitively, we need to search through a hand and inspect the rank of each card. If we find a card whose rank matches the given one, then we return the corresponding suit. But what shall we return, if there is no such card?

Option 1: Raise an error

Option 2: Use an optional datatype!

```
type 'a option = None | Some of 'a
```

The 'a `option` type is a non-recursive type.

Let us return to writing the function `find` whose type is:

```
find: rank * hand -> suit option
```

Main idea of the function `find`

It takes a tuple consisting of a `(r:rank)` and a `(h:hand)` as input and returns an optional value of type `suit`.

- If our given hand `h` contains a card of rank `r'` and suit `s'` where `r = r'`, then we return `Some s'`
- Otherwise the result will be `None`.

We implement this function recursively by pattern matching on the hand `h`.

```
let rec find (r, h) = match h with
  | Empty                -> None
  | Hand ((r', s'), h') -> if r = r' then Some s'
                            else find (r, h')
```

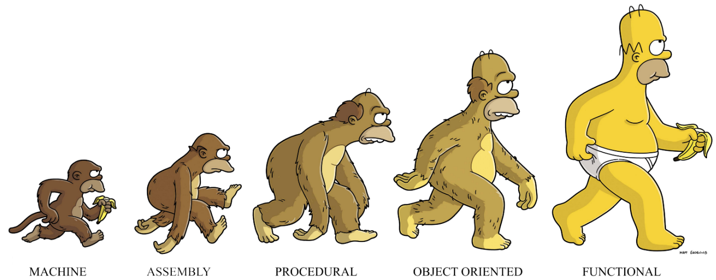
type name: hand

constructor: Hand

Take-Away

- Data types allow us to define a collection of elements; these abstractions make code more readable and easier to reason about

- Pattern matching is an elegant, powerful way to analyze data and extract information from data.



COMP 302: Programming Languages and Paradigms

Prof. B. Pientka, McGill University McGill

Week 2 (Part 2): Data Types and Pattern Matching

Part 1: Introduction to defining data types

Quick recap:

- Non-recursive data types such as

```
type suit = Clubs | Spades | Hearts | Diamonds
```

  or

```
type 'a option = None | Some of 'a
```

- Recursive data types such as

```
type hand = Empty | Hand of card * hand
```

Today, we revisit one of the most well-known data types: lists.

Part 2: Lists as Recursive Data Types

We can define polymorphic lists of type `'a list` inductively:

- The constant `[]` is of type `'a list`. It describes the empty list.
- If `h` is an element of type `'a` and `t` is of type `'a list` then `h :: t` is of type `'a list`.
- Nothing else is of type list.

or α

Note: `'a` is a *type variable*. Our inductive definition for lists is *polymorphic* (from the Greek meaning "having multiple forms").

Examples A lists of floating point numbers:

Version 1:

float float list

```
let fl0 : float list = 8.6::(5.4::[])
```

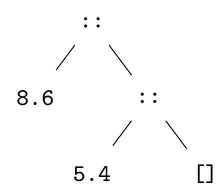
float float list

Version 2:

```
let fl10 = [8.6 ; 5.4]
```

Building lists can also be thought of as a binary tree where the leaf is the constant `[]` and the constructor `::` builds a binary tree with two children:

- the left child is an expression of type `'a`
- the right hand side is another binary tree.



The tree is a degenerate tree that will grow to the right

As lists are defined polymorphically, we can build not only lists containing floating point numbers, but also lists containing strings, lists containing integers, even lists containing other lists!

```
# let l0 = 1::2::3::[];;
val l0 : int list = [1; 2; 3]

# let l1 = "a"::"b"::"c"::"d"::[];;
val l1 : string list = ["a"; "b"; "c"; "d"]

# let l2 = "x"::"y"::"z"::[];;
val l2 : string list = ["x"; "y"; "z"]
          string list        (string list) list
# let l3 = l1 ::(l2 :: []);;
val l3 : string list list = [["a"; "b"; "c"; "d"]; ["x"; "y"; "z"]]

# let l4 = [3;4;0;7];;
val l4 : int list = [3; 4; 0; 7]
```

Appending Lists Using Pattern Matching Let's see how we write some simple recursive programs about lists using pattern matching.

Task: Implement a function `append: 'a list -> 'a list -> 'a list`.

Given a list `l1` of type `'a list` and a list `l2` of the same type `'a list`, we return list containing all the elements from `l1` followed by the elements from `l2`.

For example: `append [1; 2; 3] [4; 5]` returns `[1;2;3;4;5]`

How do we proceed?

Idea: recursively traverse the first list until it is empty and then return the second list. As we come back out of the recursion, we concatenate each element of the first list back onto the computed result. This will rebuild the full list.

```
(* append: 'a list -> 'a list -> 'a list *)
let rec append l1 l2 = match l1 with
  | []    -> l2
  | x::t -> x::append t l2
```

Note: OCaml has the append operation built-in. One can use `@` as an infix operator to append two lists simply writing `[1; 2; 3] @ [4; 5]`.

Appending Lists: The Old and Ugly Way In languages without sophisticated pattern matching, we might first define two destructors `head` and `tail` which allow us to take a list apart. We can define these destructors as functions in OCaml as follows:

```
(* tail: 'a list -> 'a list *)
let tail l = match l with
  | [] -> []
  | h::t -> t
```

```
(* head : 'a list -> 'a *)
let head (h::t) = h
```

head [] undefined

When type checking the program `head` in OCaml, you will get a warning:

```
# let head (h::t) = h;;
           ^^^^^^^^^^
Warning P: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
```

Coverage checker can give useful error messages based on patterns written.
Great for debugging and thinking about the program before running it.

Let's revisit now how to append two lists using if-expressions. In a language without pattern matching you might write:

```
let rec app (l1, l2) =
  if l1 = [] then l2
    else
      head(l1)::(app (tail(l1), l2))
```

Why is the above program ugly?

- harder to read
- harder to understand
- harder to reason about

> Pattern matching makes it easier to read and understand programs. It also makes it easier to write correct programs.

Revisiting Tail Recursion

Task: Implement the function `rev: 'a list -> 'a list`.

Given a list `l` of type `'a list`, `rev l` returns a list consisting of the elements of `l` in reverse order.

Idea: We recursively traverse the input list `l`.

- If `l` is the empty list (i.e. `[]`), then we simply return the empty list.

- If it is non-empty, pattern matching tells us that `l` stands for a list `x::xs` where `x` is the head of the list `l` and `xs` is the tail of the list `l`. We therefore recursively reverse the tail `xs` and glue `x` to the back of its result.

Note: the buil-in append operation `@` takes in two lists. Hence, we create a one element list `[x]` which only contains `x` and append `[x]` to the result of the recursive call `rev xs`.

```
let rec rev l = match l with
  | []     -> []
  | x::xs -> (rev xs) @ [x]
```

append ⇒ quadratic run time
non linear!

precedence order

Why is the above program unsatisfying?

Tail-recursive version of reverse

Idea: Write a helper function `rev_tr:'a list -> 'a list -> 'a list` which takes as input a list `l` and an accummulator `acc` as an additional argument.

- In the base case, we simply return the accummulator `acc`.
- In the step case, we build the accummulator by simply pushing the head `x` of the input `l` onto the accummulator `acc`.

We call the helper function `rev_tr` by initializing the accumulator with the empty list.

```
let rev' l =
  let rec rev_tr l acc = match l with
    | []   -> acc
    | h::t -> rev_tr t (h::acc)
  in
    rev_tr l []
```

This program will now run in linear time and is tail-recursive.

Take Away

We introduced basic concepts that allow you already many recursive programs directly. The key ideas we introduced are:

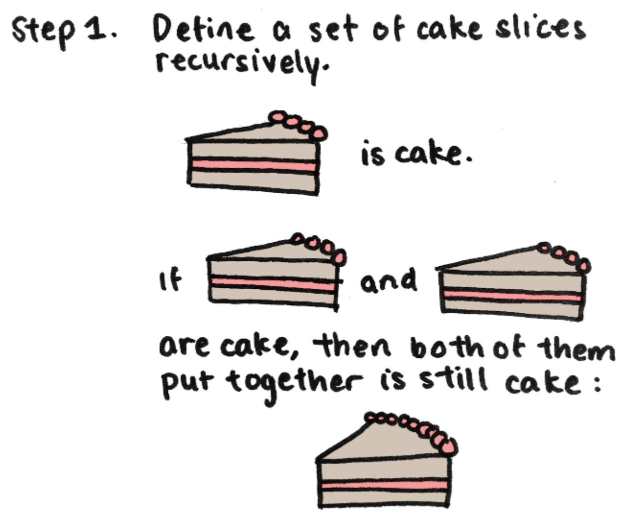
- Lists are inductively defined
- We can translate the inductive definition into a recursive data-type
- Defining recursive functions using pattern matching
  It's elegant, code is easier to read, and more likely to be correct.

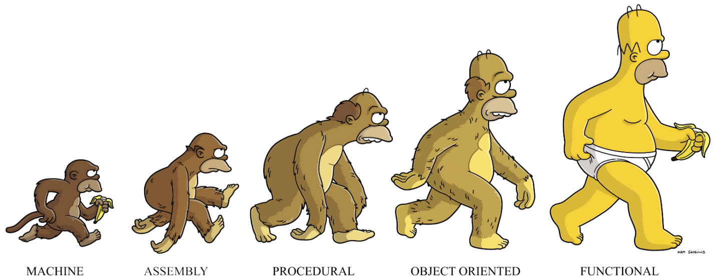
Ultimately the best way to learn a programming language is to use it!

Food for Thought: Get me some cake!

Not quite ...

Task: Give an OCaml data type definition for cake!





COMP 302: Programming Languages and Paradigms
Prof. B. Pientka, McGill University

McGill

Week 2 (Part 3): Data Types and Pattern Matching

Recursive Data Types: Binary Trees

As a last example, we consider here the representation of binary trees as recursive data type. This is one of the most commonly used data-structures.

