Benjamin Fenster
CS-103 (Skalka)
5 September 2003

Notes - OCaml

I. Overview
   a. Dialect of (standard) ML
      i. ML was originally developed in the late 70s, early 80s.
      ii. Meta Language for theorem provers (ML)
      iii. Also Caml, Caml-Light
      iv. In France, Caml is used to teach in schools: the equivalent of Pascal.
   b. Features
      i. Functional Language
      ii. Based on evaluation of expressions, not mutation of variables.
      iii. Strongly typed
      iv. Has type inference algorithm
      v. Type system is static
         1. Static: at compile time
         2. As opposed to dynamic (relevant to runtime)

II. Functional Languages
   a. In Procedural Languages
      i. Define variables (like boxes)
      ii. Change the contents of the boxes
      iii. Everything is about what's stored in variables
      iv. Example
         1. x = 1 + 2;
         2. First (1 + 2) is evaluated.
         3. Then throw it in the box.
   b. Functional Languages
      i. 1 + 2 by itself is the statement.
      ii. Don't throw anything into any boxes!

III. Fundamentals and Course Notation
   a. OCaml programs are defined in terms of expressions
      i. Expressions are always denoted e
      ii. Every OCaml expression e…
         1. Has a type τ, denoted e: τ
            a. Otherwise, e is rejected (as being illegal)
            b. Any program that's not well-typed may be unstable.
            c. Well-typed expressions in OCaml won't go wrong
               i. May not work semantically
               ii. Won't do anything horrid like dump core
         2. May evaluate to a value (V), denoted e ⇩ v
         3. May also have an effect
            a. Something happening behind the scenes
            b. I/O, mutation of static (assignment)
            c. Purely functional OCaml: A subset of the language without any effects.
   b. Type
      i. A set of values
      ii. Includes a set of operations on values of that type.

IV. Basic Types
   a. int
      i. Values: … -3, -2, -1, 0, 1, 2, 3, …
      ii. Operations:
         1. Arithmetic Operations
         2. (e1 + e2) : int iff e1 : int and e2 : int
         3. (e1 * e2) : int iff e1 : int and e2 : int
         4. et cetera

iii. Evaluation
   1. e1 + e2 ⇩ “n1 + n2” iff e1 ⇩ n1 and e2 ⇩ n2
   2. NB: “e” denotes the meaning of e in our idealized mathematical universe
iv. Example
   1. 2 * 7 : int
   2. 2 * 7 ⇩ 14
   3. (2 * 7) + 4 : int
   4. (2 * 7) + 4 ⇩ 14
   5. NB: All arithmetic operations have built-in precedence, also known as binding strength, where ≤ means “binds weaker than”
   6. + ≤ div ≤ *
   7. Note that we write these expressions and then evaluate them. There’s no “assignment” component.

b. float
   i. Values:
      1. -1.0, 2.34, et cetera : float.
      2. 1 : int (not float). 1.0 : float
   ii. Operations
      1. Arithmetic Operations
      2. e1 +. e2 : float iff e1 : float and e2 : float
      3. The dot in +. means “for floats”
   iii. Evaluation (similar to int type)
   iv. Arithmetic operations are not overloaded

c. Unit
   i. Value
      1. () : unit
      2. Just one value
   ii. No operations
   iii. Useful application might be to write a function with a dummy argument.

d. char, string
   i. Values
      1. ‘a’, ‘b’, ‘c’, … : char
      2. “hello”, “hi”, “ho” : string
   ii. Operations
      1. Many operations not worth discussing in class
      2. e_1 ^ e_2 ⇩”s_1s_2” iff e_1 ⇩ “s_1” and e_2 ⇩ “s_2”

e. bool
   i. Values
      1. true, false : bool
      2. Genuine Boolean
   ii. Operations
      1. e_1 && e_2 : bool iff e_1 : bool, e_2 bool
      2. e_1 || e_2 : bool iff e_1 : bool, e_2 bool
      3. not e : bool iff e : bool
   iii. Evaluation
      1. && is conjunction
      2. || is conjunction
      3. not is negation

V. Program Errors
   a. Syntax Errors (e.g. missing delimiters, mismatched quotes, et cetera)
   b. Type Errors
      i. Syntactically correct, but has error in terms of type
      ii. Predict ill behavior at runtime.
      iii. 2 + 2.3 is rejected as not well typed
      iv. 2 +. 2.3 is also rejected

v. float(2) + 2.3 is well-typed
vi. This is not type casting because the value Is really changed, we're not just "pretending" it's a float for a moment (as is the case in type casting)
vii. Example
1. 3 div 0 : int
2. Will generate exception at runtime.
3. 3 / 0 ⇩ raise division-by-zero error

c. Semantic Errors
i. Programmer wanted the program to do one thing but it does something different.
ii. "Programmer Error"

d. Runtime Errors
i. Array out of bounds subscripting, et cetera
ii. Type errors predict ill behavior at runtime, so these don't exist in OCaml
iii. Note that this does not address exceptions, but genuinely bad behavior (core dump, et cetera)

VI. Interacting with OCaml
a. At CS unix prompt, type "ocaml"
b. Get # prompt. Enter OCaml expressions.
c. Delimit expressions with double semicolon
d. # print_string "Hello World";;
e. Will return with type and value of expression
f. Example
i. # (2 + 4) * 3;;
ii. : int = 18
g. Reading, Writing Files
i. # #use "eqn.ml";;
ii. By convention, source code written in .ml files
h. Use Ctrl+D to exit, or exit 0;;

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: