

# O'Caml

## Interpreter

- Expressions syntax  
expression;;

## Variables

- Syntax  
let name = expression;;
- Must start with a lower case letter or an underscore
- May only contain letters, numbers, underscores and single quotes (')

## Standard Types

- Integers ([ $-2^{31}$ ,  $2^{31}-1$ ] in a 32 bit environment)  
int
- Floating point values (uses double precision)  
float
- Boolean values (false, true)  
bool
- Characters  
char
- Strings  
string

## Operators

### Integers:

- Arithmetic  
+, -, \*, /
- Comparison  
<, >, <=, >=, =, <>
- Other  
mod

### Floats:

- Arithmetic  
+., -., \*., /., \*\*
- Comparison  
<, >, <=, >=, =, <>

## Type conversion

- It is not allowed to mix types in an expression
- It is possible to convert between types explicitly
- In general, type conversion uses the syntax  
type<sub>1</sub>\_of\_type<sub>2</sub> expression  
Converts from values of type<sub>1</sub> to values of type<sub>2</sub>

## Characters

- Uses the ISO 8859-1 standard
- Syntax  
'a' 'A' '7' '%' '\n' 'è'
- Basic operators on characters:  
<, >, <=, >=, =, <>

- Type conversion  
`int_of_char, char_of_int`

## Strings

- A sequence of characters enclosed in double quotes
- Syntax  
`"This is a string"`  
`"This string has a newline\n"`
- Basic operators  
`^, <, >, <=, >=, =, <>`

## Booleans

- The truth values `false` and `true`
- Basic operators  
`&&, ||, not`
- Comparison  
`<, >, <=, >=, =, <>`

## Functions

- Syntax  
`let name par1 par2 ... parn = expression;;`
- Functions are values too, they are first class citizens
  - Anywhere you can place an ordinary expression, you can have a function
- We can use functions as arguments to functions
- If we evaluate a function (supplying no arguments) the value is a function, these functions are called higher order functions

## Function type

- A function type determines the type of the arguments and of the result
- The type is automatically inferred by the O'CamL interpreter  
`type1 → type2 → ... → typen → typeresult`

## Printing

- Printing on standard output is easiest done with  
`print_string`  
`print_char`  
`print_int`  
`print_float`
- The result type of these functions is the special type `unit`.  
`unit` is similar to the `void` type in C++  
there is only a single value in `unit`: `()`

## Sequences

- Syntax  
`expression1; expression2; ...; expressionn;;`
- Each expression is evaluated in turn
- The value of the sequence is the value of the last expression

## If-then-else

- Syntax
 

```
if expr1 then expr2 else expr3;;
```
- expr<sub>1</sub> must be of type bool
- expr<sub>2</sub> and expr<sub>3</sub> must be of the same type
- The type of the if-then-else is the same as expr<sub>2</sub> and expr<sub>3</sub>

## Arithmetic

- For floating point:
 

```
ceil, floor, sqrt, exp, log, log10, cos, sin, ...
```
- For integers:
 

```
succ, pred, abs, ...
```

## Finding out the type of a function

- The interpreter can tell us the type of any defined function
- Syntax
 

```
function_name;;
```

## Operators

- Operators are infix functions
- An infix function takes two arguments
- Syntax
 

```
expression1 op expression2
```

 or
 

```
(op) expression1 expression2
```

## Tuples

- Several values can be combined to form a single value called a tuple
 

```
(val1, val2, ..., valn)
```
- Useful for instance to compute several values simultaneously

## Structural and Physical equality

- Structural equality
  - syntax:
 

```
exp1 = exp2
```
  - compares two values for equality
  - usually the desired semantics (but can be slower than testing physical equality)
- Physical equality
  - syntax
 

```
exp1 == exp2
```
  - compares two values for identity
  - comparison on pointer values for dynamically allocated objects in C++
  - is always fast (constant time) but needs more care to be used correctly
- For some types physical and structural equality is the same (int, bool, char)
- Recommendation: When in doubt, use structural equality

## The let construct

- The let construct is a definition of a variable or function

- Creates a binding from a name to a value/function
- Definition does not have a value, but the interpreter reports the value of whatever is defined
  - For variable definitions, the value of the variable
  - For function definitions, the value of the function
  - Definitions can not be used in expressions