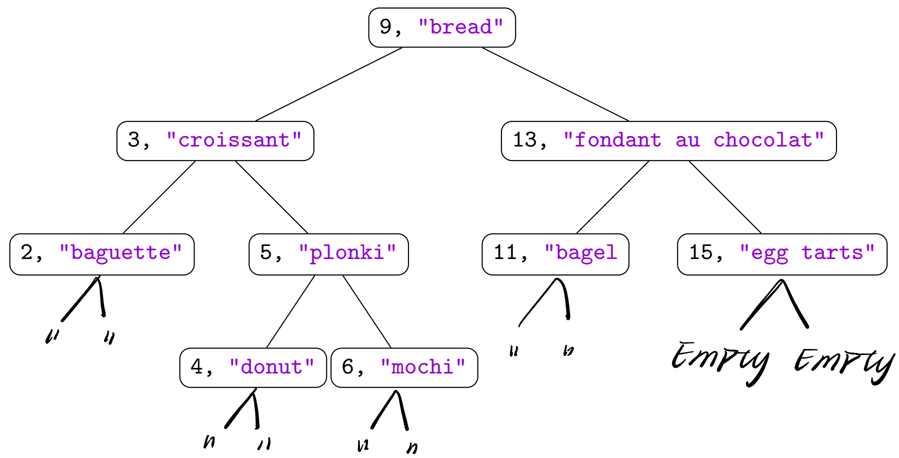
We can define a binary tree of type `'a tree` inductively as follows:

- The empty binary tree `Empty` is a binary tree of type `'a tree`.
- If `l` and `r` are binary trees and `v` is a value of type `'a` then `Node (v, l, r)` is a binary tree of type `'a tree`.
- Nothing else is a binary tree.

Translating the inductive definition into using the following recursive data type definition:

```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

Example




```
let t1 =
Node((9, "bread"),
   Node((3, "croissant"),
     Node((2, "baguette"), Empty, Empty),
     Node((5, "plonki"), Node((4,"donut"),Empty, Empty),
                         Node((6,"mochi"), Empty, Empty))),
   Node((13,  "fondant au chocolat"),
     Node((11, "bagel"),Empty, Empty),
     Node((15, "egg tarts"), Empty, Empty)));;
```

Our example of a binary tree here is in fact a *binary search tree* where the keys in all children in the left side of the tree are smaller than the key in the parent node and keys of all children on the right side of the tree are larger than the parent node.

Note that our data type definition does not require us to store always data together with their key but is more general.

How to insert an element into a binary search tree

The type of the function `insert` that we want to write can be described as:

```
insert:  'a * 'b -> ('a * 'b) tree -> ('a * 'b) tree
```

This is our goal.

How do we proceed?

Intuitively, we proceed as follows.

Given a data entry `e` of type `'a * 'b` and a binary tree `t` of type `('a * 'b) tree` we consider two cases:

1. If `t` is the empty tree `Empty`, then we return a new tree with `e` as the parent node.
2. If `t` is a tree of the form `Node ((y,d'), l, r)` then we want to insert our data entry `e` into the left side `l` or right side `r` resp. depending on whether the key of `e` is smaller or larger than `y`. Note that we need to re-build the tree when we have inserted the entry in the corresponding sub-tree. Last, we need to decide what should happen if the key of `e` is equal to the key `y`. Intuitively, we want that when we insert an entry `(x,d) as e`, and subsequently look up an entry with key `x` that we obtain the data `d` (not `d'`!). Hence, in this case, we return a tree where we use as the new entry `e`.

These considerations lead to the following program.

```
let rec insert ((x,d) as e) t = match t with
  | Empty              -> Node(e, Empty, Empty)
  | Node ((y,d'), l, r) ->
      if x = y then Node(e, l, r)
      else
  (if x < y then Node((y,d'), insert e l, r)
   else
     Node((y,d'), l, insert e r))
```

How to implement a lookup function?

Given a binary search tree `('a * 'b) tree` and a key `'a`, we want to return the data associated with the key.

What should we return, if there is no entry for a given key?

We use here an opional value and return a result of type `'b option`.

```
(* lookup: 'a -> ('a * 'b) tree -> 'b option *)

let rec lookup x t = match t with
  | Empty              -> None
  | Node ((y,d), l, r) ->
  if x = y then Some(d)
    else
      (if x < y then lookup x l
   else lookup x r)
```

How do `insert` and `lookup` fit together?

Invariant: When we insert an entry `(x,d)` into a tree and we subsequently want to retrieve the data associated with `x` we indeed obtain `d`.

This can be made more precise by the following statement:

```
lookup x (insert (x,d) t) = Some d
```

evaluate to a value.

Check out the chapter and lecture on how to reason inductively about recursively defined data and programs!

Take-Away

We introduced the example of a binary tree and how to define it using recursive data types.

- We introduced the example of a binary tree and how to define it using recursive data types.
- We showed how to write programs using pattern matching on binary trees.

Ultimately the best way to learn a programming language is to use it!