CMSC 330: Organization of Programming Languages

OCaml Expressions and Functions
Our focus: semantics and idioms for OCaml
   – Semantics is what the language does
   – Idioms are ways to use the language well

We will also cover some useful libraries

Syntax is what you type, not what you mean
   – In one lang: Different syntax for similar concepts
   – Across langs: Same syntax for different concepts
   – Syntax can be a source of fierce disagreement among language designers!
Expressions

• **Expressions** are our primary building block
  – Akin to *statements* in imperative languages

• Every kind of expression has
  – **Syntax**
    • We use metavariable $e$ to designate an arbitrary expression
  – **Semantics**
    • Type checking rules (static semantics): produce a type or fail with an error message
    • Evaluation rules (dynamic semantics): produce a value
      – (or an exception or infinite loop)
      – Used *only* on expressions that type-check
Values

• A value is an expression that is final
  – Evaluating an expression means running it until it becomes a value
  – We use metavariable $v$ to designate an arbitrary value

• 34 is a value, true is a value

• 34+17 is an expression, but not a value
  – It evaluates to 51
Types

- **Types** classify expressions
  - Characterize the set of possible values an expression could evaluate to
  - We use metavariable \( t \) to designate an arbitrary type
    - Examples include `int`, `bool`, `string`, and more.
- Expression \( e \) has type \( t \) if \( e \) will (always) evaluate to a value of type \( t \)
  - \{ ..., –1, 0, 1, ... \} are values of type `int`
  - `34+17` is an expression of type `int`, since it evaluates to `51`, which has type `int`
  - Write \( e : t \) to say \( e \) has type \( t \)
If Expressions

• Syntax
  - if $e_1$ then $e_2$ else $e_3$

• Evaluation
  - If $e_1$ evaluates to true, and if $e_2$ evaluates to $v$, then if $e_1$ then $e_2$ else $e_3$ evaluates to $v$
  - If $e_1$ evaluates to false, and if $e_3$ evaluates to $v$, then if $e_1$ then $e_2$ else $e_3$ evaluates to $v$

• Type checking
  - If $e_1$ has type bool and $e_2$ has type $t$ and $e_3$ has type $t$ then if $e_1$ then $e_2$ else $e_3$ has type $t$
If Expressions

• Syntax
  - if e1 then e2 else e3

• Evaluation
  - If e1 evaluates to true, and if e2 evaluates to v, then if e1 then e2 else e3 evaluates to v
  - If e1 evaluates to false, and if e3 evaluates to v, then if e1 then e2 else e3 evaluates to v

• Type checking
  - If e1 : bool and e2 : t and e3 : t then if e1 then e2 else e3 : t
If Expressions

• Syntax
  – if e1 then e2 else e3

• Evaluation
  – If e1 evaluates to true, and if e2 evaluates to v, then if e1 then e2 else e3 evaluates to v
  – If e1 evaluates to false, and if e3 evaluates to v, then if e1 then e2 else e3 evaluates to v

• Type checking
  – If e1 : bool and e2 : t and e3 : t then (if e1 then e2 else e3) : t
If Expressions: Examples

# if 7 > 42 then "hello" else "goodbye";;
- : string = "goodbye"

# if true then 3 else 4;;
- : int = 3

# if false then 3 else 3.0;;

This expression has type float but is here used with type int
Quiz 1

To what value does this expression evaluate?

\[
\text{if } 22=0 \text{ then 2 else 1}
\]

A. 0
B. 1
C. 2
D. none of the above
Quiz 1

To what value does this expression evaluate?

if 22=0 then 2 else 1

A. 0  
B. 1  
C. 2  
D. none of the above
Quiz 2

To what value does this expression evaluate?

```plaintext
if 22=0 then "hello" else 2
```

A. 0  
B. 1  
C. 2  
D. none of the above
Quiz 2

To what value does this expression evaluate?

if 22=0 then "hello" else 2

A. 0
B. 1
C. 2
D. none of the above: doesn’t type check so never gets a chance to be evaluated
Function Definitions

- OCaml functions are like mathematical functions
  - Compute a result from provided arguments

```
(* requires n>=0 *)
(* returns: n! *)
let rec fact n =
  if n = 0 then
    1
  else
    n * fact (n-1)
```

- Use (* *) for comments (may nest)
- Parameter (type inferred)
- rec needed for recursion
- Structural equality
- Line breaks, spacing ignored (like C, C++, Java, not like Ruby)
Function Types

• In OCaml, \(\to\) is the function type constructor
  – Type \(t1 \to t\) is a function with argument or domain type \(t1\) and return or range type \(t\)
  – Type \(t1 \to t2 \to t\) is a function that takes two inputs, of types \(t1\) and \(t2\), and returns a value of type \(t\). Etc.

• Examples
  – let next x = x + 1 (* type int \(\to\) int *)
  – let fn x = (int_of_float x) * 3
    (* type float \(\to\) int *)
  – fact (* type int \(\to\) int *)
Type Checking Functions

• Syntax  
  \[ \text{let rec } f \ x_1 \ldots \ x_n = e \]

• Type checking
  – Conclude that \( f : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u \) if \( e : u \) under the following assumptions:
    • \( x_1 : t_1, \ldots, x_n : t_n \) (arguments with their types)
    • \( f : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u \) (for recursion)

• Example
  – Given \( n : \text{int}, \text{fact} : \text{int} \rightarrow \text{int} \)
  – Does \( \text{if } n = 0 \text{ then } 1 \ldots : \text{int} \) ?
    • It does!
  – Conclude \( \text{fact} : \text{int} \rightarrow \text{int} \)

\[
\text{let rec fact } n = \\
\text{ if } n = 0 \text{ then } 1 \\
\text{ else } n \ast \text{ fact } (n-1)
\]
Calling Functions

• Syntax $f\ e_1\ ...\ e_n$
  – Parentheses not required around argument(s)
  – No commas; use spaces instead

• Type checking
  – If $f: t_1 \rightarrow \ ... \rightarrow t_n \rightarrow u$ and $e_1: t_1, \ ... , e_n: t_n$
    then $f\ e_1\ ...\ e_n: u$

• Example:
  – $\text{fact}\ 1: \text{int}$
  – since $\text{fact}: \text{int} \rightarrow \text{int}$ and $1: \text{int}$

• Function call aka function application
Calling Functions

• Syntax \( f \, e_1 \ldots \, e_n \)

• Evaluation
  – Evaluate arguments \( e_1 \ldots \, e_n \) to values \( v_1 \ldots \, v_n \)
    • Order is actually right to left, not left to right
    • But this doesn’t matter if \( e_1 \ldots \, e_n \) don’t have side effects
  – Find the definition of \( f \)
    • \( \text{let rec } f \, x_1 \ldots \, x_n = e \)
  – Substitute \( v_i \) for \( x_i \) in \( e \), yielding new expression \( e' \)
  – Evaluate \( e' \) to value \( v \), which is the final result
let rec fact n =
  if n = 0 then
      1
  else
      n * fact (n-1)

Example evaluation

• fact 2
  ➢ if 2=0 then 1 else 2*fact(2-1)
  ➢ 2 * fact 1
  ➢ 2 * (if 1=0 then 1 else 1*fact(1-1))
  ➢ 2 * 1 * fact 0
  ➢ 2 * 1 * (if 0=0 then 1 else 0*fact(0-1))
  ➢ 2 * 1 * 1
  ➢ 2
Type Annotations

• The syntax \((e : t)\) asserts that “\(e\) has type \(t\)”
  – This can be added anywhere you like
    
    ```
    let (x : int) = 3
    let z = (x : int) + 5
    ```

• Define functions’ parameter and return types
  ```
  let fn (x:int):float =
      (float_of_int x) *. 3.14
  ```
  – Note special position for return type
  – Thus `let g x:int = ...` means `g` returns `int`
    • Not that \(x\) has type \(int\)

• Checked by compiler: Very useful for debugging
Quiz 3: What is the value of `foo 4 3`?

```plaintext
let rec foo n m =
    if n >= 9 || n<0 then
        m
    else
        n + m + 1
```

- Type Error
- 2
- 8
- 7
Quiz 3: What is the value of \texttt{foo 4 3}?

\begin{verbatim}
let rec foo n m =
  if n >= 9 || n<0 then
    m
  else
    n + m + 1
\end{verbatim}

- Type Error
- 2
- 8
- 7
Quiz 4: What is the value of \texttt{bar 3}

\begin{center}
\begin{minipage}{0.6\textwidth}
\begin{verbatim}
let rec bar(n:int):int =
  if n = 0 || n = 1 then 1
  else bar (n-1) + bar (n-2)
\end{verbatim}
\end{minipage}
\end{center}

\begin{itemize}
  \item Syntax Error
  \item 3
  \item 4
  \item 5
\end{itemize}
Quiz 4: What is the value of $\text{bar } 3$

let rec bar(n:int):int =
  if n = 0 || n = 1 then 1
  else
    bar(n-1) + bar(n-2)

• Syntax Error
• 3
• 4
• 5