**_Benjamin Fenster_**
CS-103 (Skalka)
13 September 2003

# Notes – Complex OCaml Expressions

I. Conditionals and Relationals
   a. Relational Operations
      i. These use other types, yield bool
      ii. Operations
         1. $e_1$ `=` $e_2$ `: bool`
         2. $e_1$ `<>` $e_2$ `: bool`
         3. $e_1$ `>` $e_2$ `: bool`
         4. $e_1$ `>=` $e_2$ `: bool`
         5. $e_1$ `<` $e_2$ `: bool`
         6. $e_1$ `<=` $e_2$ `: bool`
      iii. Note that the = operator is not assignment – it's equality!
      iv. All are overloaded for int, float, string, char, and unit, except that the type of the two operands must agree.
      v. Examples
         1. `1 = 2 : bool`
         2. `1 = 2` ⇓ `false`
         3. `1 = true` (nonsensical)
      vi. Evaluation
         1. Interpretation is obvious for numbers
         2. For char and string types, uses dictionary ordering based on ASCII values.
         3. Examples
            a. "aa" < "ab" ⇓ true
            b. "aaa" < "ab" ⇓ true
            c. "aa" < "aaa" ⇓ true
            d. "aa" < "aba" ⇓ true
   b. Conditionals
      i. If… Then (conditional branching)
      ii. if e then $e_1$ else $e_2$ : $\tau$ iff e : bool and $e_1 : \tau$ and $e_2 : \tau$
      iii. Examples
         1. `if 1 = 0 then 5 else 3 : int`
         2. `if 1 = 0 then 5 else true` (nonsensical)
      iv. Note that the last example would always be valid since it would always yield "true" (there's no chance 5 would result)
      v. We guarantee well-typed expressions are safe, but expressions that aren't well typed may or may not be okay.
      vi. We end up throwing away some good expressions in exchange for a better guarantee
      vii. This is an important point that we'll discuss later.

II. Declarations of Variables
   a. Declarations
      i. Values and types are associated with names via <u>declarations,</u> which <u>bind</u> values and types to <u>names</u> in <u>environments</u>.
      ii. It's not a box, it's a name for a value.
      iii. Variable names
         1. Sequences of letters, numbers, and _ characters. They must begin with a lowercase letter or an underscore.
         2. We will let x range over variable names.
      iv. Environments
         1. Also, "value environment"

        2. A lookup table that associates a collection of variable names with values.
        3. Each entry is a binding: x = v.
        4. Once you define a name in a dictionary its definition sticks.
    v. Type Environment
        1. Like the value environment, but entries are type bindings.
        2. $x = \tau$

b. Value Binding
    i. Form of declarations: let $x : \tau = e$
    ii. Example
        1. `let two : int = 1 + 1;;`
        2. `two + 5` ⇓ `7`
    iii. Bindings are always type checked
        1. let $x : \tau = e$
        2. Type check e, say $e : \tau'$
        3. Make sure $\tau' = \tau$
        4. Add $x : \tau$ to the top-level environment.
    iv. Evaluation
        1. First evaluate $e \Downarrow v$
        2. Then add x = v to the top-level environment.
        3. Note: e is evaluated *before* adding it to the environment.
        4. That means the variable you're naming isn't in scope when you're declaring it.
    v. Example
        1. `let x : int = 1;;`
        2. `let y : int = 2;;`
        3. `x + y;;` ($x + y \Downarrow 3$)
        4. `x = 3;;` ($x = 3 \Downarrow$ false)
        5. Note that x = 3 is NOT an assignment.
    vi. Variables don't vary!

c. Shadowing
    i. The most recent declaration of a variable overrides (shadows) all previous bindings
    ii. Example
        1. `let x : int = 5;;`
        2. `let x : int = 7;;`
        3. $x = 5 \Downarrow$ false
    iii. There's no reason you can't re-declare the same variable with a different type. The new variable still shadows the earlier declaration

d. Scope
    i. Localization of declarations is possible
    ii. Done with let expressions
    iii. let $x : \tau = e_1$ in $e_2$
    iv. Localizes the definition of x to just $e_2$
    v. Example
        1. `let x : int = 5;;`
        2. `let x : int = 1 in 2 * x` ⇓ `2`
        3. $x = 5 \Downarrow$ true
    vi. Type checking
        1. First type check let $x : \tau = e_1$ the same way as before
        2. Then *temporarily* extend the type environment with $x : \tau$
        3. Now type check $e_2$ in the extended environment, yield $e_2 : \tau'$
        4. *Retract* $x : \tau$ binding from the type environment, yield $\tau'$ as the type of the whole expression.

vii. Evaluation
    1. First evaluate $e_1$ to v ($e_1 \Downarrow v$)
    2. *Temporarily* extend the environment with x = v
    3. Then evaluate $e_2$ in the extended environment, yield v'
    4. *Retract* the binding x = v from the environment
    5. Yield v' as the value of the whole expression.

viii. Definition: The scope of x in let x : $\tau = e_1$ in $e_2$ is $e_2$

ix. Environments have stack-like behavior

x. Example – `let x : int = 2 in (let x : int = x + x in 2 * x) + x` ⇓ `10`

ERROR: undefinedfilename
OFFENDING COMMAND: </FONT>

STACK: