Discussion 1

Friday, January 7, 2022 12:00 PM

Functional programming & OCaml

- Functional programming is a programming paradigm where programs are constructed by applying and composing functions
  - No side effect, the value of "variable" never changes
    - Calling a function twice with the same arguments will have the same results
    - "pure" function, behaving similar to mathematical functions
  - First-class, higher order functions
    - Functions can take other functions as arguments or return them as result
  - Recursion (NO for/while loops), no side effects
- Why learning functional programming
  - Makes compiling and testing easier
    - Function without side effects makes reasoning on the code easier
  - Easy to build scalable systems
    - Computing can be distributed on multiple machines more easily when there is no shared states or side effects

OCaml Programming Language

- Functional programming language
- Statically typed
  - Every value (including function) has a type
  - Type checking at compile time
- Strong Type inference
  - In many cases, you do not need to specify types
- Garbage Collection
- Compiled language

Running OCaml

- CTRL-D quits
- Example simple code:
  - print_string "Hello World!\n" ;;
  - let my_value = 5 ;;
- Types: int, float, bool, char, string, char, string, unit (empty, or null type, written as ())
- Functions:
  - Created with "let" keyword
    - let <name> <parameter(s)> = <expressions>;;
  - Do not need to add parentheses or commas for arguments
  - Recursive functions are specified with "let rec"
    - Let rec factorial a = if a = 1 then 1 else a * factorial (a -1);;
- Define local bindings
  - Add keyword "in" after "let" to make the value available in the following expression
    - let average a b = let sum = a +. B in sum /. 2.0;;
    - Can also create helper functions with it
    - Use "and" for multiple key bindings or chain multiple "let … in" together
- Lambda functions: anonymous functions that don't have names
  - Useful when using a function as a function parameter
    - (fun x-> x * x) 5;;
- Lists

- let numbers = [1; 2];;
- All elements have the same type
- Non-empty list consist of a head and a tail
  - List.hd <list name> #first element
  - List.tl <list name> #the rest of the list
- Last tail of a list is an empty list[]
- You can iterate the list by running the head of the tail of the list
- :: to add something to the list beginning
- List modules:
  - Map: transforms each element with a function
    -
```
# List.map (fun x -> x * x) [1;2;3;4;5];;
- : int list = [1; 4; 9; 16; 25]
```
  - Filter: returns a list that contains elements that matches the function
    -
```
# List.filter (fun x -> x mod 2 = 0) [1;2;3;4;5];;
- : int list = [2; 4]
```
  - Rev: reverse the list
    -
```
# List.rev [1;2;3;4;5];;
- : int list = [5; 4; 3; 2; 1]
```
  - For_all: checks if all elements satisfy the specified function
    -
```
# List.for_all (fun x -> x < 3) [1;2;3;4;5];;
- : bool = false
```
  - Exists: checks if there exists an element that satisfied the specified function
    -
```
# List.exists (fun x -> x < 3) [1;2;3;4;5];;
- : bool = true
```
- Pattern matching
  - Powerful version of control statements
  -
```
# let is_zero x = match x with
    0 -> true
  | _ -> false;;
```
  - Allows to list all different cases in a clean way
    - Sequential check from the beginning and find the first rule that matches
    - Use “|” before sequential checks
    - Underscore (_) is used as a wildcard that matches any value
    - Cleaner than conditionals when there is a large number of possible values
  - Compiler lets you know which cases do not match to any of the rules
- Pattern match lists
  -
```
# let first l = match l with
    head :: tail -> head
  | _ -> 0;;
val first : int list -> int = <fun>
# first [1;2;3];;
- : int = 1
```

```
# let listHasOneElement l = match l with
    [] -> false
  | hd::tl -> if tl = []
              then true
            else false;;
val listHasOneElement : 'a list -> bool = <fun>
# listHasOneElement [];;
- : bool = false
# listHasOneElement [1];;
- : bool = true
# listHasOneElement [1;2];;
- : bool = false
```

  - Can be used to enumerate the list by using head::tail
- Patterns can also include conditions using “when” statement

- Tuples: a collection of values can be different types
  - Values separated by commas
  - Often surrounded by parentheses
  - ```
    # let my_tuple = "foo", 3.14, 'c', false;;
    val my_tuple : string * float * char * bool = ("foo", 3.14, 'c', false)
    ```
  - Type representation: individual element types, separated with (*)
  - Accessing tuple elements:
    - Tuples with two elements: fst my_tuple; snd my_tuple
    - Pattern matching (even without “match”, which can get quite versatile):
- Variants
  - One of the user-defined types in OCaml, "type" keyword used to define
  - ```
    # type color = Red | Green | Blue;;
    type color = Red | Green | Blue
    # let my_blue = Blue;;
    val my_blue : color = Blue
    ```
  - ```
    # type my_type =
        | A of int
        | B of string;;
    type my_type = A of int | B of string
    # let my_a = A 15;;
    val my_a : my_type = A 15
    # let my_b = B "hello";;
    val my_b : my_type = B "hello"
    ```
  - Using variants with pattern matching
  - ```
    # type my_type =
        | A of int
        | B of string;;
    type my_type = A of int | B of string
    # let my_a = A 15;;
    val my_a : my_type = A 15
    # let my_b = B "hello";;
    val my_b : my_type = B "hello"
    ```
  - ```
    # let my_print x =
        match x with
          A a -> print_int a
        | B b -> print_string b;;
    val my_print : my_type -> unit = <fun>
    # my_print (A 8);;
    8- : unit = ()
    # my_print (B "foobar");;
    foobar- : unit = ()
    ```

- Polymorphic types
  - Sometimes there isn't enough information to infer a concrete type
    ```

- In this case 'a and 'b tell us they are polymorphic
  - All instances of 'a, 'b should be of same type
- When there's enough information, the type will be inferred
  - If you call f 2 3.0, ''a, 'b will be int and float
  -
```
# let f x y = x ;;
val f : 'a -> 'b -> 'a = <fun>
# let g x y = y /. 2.0;;
val g : 'a -> float -> float = <fun>
```

Homework 1:

- Context-free grammars
  - Grammar in programming: what strings are valid for a language, defines allowed syntax
- Basic elements:
  - Symbol
    - Terminal: symbol that cannot be replaced by other symbols (e.g. +)
    - Nonterminal: symbol that can be replaced by other symbols (e.g. BINOP)
  - Rule
    - Defines how a nonterminal symbols can be replace by other symbols
  - Grammar contains a starting symbol (e.g., EXPR) and a set of rules

- Homework 1: unreachable rules
- A slightly modified grammar
-
```
Example Grammar:

EXPR -> NUM
EXPR -> NUM BINOP NUM
BINOP -> +
BINOP -> -
NUM -> 0
NUM -> 1
INCROP -> ++
```
- The nonterminal symbol INCROP can never be reached from EXPR (starting symbol)
  - Thus, rules with INCROP are unreachable rules in this grammar
  - In the homework, you need to remove all the unreachable rules according to grammar and starting point
-