

CS 2SC3 and SE 2S03 Fall 2009

# 02 Functional Programming

William M. Farmer

Department of Computing and Software  
McMaster University

27 September 2009



# Values

- In programming, **values** are the entities that are manipulated by programs.
- Examples:
  - ▶ Booleans.
  - ▶ Numbers (integers and floating point numbers).
  - ▶ Characters and strings.
  - ▶ Tuples.
  - ▶ Lists.
  - ▶ Functions.
- In a programming language, some values are **primitive**, i.e., they are built into the language; others are created as needed by the programmer.

# Common Mathematical Values

There are various ways that common mathematical values are represented in programming languages.

Mathematical	Programming
truth values	booleans
integers	fixnums, bignums
rationals	fixed point, floating point, rationals
real numbers	fixed point, floating point
tuples	tuples, records
finite sequences	lists, arrays
infinite sequences	streams
sets	lists
relations	procedures
functions	procedures

# Expressions

- An **expression** is a syntactic entity that denotes a value.
  - ▶ OCaml Example: `2 + x`
- The value of an expression is obtained by **evaluating** the expression.
- The evaluation of an expression may in some cases cause a program's state to be modified.
  - ▶ This is known as a **side-effect**.

# Statements

- A **statement** is a syntactic entity that states something to be done.
  - ▶ **OCaml Example:** `x := 0`
- The effect of a statement is obtained by **executing** the statement.
- Two statements are usually separated by a semicolon (;).
- An expression with side-effects can be viewed as a statement.
- A statement can be viewed as an expression with side-effects that does not denote a meaningful value.
- In a programming language that is both imperative and functional, statements are considered expressions.
- A purely functional programming language does not use statements (or expressions with side-effects).

# Types

- A **type** is a syntactic entity  $t$  that denotes a set  $s$  of values.
  - ▶  $t$  and  $s$  are often confused with each other.
- OCaml example: The type **int** denotes the set of machine integers.
- Types are used in a variety of ways:
  1. To classify values.
  2. To restrict the values of variables.
  3. To control the formation of expressions.
  4. To classify expressions by their values.
- Types are also used as **mini-specifications**.

# Declarations and Definitions

- A **declaration** states that a name will be used for a particular purpose.
  - ▶ C Example: `int x;`
- A **definition** states that a name is assigned a particular value.
- A definition may include a declaration of the name being defined.
  - ▶ OCaml Example: `let x = 0 ;;`

# Variables

- The meaning of “variable” is different in logic, control theory, and programming.
- In logic, a **variable** is a **symbol** that denotes an **unspecified value**.
  - ▶ Example:  $\forall x . x < x + 1$ .
- In control theory, a **variable** is a **changing value** that is a component of the **state** of a system.
  - ▶ A **monitored variable** is a variable the system can observe but not change.
  - ▶ A **controlled variable** is a variable the system can both observe and change.
- In programming, a **variable** is a **name** bound to a **value**.
  - ▶ OCaml example: `let x = 0 ;;`



# Constants

- The meaning of “constant” is different in logic, control theory, and programming.
- In logic, a **constant** is a **symbol** that denotes a **specified value**.
- In control theory, a **constant** is an **unchanging value**.
- In programming, a **constant** is a name bound to an **immutable value**.
  - ▶ A constant is a read-only variable.
  - ▶ The use of constants is essential for code readability and software maintenance.
- Most variables in OCaml act like constants.
  - ▶ Usually an a variable in OCaml is bound to an immutable value.

# Scope

- The **scope** of a variable  $x$  bound to a value  $v$  is the region of program code in which the binding is effective.
  - ▶ The scope is usually the region of code from the place where  $x$  was first bound to the end of the smallest enclosing “block” of code.
  - ▶ A variable  $x$  is only visible in its scope, i.e., outside of its scope  $x$  will normally not be bound to  $v$ .
- If  $x$  is rebound within its scope, a new scope of  $x$  is created in which the old binding is not visible.
  - ▶ Different variables with different scopes may have the same name.
- A variable is **global** if its scope is unrestricted.
- A variable is **local** if its scope is restricted.
- In accordance with the **Principle of Least Privilege**, the scope of a variable name should be as narrow as possible.

# Global Variable Declarations in OCaml

- A variable can be defined *globally* in OCaml:

```
let x = expr ;;
```

- A global definition of a variable  $x$  masks the previous global definition of  $x$ .
- A set of global definitions can be simultaneous:

```
let x1 = expr1  
and x2 = expr2  
  ⋮  
and xn = exprn ;;
```

# Local Variable Declarations in OCaml

- Variables can also be defined *locally* in OCaml:

```
let x = expr1 in expr2 ;;
```

- A local definition of a variable  $x$  masks all previous local and global definitions of  $x$ .
  - ▶ Previous definitions can come back into effect when the scope of the current definition is exited.
- A set of local definitions can be simultaneous:

```
let x1 = expr1  
and x2 = expr2  
  ⋮  
and xn = exprn  
in expr ;;
```

# Numbers

- A **numeric type** is a type of values representing numbers.
- Most programming language have several numeric types.
- Examples:
  - ▶ A type of **machine integers** or **fixnums** is a finite set of integers that can be represented in a machine word.
  - ▶ A type of **bignums** is the complete infinite set of integers.
  - ▶ A type of **fixed-point numbers** is a finite set of rational numbers expressed in a fixed decimal format.
  - ▶ A type of **floating-point numbers** is a finite set of rational numbers expressed in scientific notation.
  - ▶ A type of **rationals** is the complete infinite set of rationals.
- Each numeric type has its own set of arithmetic operators.
  - ▶ Sometimes the operators are shared between different numeric types.

# OCaml `int` and `float` Types

- `int` is the OCaml type of 31-bit or 63-bit machine integers.
- The arithmetic operators for `int` are:
  - ▶ Addition (`+`).
  - ▶ Unary negation and binary subtraction (`-`).
  - ▶ Multiplication (`*`).
  - ▶ Integer division (`/`).
  - ▶ Division remainder (`mod`).
- `float` is the OCaml type of double-precision floating-point numbers (53-bit coefficient, 11-bit exponent, 1-bit sign).
- The arithmetic operators for `float` are:
  - ▶ Addition (`+.` ).
  - ▶ Unary negation and binary subtraction (`-.`).
  - ▶ Multiplication (`*.` ).
  - ▶ Division (`/.`).
  - ▶ Exponentiation (`**`).

# Booleans

- A **boolean** is a standard truth value, either **true** or **false**.
  - ▶ In OCaml, the truth values are named **true** and **false**.
- The **type of booleans** consists of the two boolean values.
  - ▶ Named **bool** in OCaml.
- A **formula** is an expression of type boolean.
- Propositional operators like  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\Rightarrow$  combine formulas.
- In OCaml, the proposition operators are:
  - ▶ Negation (**not**).
  - ▶ Sequential and (**&&**).
  - ▶ Sequential or (**||**).
- Equations and inequalities are common formulas.
- In OCaml, these are formed using the following operators:  
**=, ==, <>, !=, <, >, <=, >=.**

# Unit Type

- The `unit` type denotes a set of a single member.
- The `()` denotes the single member of `unit`.
- The unit type is used to represent:
  1. Procedures that do not take an input.
  2. Procedures that do not return an output.
- In OCaml, expressions of type `unit` represent `statements`.



# Characters and Strings

- A **character** is a member of an **alphabet**.
- The **ASCII alphabet** contains 128 characters corresponding to the integers  $0, \dots, 127$ .
  - ▶ A “text file” is usually a file of ASCII characters.
- In OCaml, the character type **char** corresponds to the integers  $0, \dots, 255$  with the ASCII characters being the first 128 members of the type.
- A **string** is a finite sequence “ $c_1 \cdots c_n$ ” of characters.
- The **empty string** is the string “” of no members.
- In OCaml, the string type **string** is the set of strings of characters of **char** with length  $\leq 2^{24} - 6$ .
- Strings are appended with a concatenation operator (**^**).
  - ▶ OCaml example: **"abc" ^ "XYZ" = "abcXYZ"**.

# Tuples

- An  ***$n$ -tuple*** is finite sequence  $(a_1, \dots, a_n)$  of values of possibly different types.
- A ***cartesian product***  $A_1 \times \dots \times A_n$  is the set of  $n$ -tuples  $(a_1, \dots, a_n)$  such that  $a_i \in A_i$  for all  $i$  with  $1 \leq i \leq n$ .
- In OCaml, the cartesian product of types  $t_1, \dots, t_n$  is written

$$t_1 * \dots * t_n.$$

- A ***pair*** is a 2-tuple  $(a_1, a_2)$ .
- The OCaml function **`fst`** [**`snd`**] selects the first [second] member of a pair.
  - ▶ Note: Use pattern matching for selecting from longer tuples.
- Tuples can be implemented as ***records***.

# Lists

- A **list** is a finite sequence  $[a_1, \dots, a_n]$  of values of the same type.
- The **empty list** is the list  $[]$  of no members.
- In OCaml, a list  $[a_1, \dots, a_n]$  is written

$[a_1 ; \dots ; a_n]$

and the type of lists of type  $t$  is written

$t$  **list**

- Operators on lists:
  - ▶  $\text{cons}(a, [b_1, \dots, b_n]) = [a, b_1, \dots, b_n]$  (infix **::**).
  - ▶  $\text{append}([a_1, \dots, a_n], [b_1, \dots, b_n]) = [a_1, \dots, a_n, b_1, \dots, b_n]$  (infix **@**).
  - ▶  $\text{head}([a_1, \dots, a_n]) = a_1$  if  $n \geq 1$  (**List.hd**).
  - ▶  $\text{tail}([a_1, \dots, a_n]) = [a_2, \dots, a_n]$  if  $n \geq 1$  (**List.tl**).
- Lists can be implemented as **arrays**.

