Principles of Programming Languages

CS496

Basic Types

- Next we will begin experimenting with
  - basic types and
  - expressions of basic types
- These types include
  - `int`
  - `bool`
  - `float`
  - `string`
  - `char`
  - `unit`
- There are other types, we'll see them later

`int` – integers

```
# 1;;
- : int = 1
# 12345 + 1;;
- : int = 12346
# 12345 - 1;;
- : int = 12344
# 3+4;;
- : int = 7
# 8/3;;
- : int = 2
# 30_000_000 / 300_000;;
- : int = 100
```

`float` – floating point numbers

```
# 3.5 +. 6.;; (* notice the dot after the + *)
- : float = 9.5
# sqrt 9.;;
- : float = 3.
# 1 + 2.0;;
      ^^^
  This expression has type float but is here used with type
```

Note the last expression:

- The `+` function operates on integers, but `2.0` is not an integer

float – floating point numbers

- It is possible to convert from integers to floats and back
- float_of_int is called a function; we will study functions later

```
# float_of_int 1;;
- : float = 1.
# int_of_float 1.2;;
- : int = 1
# 1 + int_of_float 2.0;;
- : int = 3
```

`char` – characters

```
# 'a';;
- : char = 'a'
# 'x';;
- : char = 'x'
# "hello".[1];;
- : char = 'e'
```

char – characters

- OCaml provides a set of built-in modules
- Modules define useful operations on numerous types
- An example is the Char module which provides useful operations on chars

```
# Char.uppercase 'z';;
- : char = 'Z'
# Char.uppercase '[';;
- : char = '['
# Char.chr 97;;
- : char = 'a'
# Char.code 'a';;
- : int = 97
```

string – strings

```
# "Hello";;
- : string = "Hello\n"
# "Hello " ^ " world\n";;
- : string = "Hello world\n"
# "The character     \000     is not a terminator";;
- : string = "The character     \000     is not a terminator"
# "\072\105";;
- : string = "Hi"
# "Hello".[1];;
- : char =     e
```

`string` – strings

The `String` module provides many useful functions on strings

```
# String.length "Ab\000cd";;
- : int = 5
# String.sub "Abcd" 1 2;;
- : string = "bc"
```

`bool` - booleans

```
# 2 < 4;;
- : bool = true
# "A good job" > "All the tea in China";;
- : bool = false
# 2 + 6 = 8;;
- : bool = true
# 1.0 = 1.0;;
- : bool = true
# 2!=4;;
- : bool = true
 # true && false;;
- : bool = false
# true || false;;
- : bool = true
```

use = for equality checking

bool - booleans

```
# if 1 < 2
  then 3+7
  else 4;;
- : int = 10
# if 3!=4 then 1 else 2;;
- : int = 1
# if 2 then 3 else 4;;

Error: This expression has type int but an expression was e
type bool
```

`unit` - unit type

- Special type typically assigned to expressions that cause effects
- Example: `print_string` for printing a string

```
# ();;
- : unit = ()
# print_string "hello";;
hello- : unit = ()
# print_char 'a';;
a- : unit = ()
# print_int 3;;
3- : unit = ()
```

`unit` - unit type

- Expressions of unit type can be composed using “;”

```
# print_string "hello"; print_string "bye";;
hellobye- : unit = ()
```

Variables

- Variables are names given to values
  - These names always start with lowercase
- Variables allow these values to be reused
- Variables are declared using: `let` identifier = expression

```
# let x = 1;;
val x : int = 1
# let y = 2;;
val y : int = 2
# let z = x + y;;
val z : int = 3
```

Nesting Declarations

Declarations can be nested using the form

```
let variable=expression in expression
```

```
# let x = 1
  in let y = 2
  in x + y;;
- : int = 3
# let z =
   let x = 1
   in let y = 2
   in x + y;;
val z : int = 3
# let x =
    let y = 2 in y
  in x+1;;
- : int = 3
```

Basic Types vs Agregate Types

- Basic types seen so far
  - `int`
  - `bool`
  - `float`
  - `string`
  - `char`
  - `unit`
- Agregate types we shall see now
  - They are built out of simple types by composing them
  - We will see two composite type constructors (more, eg. lists, later)
    - Functions
    - Tuples

Functions

```
#  let succ i = i + 1;;
val succ : int -> int = <fun>
# succ 1;;
- : int = 2
# succ (succ 1);;
- : int = 3
# succ;;
- : int -> int = <fun>
```

Functions

An alternative definition using anonymous functions

```
#  let succ i = i + 1;;
val succ : int -> int = <fun>
# let succ2 = fun i -> i + 1;;
val succ2 : int -> int = <fun>
# succ2 1;;
- : int = 2
```

`fun`, used above, allows anonymous functions to be defined

```
# fun x -> x+1;;
- : int -> int = <fun>
```

Function Types

Lets take a closer look at the type of `succ`

```
# let succ i = i + 1;;
val succ : int -> int = <fun>
```

The type of `succ` is `int -> int`

What does this function do and what is its type?

```
# let f i = i>0;;
```

What happens if you evaluate `f 3.5`?

Exercise

- Define the function `sign` which given an integer returns 1 if it is positive, -1 if it is negative and 0 if it is zero.
- What is the type of `sign`?

Exercise

- Suppose we use a function of type `int -> bool` to denote a subset of integers, namely those for which the function returns true
- Such functions are called characteristic function of a set
- For example, `let f x = x mod 2` denotes the set of even numbers
- Define union and intersection of sets represented through their characteristic functions
- `union` and `intersection` should have type
  `(int->bool) -> (int->bool)-> int -> bool`

Functions with Multiple Arguments

```
# let add i j = i + j;;
val add : int -> int -> int = <fun>
# add 2 3;;
- : int = 5
# add 2 3 4;;
Error: This function has type int -> int -> int
       It is applied to too many arguments; maybe you forgo
```

Functions with Multiple Arguments

An alternative definition using anonymous functions

```
# let add2 = fun i j -> i + j;;
val add2 : int -> int -> int = <fun>
# add2 2 3;;
- : int = 5
```

Function Types

- Lets take a closer look at the type of `add`

```
# let add i j = i + j;;
val add : int -> int -> int = <fun>
```

- The type of `succ` is `int -> int -> int`
- How do we read this type?
  - `succ` is a function that

    given an integer `i`, returns a function that
    given an integer `j`, returns `i+j`

Partial Application

- `succ` is a function that

  given an integer `i`, returns a function that
  given an integer `j`, returns `i+j`

This means we can apply `add` to just ONE argument and get back a function

```
# add 1;;
- : int -> int = <fun>
```

A new way to define succesor!

```
# let succ3 = add 1;;
val succ3 : int -> int = <fun>
# succ3 4;;
- : int = 5
```

Exercise

- Define the function `min3` that given three integers returns the smallest one.
  - Use if-then-else and conjunction
- What is the type of `min3`?

Exercise

- Define the functions `and'`, `or'`, `not'` and `xor'` which implement the standard boolean operations.
- What is the type of each of these functions?

Tuples

Just like ordered tuples in math

```
# (2,3);;
- : int * int = (2, 3)
# (true,3);;
- : bool * int = (true, 3)
# (true,2,4);;
- : bool * int * int = (true, 2, 4)
#
# (2,(true,23));;
- : int * (bool * int) = (2, (true, 23))
```

- Tuples that have just two components are called pairs
- The type of a tuple is `t1 * t2 * ... * tn` where each `ti` is the type of the respective component

Tuples

How do we access the components of a tuple?

```
# let fst (x,y) = x;;
val fst : 'a * 'b -> 'a = <fun>
# fst (2,3);;
- : int = 2
# fst (2,3,4);;
Error: This expression has type 'a * 'b * 'c
  but an expression was expected of type 'd * 'e
```

Note that `fst` uses pattern matching:

- `(x,y)` is a pattern that can only match a pair
- binds variables `x` and `y` to the first and second component of the pair, resp.

Tuples

```
# let f2 (x,_)=x;;
val f2 : 'a * 'b -> 'a = <fun>
# f2 (2,3);;
- : int = 2
```

Exercise on Types

Provide expressions of the following types:

1. `bool`
2. `int * int`
3. `bool -> int`
4. `(int * int) -> bool`
5. `int -> (int -> int)`
6. `(bool -> bool) * int`

Polymorphism

- What is the type of this function?

```
let id x = x
```

- Here are examples of its use:

```
# id 2;;
- : int = 2
# id true;;
- : bool = true
# id "hello";;
- : string = "hello"
```

- Its type should be `t -> t`, for any type `t`
- How do we express such a type? We use type variables

```
'a -> 'a
```

Polymorphism

```
let id x = x;;
val id : 'a -> 'a = <fun>
```

- This type is read as follows:

  > *`id` is a function that given a value of type `'a`, returns another value of the same type `'a`*

- We say `id` is polymorphic
- Notice that
  - `id 2` has type `int` and
  - `id true` has type `bool`

Polymorphism

- It is a feature of type systems
- It allows an expression to have infinite types
- The type system then adjusts these types to more concrete ones depending on the use of these expressions
  - `id: 'a -> 'a (*general type *)`
  - `id: int -> int (*more concrete type *)`
- This style of polymorphism is called parametric polymorphism (the parameter is the type variable)

More Examples

```
# let f x = 7;;
```

- What does `f` do and what is its type?

```
# let f x = 7;;
val fst : ’a -> ’int = <fun>
```

More Examples

```
# let fst (x,y) = x;;
val fst : 'a * 'b -> 'a = <fun>
# fst (2,3);;
- : int = 2
```

- `fst` takes a pair of type `'a * 'b` and returns a result of type `'a`

More Examples

```
# let f x y = x;;
```

- What does this function do?
- What is its type?

Exercise

- Write a function `swap` that takes in a pair and returns the same pair but where the components have been swapped
- For example, `swap (2,true)` should return `(true,2)`
- What is the type of this function?

Motivating Example

What is the type of the following function?

```
# let twice f x = f (f x);;
```

Consider the following example

```
# let twice f x  =  f (f x);;
val t : ('a -> 'a) -> 'a -> 'a = <fun>
# let sqr x =  x*x;;
val sqr: int -> int = <fun>
# twice sqr 2
- : int = 16
```

Higher-Order Functions

A function that takes another function as argument or that returns a function as result[1]

[1] The precise notion is more technical; this suffices for us.

Higher-Order Functions

Are these functions higher-order?

```
# let add x y = x + y;;
val add : int -> int -> int = <fun>
# let myAnd x y = x && y;;
val myAnd : bool -> bool -> bool = <fun>
```

- According to our definition, they are
  - `add`, given an integer, returns a function
- Note: `int -> int -> int` is the same as writing `int -> (int -> int)`
  - `->` associates to the right

Higher-Order Functions

A function that takes another function as argument or that returns a function as result[2]

Some more examples

```
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
# apply (fun x -> (x,x)) 3;;
- : int * int = (3, 3)
# let apply' =  fun f -> (fun x -> f x);;
val apply' : ('a -> 'b) -> 'a -> 'b = <fun>
# apply' (fun x -> (x,x)) 3;;
- : int * int = (3, 3)
```

[2] The precise notion is more technical; this suffices for us.

More Examples

```
# let compose f g x = f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let sqr x = x*x;;
val sqr : int -> int = <fun>
# compose sqr sqr 2;;
- : int = 16
```

Lists

A list is an ordered sequence of values of the same type

```
# [1;2;3];;
- : int list = [1; 2; 3]
# [1;1;3];;
- : int list = [1; 1; 3]
# ["hello"; "bye"];;
- : string list = ["hello"; "bye"]
# [1;"hello"];;
Error: This expression has type string but an expression wa
of type int
# 1::[2;3];;
- : int list = [1; 2; 3]
# [];;
- : 'a list = []
# 1::(2::(3::[]));;
- : int list = [1; 2; 3]
```

`::` is called cons, it adds an element to the beginning of a list

Lists – cons Operator

`::` is called cons

- it adds an element to the beginning of a list
- its type is `'a -> 'a list -> 'a list`

```
# 1::[2;3];;
- : int list = [1; 2; 3]
# [];;
- : 'a list = []
# 1::(2::(3::[]));;
- : int list = [1; 2; 3]
```

Lists – Append

```
# [1;2;3] @ [4;5];;
- : int list = [1; 2; 3; 4; 5]
# [1;2;3] @ [];;
- : int list = [1; 2; 3]
# [1;2] @ ["hello";"bye"];;
Error: This expression has type string but an expression wa
of type int
```

Recall: use = for equality checking

Concatenating Lists

- Which of these are true and which are false?
- Under what assumptions?

```
[[]] @ xs   = xs
[[]] @ [xs] = [[],xs]
[[]] @ xs   = [xs]
[]::xs      = xs
[[]] @ [xs] = [xs]
[[]] @ xs   = []::xs
[xs] @ [xs] = [xs,xs]
[]   @ xs   = []::xs
[[]] @ xs   = [[],xs]
[xs] @ []   = [xs]
```

`List` Module

- Contains many useful operations on lists
- One example is `length`

```
# List.length [1;1;2];;
- : int = 3
# length [1;2;3];;
Error: Unbound value length
# open List;;
# length [1;2;3];;
- : int = 3
```

Note: Browsable sources of OCaml libraries

`http://caml.inria.fr/cgi-bin/viewvc.cgi/ocaml/trunk/stdlib/`

Recursion

- Problem: Write a program that, given an integer n, adds the first n integers
- Example: if $n = 10$ then we want to add

$$0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10$$

```
# let rec sum n =
  match n with
    0 -> 0
  | n -> n + sum (n-1);;
val sum : int -> int = <fun>
# sum 0;;
- : int = 0
# sum 10;;
- : int = 55
```

Recursion

```
# let rec sum n =
  match n with
    0 -> 0
  | n -> n + sum (n-1);;
```

- `rec` says that we are defining a recursive function
  - A recursive function is a function that can call itself
- `match` is used for pattern matching on `n`
  - It is typically used in combination with `rec` but doesn't have to
- Lets follow the execution of couple of uses of `sum`

Recursion

On the board:

- `sum 0`
- `sum 1`
- `sum 2`
- `sum 3`

Recursion

```
# let rec sum n =
  match n with
    0 -> 0
  | n -> n + sum (n-1);;
# sum (-3);;
Stack overflow during evaluation (looping recursion?).
# let rec sum n =
  match n with
    0 -> 0
  | n when n>0 -> n + sum (n-1)
  | _ -> failwith "sum:: argument must be non-negative";;
val sum : int -> int = <fun>
# sum (-3);;
Exception: Failure "sum:: argument must be non-negative".
# sum 10;;
- : int = 55
```

Another Example of Recursion

- Problem: Write a program that, given an integer n, multiplies the first n integers
- Note: if $n = 0$ it should return 1
- Example: if $n = 10$ then we want to return

$$1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10$$

```
# let rec fact n =
  match n with
    0 -> 1
  | n -> n * fact (n-1);;
val fact : int -> int = <fun>
# fact 0;;
- : int = 0
# fact 10;;
- : int = 3628800
```

Exercise

- Write a function `list_enum` that given a positive number `n` returns the list `[n;n-1;...;1;0]`
- For example, `list_enum 5` should return `[5;4;3;2;1;0]`
- What is the type of `list_enum`?

Exercise

- Write a function `repeat` that given an argument `x` and a positive number `n` returns the list

  `[x;x;...;x]`

  where `x` is repeated `n` times
- For example, `repeat "hello" 4` should return `["hello"; "hello"; "hello"; "hello"]`
- What is the type of `repeat`?

Exercise

- Write a function `stutter` that given two positive numbers `n` and `m` returns a new list of the form

  `[[n;n;...;n];[n-1;n-1;...;n-1];...;[0;0;...;0]]` where each nested list has `m` items
- For example, `stutter 3 2` should return

  `[[3;3];[2;2];[1;1];[0;0]]`
- What is the type of `stutter`?

The Length of a List

```
let rec length l =
  match l with
    [] -> 0
  | x::xs -> 1 + length xs
```

- Note the two cases in the definition:
  - the empty list `[]` – called the base case
  - the non-empty list `x::xs` – called the inductive case
- Run this function on a sample list
- What is the type of `length`?

Sum of a List of Numbers

```
let rec sum l =
  match l with
    [] -> 0
  | x::xs -> x + sum xs
```

Alternatively,

```
let rec sum = function
    [] -> 0
  | x::xs -> x + sum xs
```

Exercise

- Write a function that multiplies all the numbers in a list

Functions that Construct Lists

```
let rec incr l =
  match l with
    [] -> []
  | x::xs -> (x+1)::incr xs
```

Note: Function application has precedence over `::`

Stutter

What does this function do?

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

Exercise

- Define a function `is_zero_list` that given a list of numbers returns a list of booleans indicating whether each number is 0 or not.
- For example,

```
> is_zero_list [3;0;7;0;0];;
- : bool list = [false; true; false; true; true]
```

- What is the type of this function?

Functions that Filter Elements from a List

What does this function do?

```
let rec even l =
  match l with
    [] -> []
  | (x::xs) ->
       if (x mod 2=0)
       then x :: (g xs)
       else even xs
```

- Try it out on an example

Functions that Filter Elements from a List

What does this function do?

```
let rec even l =
  match l with
    [] -> []
  | (x::xs) ->
        if (x!=[])
        then x :: (g xs)
        else even xs
```

- Try it out on an example

Functions that Filter Elements from a List

- Define a function that given a list of strings and a number `n`, filters (i.e. keeps) those strings whose length is smaller or equal to `n`
- What is the type of this function?

Functions on Lists that Deviate from Standard Patterns

- The standard patterns when defining a recursive function `f` over lists are:
  - Define `f` over the empty list `[]` (called base case)
  - Define `f` over the non-empty list `x::xs` (called inductive case)
- Some functions however don't fall in that scheme
- Here are some examples that we will develop on the board:
  - `head`
  - `tail`
  - `maximum`
  - `last`
  - `remove_adjacents`

Motivating Examples

- Let us implement the following functions
  - `succl : int list -> int list`
  - `to_upperl : char list -> char list`
  - `all_zero : int list -> bool list`
- What do you notice in common among all these implementations?

Map

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

- What does map do?
- What is its type?
- How can we use it to define succl, to_upperl and all_zero?

```
let succl' = map (fun x -> x+1)
let to_upperl' = map Char.uppercase_ascii
let all_zero = map (fun x -> x=0)
```

Filter

- Lets implement the following functions:
  - `greater_than_zero : int list -> int list`
  - `uppercase : char list -> char list`
  - `non_empty : 'a list list -> 'a list list`
- What do you notice that they have in common?

Filter

```
let rec filter p l =
 match l with
 | [] -> []
 | (x::xs) -> if (p x)
                then x::(filter p xs)
                else filter p xs
```

- What does `filter` do and what is its type?
- How can we use filter to implement `greater_than_zero`, `uppercase` and `non_empty`?

```
let greater_than_zero = filter (fun x -> x>0)
let uppercase = filter (fun x -> x=Char.uppercase_ascii x)
let non_empty = filter (fun x -> x!=[])
```

Iterate

- Suppose we want to print out all the strings in a list of strings
- Here is one possible implementation of `print_list_of_strings`

```
let rec print_list_of_strings l =
match l with
| [] -> ()
| (x::xs) -> print_string x;
             print_list_of_strings xs
```

Iterate

- OCaml provides `List.Iter`

```
List.iter print_string
```

Fold

Consider the implementation of the following functions

- `sum_list : int list -> int`, that adds all the elements in a list of integers
- `and_list : bool list -> bool`, that indicates whether all the booleans in the list are true
- `concat : 'a list list -> 'a list`, that concatenates all the lists in a list

What do you notice in common among their implementations?

Fold

```
let rec fold_right f l a =
match l with
| [] -> a
| (x::xs) -> f x (fold_right f xs a)
```

- Here is a description of the result of `fold_right f [x1; ...; xn] a`:

  `f x1 (f x2 (... (f xn a) ...))`

- What is its type?
- How can we define `all_fives`, `all` and `concat` in terms of `fold_right`?

Function Schemes

- map, filter, iter and fold are known as function schemes
- They abstract common patterns of behaviour
- Also, they allow for code reuse
- Finally, they help better understand the problem

Higher-Order Function Schemes

```
take

append
```

- Function schemes over function types

Summary

- OCaml expressions
  - Every expression has a unique type
- Types:
  - Basic types such as `int` and `bool`
  - Agregate types such as `int -> int` and `int * int`
- We've learned to evaluate programs in `utop`
- A word on style: `https://www.seas.upenn.edu/~cis341/current/programming_style.shtml`
- Polymorphism
- Higher-order functions
- Recursion on numbers and lists
- Functions on lists
- Function schemes (map, filter, fold)