## Type inference

- Figuring out the type of an expression or definition is called type inference
- Given an OCaml snippet, the interpreter will
  - Infer the type (and check that it is consistent) using an abstract syntax trees (a representation of an expression or definition as a tree)
  - Evaluate the expression/Add the definition

## Recursion

- Syntax
 

```
let rec name par1 par2 ... parn = expression;;
```

## Local definitions

- Syntax
 

```
let id id* = expr1 in expr2;;
```

## Pattern matching

- Pattern matching is a construct similar to the switch statement of C++
- Syntax
 

```
match expr with
  pattern1 -> expr1
| pattern2 -> expr2
| pattern3 -> expr3
...

```
- It matches not just against explicit values, but against patterns
- We can use the special symbol `_` to denote that we'd like to match against anything without giving it a name

## Lists

- A list is an arbitrarily long sequence of values
- In OCaml, all elements of a list must have the same type
- Syntax
 

```
[elem1; elem2; ...; elemn]
```
- We can build lists of any element type

## List operations

- Cons
  - Syntax:
 

```
elem1 :: [elem2; elem3; ...; elemn]
```
- Concatenation
  - Syntax:
 

```
[elem1; elem2] @ [elem3; elem4]
```

## Cons and [] are constructors

- `::` is not an operator or function, it is a constructor
- The syntax  
`[elem1; elem2; elem3]`  
is just shorthand for  
`elem1 :: elem2 :: elem3 :: []`  
(`[]` the empty list constructor is called `nil`)
- The second form is what the value actually looks like, the first is just to make it more readable

## Pattern matching revisited

- Pattern matching can match on the structure of values
- Lists have a structure, made up by values and constructors
- In pattern matching, we get new variables, can have any name
- Although we could use arbitrary names in patterns we will follow this naming convention:
  - `x, y, z, ...` for "simple" values
  - `xs, ys, zs, ...` for lists of values  
(motivation: `xs` is the "plural" of `x`)
- The set of patterns must be exhaustive
  - For any value, at least one alternative must match
- Patterns can be
  - `[]`
  - `x::xs`
  - `x::y::[]`
  - `_::xs`
  - `true::_`
  - ...

## Polymorphic functions

- Functions which work for many types are called polymorphic functions
- In O'Caml, we use `'a, 'b, ...` to indicate arbitrary types

## Guarded patterns

- Patterns can have the extended syntax  
`pattern when guard`  
where the guard is an expression of type `bool`
- The alternative will match only when
  - The pattern matches
  - The guard evaluates to true
- These are called guarded patterns
- When using guards, the interpreter can not determine if the pattern matching is exhaustive so in many cases we can drop the guard on the final pattern

## Sorting

- There are many different sorting algorithms
- The insertion sort take one element at a time and insert it into a list of already sorted elements

- There is a predefined function for sorting lists:

```
List.sort
```

## Anonymous functions

- Sometimes a function is only ever used once, and the name does not matter then we can use an anonymous function

- Syntax

```
fun var -> expression
```

## Ready-made functions

- `List.map` : ('a -> bool) -> 'a list -> 'a list
  - `List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f` (not tail-recursive)
- `List.fold_left` : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
  - `List.fold_left f a [b1; ... ; bn]` is `f (... (f (f a b1) b2) ...) bn`
- `List.filter` : ('a -> bool) -> 'a list -> 'a list
  - `List.filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of the elements in the input list is preserved

## Disjoint unions

- We often need to store values that can be one of several different things
- The new type can be used just like the predefined types
- The new values are constants, just like `false`, `true`, `0`, `1`, `2`, ...

- Syntax

```
type name = Name1 [of type]
          | Name2 [of type]
          ...
          | Namen [of type];;
```

- The symbol `Name` is called a constructor
  - Just like constants, constructors can be used in pattern matching

## Pre-defined types

- The pre-defined list type is basically just
 

```
type 'a list = [] | :: of ('a * 'a list);;
```
- `Bool` is defined as
 

```
type bool = false | true;;
```
- `Unit` is defined as
 

```
type unit = ();;
```

## Pre-defined type option

- There is a pre-defined type option with the following definition
 

```
type 'a option = None | Some of 'a;;
```
- This can be used when values are not always well defined