# Conditionals

- A **conditional** is an expression  $\text{if}(A, b, c)$  where:
  1.  $A$  is a formula.
  2.  $b$  and  $c$  are expressions of the same type.
- The value of  $\text{if}(A, b, c)$  is  $b$  if  $A$  is true and is  $c$  if  $A$  is false.
- In OCaml,  $\text{if}(A, b, c)$  is written as

`if A then b else c`

# Pattern Matching

- Pattern matching is a simple, but powerful, programming method that facilitates (1) **conditional execution** and (2) **value retrieval**.
- The basic form for pattern matching in OCaml is:

```
match expr with  
| p1 -> expr1  
  ⋮  
| pn -> exprn
```

- A **pattern** in an informal expression *p* that may contain variables.
  - ▶ Each variable may occur at most once in *p*.
  - ▶ `_` is the **wildcard pattern**.
  - ▶ *p*<sub>1</sub> | ... | *p*<sub>*n*</sub> is a compound pattern.
  - ▶ A name may be bound to a pattern (*p* **as** *x*).
  - ▶ A pattern may be guarded by a condition (*p* **when** *cond*).

# (Unary) Functions

- **Definition 1:** A **function** is a rule  $f : I \rightarrow O$  that associates members of  $I$  (inputs) with members of  $O$  (outputs).
  - ▶ Every input is associated with at most one output.
  - ▶ Some inputs may not be associated with an output.  
Example:  $f : \mathbf{Z} \rightarrow \mathbf{Q}$  where  $x \mapsto 1/x$ .
- **Definition 2:** A **function** is a set  $f \subseteq I \times O$  such that if  $(x, y), (x, y') \in f$ , then  $y = y'$ .
- Each function  $f$  has a **domain**  $D \subseteq I$  and a **range**  $R \subseteq O$ .
  - ▶  $f$  is **total** if  $D = I$  and **partial** if  $D \subset I$ .

# Lambda Notation

- **Lambda notation** is a precise, convenient way to define functions.
- If  $B$  is an expression of type  $t$ ,

$$\lambda x : s . B$$

denotes a function  $f : s \rightarrow t$  such that  $f(a) = B[x \mapsto a]$ .

- **Example:** Let  $f = \lambda x : \mathbf{R} . x * x$ .
  - ▶  $f(2) = (\lambda x : \mathbf{R} . x * x)(2) = 2 * 2$ .
  - ▶  $f$  denotes the squaring function.
- In OCaml, a function  $\lambda x : t . x * x$  is written as

```
function (x : t) -> x * x
```

or more simply as

```
function x -> x * x
```

# Procedures

- A **procedure** is a unit of code that implements a function.
- Procedures are also called **functions**, **subroutines**, and **methods**.
  - ▶ Procedures are called **functions** in both OCaml and C.
- Unlike mathematical functions, procedures can have **side-effects**.
- A procedure may take no input or return no output and be useful only by virtue of its side-effects.



# $n$ -Ary Functions

- **Definition 1:** For  $n \geq 0$ , an  $n$ -ary function is a rule  $f : I_1, \dots, I_n \rightarrow O$  that associates members of  $I_1, \dots, I_n$  (inputs) with members of  $O$  (outputs).
  - ▶ Every list of inputs is associated with at most one output.
  - ▶ Some lists of inputs may not be associated with an output.
- **Definition 2:** For  $n \geq 0$ , an  $n$ -ary function is a set  $f \subseteq I_1 \times \dots \times I_n \times O$  such that if  $(x_1, \dots, x_n, y), (x_1, \dots, x_n, y') \in f$ , then  $y = y'$ .
- Each function  $f$  has a **domain**  $D \subseteq I_1 \times \dots \times I_n$  and a **range**  $R \subseteq O$ .

