CS 2SC3 and SE 2S03 Fall 2008

02 Simple Procedures

William M. Farmer

Department of Computing and Software
McMaster University

30 October 2008

Values

- In programming, values are the entities that are manipulated by programs.
- Examples:
  - Numbers (integers and floating point numbers).
  - Booleans.
  - 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.

Expressions

- An expression is a syntactic entity that denotes a value.
  - C Example: `2 + x`
  - 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.
  - C Example: `x = 0;`
  - 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).

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`

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.

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.

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.

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: `=`, `==`, `<>`, `!=`, `<`, `>`, `<=`, `>=`.

Characters and Strings

- A character is a member of an alphabet.
- The ASCII alphabet contains 128 characters corresponding to the integers $0, \ldots, 127$.
  - A "text file" is usually a file of ASCII characters.
- In OCaml, the character type `char` corresponds to the integers $0, \ldots, 255$ 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 $(a_1, \ldots, a_n)$ of values of possibly different types.
- A cartesian product $A_1 \times \cdots \times A_n$ is the set of n-tuples $(a_1, \ldots, a_n)$ such that $a_i \in A_i$ for all i with $1 \leq i \leq n$.
- In OCaml, the cartesian product of types $t_1, \ldots, t_n$ is written

  $$t_1 * \cdots * t_n.$$

- A pair is a 2-tuple (a_1, a_2).
- The OCaml function `fst` [`snd`] chooses the first [second] member of a pair.
- Tuples can be implemented as records.

Lists

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

  $[a_1 ; \ldots ; a_n]$

  and the type of lists of type t is written

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

Conditionals

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

  `if` A `then` b `else` c.

(Unary) Functions

- Definition 1: A function is a rule $f : I \to O$ 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: $f : \mathbf{Z} \to \mathbf{Q}$ 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 $f : s \longrightarrow t$ 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.
- Unlike mathematical functions, procedures can have side-effects.
- A procedure may produce 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 $f : I_1, \ldots, I_n \rightarrow O$ that associates members of $I_1, \ldots, I_n$ (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 $f \subseteq I_1 \times \cdots \times I_n \times O$ such that if $(x_1, \ldots, x_n, y), (x_1, \ldots, x_n, y') \in f$, then $y = y'$.
- Each function f has a domain $D \subseteq I_1 \times \cdots \times I_n$ 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: $f : I_1, \ldots, I_n \longrightarrow O$ is represented as

   $$f' : I_1 \times \cdots \times I_n \longrightarrow O$$

   where

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

2. As a curryed function: $f : I_1, \ldots, I_n \longrightarrow O$ is represented as

   $$f'' : I_1 \longrightarrow (I_2 \longrightarrow (\cdots (I_n \longrightarrow O) \cdots))$$

   where

   $$f(x_1, \ldots, x_n) = f''(x_1) \cdots (x_n).$$

Example

- Let $f = \lambda\, x, y : \mathbf{R} \,.\, x^2 + y^2$.
- $f' = \lambda\, p : \mathbf{R} \times \mathbf{R} \,.\, [\mathsf{fst}(p)]^2 + [\mathsf{snd}(p)]^2$.

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

- $f'' = \lambda\, x : \mathbf{R} \,.\, \lambda\, y : \mathbf{R} \,.\, x^2 + y^2$.

$$\begin{aligned} f''(a)(b) &= (\lambda\, x : \mathbf{R} \,.\, \lambda\, y : \mathbf{R} \,.\, x^2 + y^2)(a)(b) \\ &= (\lambda\, y : \mathbf{R} \,.\, a^2 + y^2)(b) \\ &= a^2 + b^2. \end{aligned}$$

Functions in OCaml

- All functions in OCaml are unary.
- n-ary functions are represented as curryed 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 syntax

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

  is a short form for

  `let` f `=`
  `function` p_1 `->` $\cdots$ `->` `function` p_n `->` 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 syntax to define a function f by recursion is:

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

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

  `let rec` $f_1\ p_1 \cdots p_n$ `=` expr_1
  `and` $f_2\ p_1 \cdots p_n$ `=` expr_2
  $\vdots$
  `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.

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 $(e : t)$.

Executing OCaml: Toplevel System

- The toplevel system for OCaml is an interactive read-eval-print loop.
- The toplevel system is started by the command `ocaml`.
- OCaml phrases are repeatedly read, type-checked, compiled, executed, and then the results of the execution are printed.
- A list of OCaml phrases can be executed as a script.

Executing OCaml: Native Code Compilation

- The OCaml native-code compiler `ocamlopt` compilers OCaml source code files to native code object files and links these object files to produce standalone executables.
- Example: `ocamlopt -o nc-prog prog.ml`
- Native code compilation results in slower compilation time, faster run time.

Executing OCaml: Bytecode Compilation

- The OCaml bytecode compiler `ocamlc` compiles OCaml source files to bytecode object files and links these object files to produce standalone bytecode files.
- Example: `ocamlc -o bc-prog prog.ml`
- A standalone bytecode file can be executed by the OCaml bytecode interpreter `ocamlrun`.
- Bytecode compilation results in faster compilation time, slower run time.

C: Variables

- A variable in C is bound to a location that can hold a value of a certain type.
  - For example, a variable of type `int` is bound to a memory location that can hold a value of type `int`.
- Hence, every variable in C has:
  - A name.
  - A type.
  - A location (its direct value).
  - A value (its indirect value).
- The statement

  ```
  int i;
  ```

  declares a variable with name `i` and type `int`.
- The statement

  ```
  const int two = 2;
  ```

  declares a constant with name `two`, type `int`, and value `2`.

C: The Structure of a Simple Program

- Here is a simple C program:

```
# include <stdio.h>
int main ()
{
  printf("Hello!\n");
  return 0;
}
```

- `main` is the procedure that starts the execution when the program is invoked.
- `main` takes no input and returns a value of type `int` as output.

C: Native Code Compilation

- A C native-code compiler such as `gcc` compilers C source code files to native code object files and links these object files to produce standalone executables.
- Example: `gcc -o prog prog.c`

C: Numbers

- C contains several primitive numeric types; the most important are:
  - `char`, a type of machine integers representing characters.
  - `int`, a type of medium-size machine integers.
  - `double`, a type of double-precision floating point numbers.
- The sizes of numeric types in C vary across hardwares and compilers.
- Numeric types of C are not strictly typed:
  - The types share a common set of arithmetic operators (`+`, `-`, `*`, `/`, `%`).
  - Numeric values of the wrong type are automatically coerced to the right type.
- Danger: Numeric value coercion may lead to incorrect or unexpected results.

C: Booleans

- C does not have a primitive boolean type.
- The standard C library with header `<stdbool.h>` provides a type `bool` with expressions `true` and `false` denoting the two truth values.
- In C, the proposition operators are:
  - Negation (`!`).
  - Sequential and (`&&`).
  - Sequential or (`||`).
- C has the following set of operators for forming equations and inequalities: `==`, `!=`, `<`, `>`, `<=`, `>=`.

C: Conditionals

- In C, a conditional expression if(A, b, c) is written as

  A `?` b `:` c

- In C, a conditional statement "If A then do b else do c" is written as

```
if A
  b;
else
  c;
```

- The else part of the conditional statement is optional.

C: Procedures (1/2)

- Procedures in C are called functions.
- The definition of a function has the following form:

  $t\ f\ (t_1\ p_1, \ldots, t_n\ p_n)$
  {
  B
  }

- t is the type of the value that is returned by a `return` statement in the body B.
- f is the name of the function.
- $t_1\ p_1, \ldots, t_n\ p_n$ is the parameter list of the function. t_i is the type of the parameter p_i.
- B is the body of the function consisting of a list of definitions and statements.
- $t\ f\ (t_1\ p_1, \ldots, t_n\ p_n)$; is the function header or function prototype for the function.

C: Procedures (2/2)

- A function prototype for a function f declares f with its type (which is given indirectly).
- Every function in C must be declared before it can be applied. (Synonyms for "applied" are "called" and "invoked".)
- Unlike OCaml, all executable code in a C program is contained in some function body.
- Unlike OCaml, a C function cannot be defined inside another function.
- Unlike OCaml, the application of a C function is only weakly type checked.
- Like OCaml, functions in C can be defined by recursion (but tail-recursive functions are usually not executed in constant space).

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 and C.
- 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 and C to program in the imperative style.