# Representing $n$ -Ary Functions as Unary Functions

There are two ways of representing a  $n$ -ary function as a unary function:

1. As a function on tuples:  $f : I_1, \dots, I_n \rightarrow O$  is represented as

$$f' : I_1 \times \dots \times I_n \rightarrow O$$

where

$$f(x_1, \dots, x_n) = f'((x_1, \dots, x_n)).$$

2. As a curried function:  $f : I_1, \dots, I_n \rightarrow O$  is represented as

$$f'' : I_1 \rightarrow (I_2 \rightarrow (\dots (I_n \rightarrow O) \dots))$$

where

$$f(x_1, \dots, x_n) = f''(x_1) \dots (x_n).$$

# Example

- Let  $f = \lambda x, y : \mathbf{R} . x^2 + y^2$ .
- $f' = \lambda p : \mathbf{R} \times \mathbf{R} . [\text{fst}(p)]^2 + [\text{snd}(p)]^2$ .  
$$\begin{aligned} f'((a, b)) &= (\lambda p : \mathbf{R} \times \mathbf{R} . [\text{fst}(p)]^2 + [\text{snd}(p)]^2)((a, b)) \\ &= [\text{fst}((a, b))]^2 + [\text{snd}((a, b))]^2 \\ &= a^2 + b^2. \end{aligned}$$
- $f'' = \lambda x : \mathbf{R} . \lambda y : \mathbf{R} . x^2 + y^2$ .  
$$\begin{aligned} f''(a)(b) &= (\lambda x : \mathbf{R} . \lambda y : \mathbf{R} . x^2 + y^2)(a)(b) \\ &= (\lambda y : \mathbf{R} . a^2 + y^2)(b) \\ &= a^2 + b^2. \end{aligned}$$

# Functions in OCaml

- All functions in OCaml are **unary**.
- $n$ -ary functions are represented as **curried functions**.
- A function may be **higher-order**, i.e., it may take other functions as input.
- A function may be defined **recursively**.
- Functions are stored as **closures** consisting of:
  1. The name of the function's formal parameter.
  2. The body of the function.
  3. The environment of the name-value bindings in which the function was defined.
- The form

**let**  $f\ p_1 \cdots p_n = \text{expr}$

is equivalent to the form

**let**  $f =$

**function**  $p_1 \rightarrow \cdots \rightarrow \text{function } p_n \rightarrow \text{expr}$

# Recursion

- **Recursion** is a method of defining something in terms of itself.
  - ▶ One of the most fundamental ideas of computing.
  - ▶ An alternative to iteration (loops).
  - ▶ Can make some programs easier to describe, write, and prove correct.
- Both procedures and data structures can be defined by recursion.
- A set of procedures or data structures can be defined by **mutual recursion**.
- The use of recursion requires care and understanding.
  - ▶ Recursive definitions can be nonsensical (i.e., nonterminating).
  - ▶ Sloppy use of recursion can lead to total confusion.
  - ▶ Correctness is proved by **induction**.

# Recursion in OCaml

- The form to define a function  $f$  by recursion is:

`let rec  $f$   $p_1 \cdots p_n$  = expr`

- The form to define functions  $f_1, f_2, \dots, f_n$  by mutual recursion is:

`let rec  $f_1$   $p_1 \cdots p_n$  = expr1  
and  $f_2$   $p_1 \cdots p_n$  = expr2  
⋮  
and  $f_n$   $p_1 \cdots p_n$  = expr $n$`

- Values other than functions may also be defined by recursion.
- Local definitions may be recursive.

# Pattern Matching in OCaml Function Definitions

- Pattern matching can be utilized in OCaml function definitions.
- The form

`function p1 -> expr1 | ... | pn -> exprn`

is equivalent to the form

`function expr -> match expr with  
p1 -> expr1 | ... | pn -> exprn`

- Examples:

- ▶ `let pair_rev (x,y) = (y,x) ;;`
- ▶ `let true_value () = true ;;`
- ▶ `let rec list_rev = function  
| [] -> []  
| head :: tail -> (list_rev tail) @ [head] ;;`

# Temporal Distinctions in Programming

- There are three main programming time periods:
  1. **Design time** is the time period during which a program is **written**.
  2. **Compile time** is the time period during which a program is **compiled**.
  3. **Run time** is the time period during which a program is **executed**.
- Requirements, decisions, checks, errors, etc. are often associated with one of these three time periods.



# Types in OCaml

- OCaml is a strictly typed language.
- Each OCaml expression is assigned a unique **static** type.
  - ▶ A type is **static** if it is determined at **compile time**.
  - ▶ A type is **dynamic** if it is determined at **run time**.
- OCaml is **type safe** — type failures cannot occur when a type-checked OCaml expression is evaluated.
- Types may be parameterized by **type variables**.
- An expression of a parameterized type is **polymorphic** — which means it can be used in contexts that require different types.
- Most types do not have to be explicitly declared in expressions.
  - ▶ The missing type declarations are **inferred** if possible.
- A expression  $e$  of a parameterized type or unspecified type can be **constrained** to a particular type  $t$  using the syntax  $(e : t)$

# Summary

- This topic has very briefly presented the basic ideas of the **functional programming paradigm** and has shown how these ideas are implemented in OCaml.
- We know enough now about OCaml to do some serious programming because OCaml supports the functional programming paradigm.
- The next topic will present the basic ideas of the **imperative programming paradigm**.
- After we finished the next topic, we will know enough about OCaml to program in the imperative style.