Functions

Prof. Clarkson
Summer 2016

Today’s music: Function by E-40 (Clean remix)
Review

Previously in 3110:
• What is a functional language?
• Why learn to program in a functional language?
• Recitation: intro to OCaml (finish those exercises!)

Today:
• **Functions:** the most important part of functional programming!
Five aspects of learning a PL

1. **Syntax**: How do you write language constructs?
2. **Semantics**: What do programs mean? (Type checking, evaluation rules)
3. **Idioms**: What are typical patterns for using language features to express your computation?
4. **Libraries**: What facilities does the language (or a third-party project) provide as “standard”? (E.g., file access, data structures)
5. **Tools**: What do language implementations provide to make your job easier? (E.g., top-level, debugger, GUI editor, …)

• All are essential for good programmers to understand
• Breaking a new PL down into these pieces makes it easier to learn
Our Focus

We focus on **semantics** and **idioms** for OCaml
- **Semantics** is like a meta-tool: it will help you learn languages
- **Idioms** will make you a better programmer in those languages

**Libraries** and **tools** are a secondary focus: throughout your career you’ll learn new ones on the job every year

**Syntax** is almost always boring
  - A fact to learn, like “Cornell was founded in 1865”
  - People obsess over subjective preferences {yawn}
  - Class rule: We don’t complain about syntax
Expressions

*Expressions* (aka *terms*):

- primary building block of OCaml programs
- akin to *statements* or *commands* in imperative languages
- can get arbitrarily large since any expression can contain subexpressions, etc.

Every kind of expression has:

- **Syntax**
- **Semantics:**
  - *Type-checking rules* (*static semantics*): produce a type or fail with an error message
  - *Evaluation rules* (*dynamic semantics*): produce a *value*
    - (or exception or infinite loop)
    - Used only on expressions that type-check
Values

A **value** is an expression that does not need any further evaluation

- **34** is a value of type `int`
- **34+17** is an expression of type `int` but is not a value
IF EXPRESSIONS
if expressions

Syntax:

if e1 then e2 else e3

Evaluation:

• if e1 evaluates to \texttt{true}, and if e2 evaluates to \texttt{v},
  then if e1 then e2 else e3 evaluates to \texttt{v}
• if e1 evaluates to \texttt{false}, and if e3 evaluates to \texttt{v},
  then if e1 then e2 else e3 evaluates to \texttt{v}

Type checking:

if e1 has type \texttt{bool} and e2 has type \texttt{t} and e3 has type \texttt{t}
then if e1 then e2 else e3 has type \texttt{t}
Types

Write *colon* to indicate type of expression

As does the top-level:

```
# let x = 22;;
val x : int = 22
```

Pronounce colon as "has type"
**if expressions**

Syntax:

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3
\]

Evaluation:

- if \( e_1 \) evaluates to \text{true}, and if \( e_2 \) evaluates to \( v \), then \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) evaluates to \( v \)
- if \( e_1 \) evaluates to \text{false}, and if \( e_3 \) evaluates to \( v \), then \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) evaluates to \( v \)

Type checking:

\[
\text{if } e_1 : \text{bool} \text{ and } e_2 : t \text{ and } e_3 : t
\]

then \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) : 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
Question

To what value does this expression evaluate?

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

A. 0  
B. 1  
C. 2  
D. none of the above  
E. I don’t know
To what value does this expression evaluate?

```plaintext
if x=0 then 1 else 2
```

A. 0  
B. 1  
C. 2  
D. none of the above  
E. I don't know
A note on equality

• OCaml has two equality operators, = and ==
  – Single equals: *structural equality*
    • are two values the same?
    • its negation <> is *structural inequality*
  – Double equals: *physical equality*
    • are two values not just the same, but at the same location in memory?
    • its negation != is *physical inequality*

• **Get in the habit now of using =**
  Some OCaml extensions actually disable == so you can't use it by accident
Question

To what value does this expression evaluate?

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

A. 0
B. 1
C. 2
D. none of the above
E. I don't know
Question

To what value does this expression evaluate?

```
if 22=0 then 1 else 2
```

A. 0  
B. 1  
C. 2  
D. none of the above  
E. I don't know
Question

To what value does this expression evaluate?

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

A. 0
B. 1
C. 2
D. none of the above
E. I don't know
Question

To what value does this expression evaluate?

```
if 22=0 then "bear" 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; note how this is (overly) conservative  
E. I don't know
FUNCTIONS
Function definition

Functions:
• Like Java methods, have arguments and result
• Unlike Java, no classes, \texttt{this}, \texttt{return}, etc.

Example \textit{function definition}:

\begin{verbatim}
(* requires: y>=0 *)
(* returns: x to the power of y *)
let rec pow x y =
  if y=0 then 1
  else x * pow x (y-1)
\end{verbatim}

Note: \texttt{rec} is required because the body includes a recursive function call
Function definition

Syntax:

\[
\text{let rec } f \ x_1 \ x_2 \ \ldots \ x_n = e
\]

note: \textit{rec} can be omitted if function is not recursive

Evaluation:

Not an expression! Just defining the function; will be evaluated later, when applied
Function types

Type $t \rightarrow u$ is the type of a function that takes input of type $t$ and returns output of type $u$.

Type $t_1 \rightarrow t_2 \rightarrow u$ is the type of a function that takes input of type $t_1$ and another input of type $t_2$ and returns output of type $u$.

etc.
**Function definition**

**Syntax:**

\[
\text{let rec } f \ x_1 \ x_2 \ldots \ x_n = e
\]

**Type-checking:**

Conclude that \( f : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u \) if \( e : u \) under these 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)
Writing argument types

Though types can be inferred, you can write them too. Parens are then mandatory.

```plaintext
let rec pow (x : int) (y : int) : int = 
    if y=0 then 1 
    else x * pow x (y-1)

let rec pow x y = 
    if y=0 then 1 
    else x * pow x (y-1)

let cube x = pow x 3 
let cube (x : int) : int = pow x 3 
```
Function application v1

Syntax: f e1 ... en

• Parentheses not required around argument(s)
• Possible for syntax to look like C function call:
  – f(e1)
  – if there is exactly one argument
  – and if you do use parentheses
  – and if you leave out the white space
Function application v1

Type-checking

\[
\text{if } f : t_1 \to \ldots \to t_n \to u \\
\text{and } e_1 : t_1, \ldots, e_n : t_n \\
\text{then } f \, e_1 \ldots \, e_n : u
\]

e.g.

\text{pow} \ 2 \ 3 : \text{int}

because \text{pow} : \text{int} \to \text{int} \to \text{int}

and \ 2 : \text{int} \ \text{and} \ 3 : \text{int}
Function application v1

Evaluation of $f \ e_1 \ldots \ e_n$:

1. Find the definition of $f$
   
   ```
   let f x_1 \ldots x_n = e
   ```

2. Evaluate arguments $e_1\ldots e_n$ to values $v_1\ldots v_n$

3. Substitute $v_i$ for $x_i$ in $e$ yielding new expression $e'$

4. Evaluate $e'$ to a value $v$, which is result
Example

let area_rect w h = w *. h
let foo = area_rect (1.0 *. 2.0) 11.0

To evaluate function application:
1. Find the definition of area_rect
   let area_rect w h = w *. h
2. Evaluate arguments (1.0 *. 2.0) and 11.0 to values 2.0 and 11.0
3. Substitute in w *. h yielding new expression 2.0 *. 11.0
4. Evaluate 2.0 *. 11.0 to a value 22.0, which is result
Exercise

let area_rt_tri a b = a *. b /. 2.0
let bar = area_rt_tri 3.0 (10.0 ** 2.0)

To evaluate function application: (you try it)

1. Find the definition of area_rt_tri
   ```plaintext
   let area_rt_tri a b = a *. b /. 2.0
   ```
2. Evaluate arguments 3.0 and (10.0 ** 2.0) to values 3.0 and 100.0
3. Substitute in a *. b /. 2.0 yielding new expression 3.0 *. 100.0 /. 2.0
4. Evaluate 3.0 *. 100.0 /. 2.0 to a value 150.0, which is result
Anonymous functions

Something that is *anonymous* has no name

- **42** is an anonymous `int`
- and we can bind it to a name:
  
  ```
  let x = 42
  ```

- **fun x -> x+1** is an anonymous function
- and we can bind it to a name:
  
  ```
  let inc = fun x -> x+1
  ```

note: dual purpose for `->` syntax: function types, function values

note: **fun** is a keyword 😊
Anonymous functions

Syntax: fun x₁ ... xₙ -> e

Evaluation:
• Is an expression, so can be evaluated
• A function is a value: no further computation to do
• In particular, body e is not evaluated until function is applied

Type checking:
(fun x₁ ... xₙ -> e) : t₁->...->tₙ->t
if e:t under assumptions x₁:t₁, ..., xₙ:tₙ
Anonymous functions

These definitions are **syntactically different** but **semantically equivalent**:

```plaintext
let inc = fun x -> x+1
let inc x = x+1
```
Lambda

• Anonymous functions a.k.a. *lambda expressions*
• Math notation: $\lambda x . e$
• The lambda means “what follows is an anonymous function”
  – $x$ is its argument
  – $e$ is its body
  – Just like `fun x -> e`, but different "syntax"

• You’ll see “lambda” show up in many places in PL, e.g.:
  – Java 8: [https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html](https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html)
  – Lambda style: [https://www.youtube.com/watch?v=CI48kqp11F8](https://www.youtube.com/watch?v=CI48kqp11F8)
Function application v2

Syntax: $e_0 \ e_1 \ \ldots \ e_n$

- Function to be applied can be an expression
  - Maybe just a defined function's name
  - Or maybe an anonymous function
  - Or maybe something even more complicated
- Example:
  - $(\text{fun } x \rightarrow x + 1) \ 2$
Function application v2

Type-checking (not much of a change)

\[
\text{if } e_0 : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow u \\
\text{and } e_1 : t_1, \ldots, e_n : t_n \\
\text{then } e_0 \ e_1 \ \ldots \ e_n : u
\]
Function application v2

Evaluation of e0 e1 ... en:

1. Evaluate e0 to a function
   fun x1 ... xn -> e

2. Evaluate arguments e1...en to values v1...vn

3. Substitute vi for xi in e yielding new expression e’

4. Evaluate e’ to a value v, which is result
Function application v2

Evaluation of $e_0\ e_1\ \ldots\ e_n$:

1. Evaluate $e_0$ to a function
   $$\text{fun} \ x_1 \ \ldots\ x_n \rightarrow e$$

Examples:
- $e_0$ could be an anonymous function expression
  $$\text{fun} \ x \rightarrow x+1$$
  in which case evaluation is immediately done
- $e_0$ could be the name of a defined function
  $$\text{inc}$$
  in which case look up the definition
  $$\text{let} \ \text{inc} \ x = x + 1$$
  and we now know that's equivalent to
  $$\text{let} \ \text{inc} = \text{fun} \ x \rightarrow x+1$$
  so evaluates to
  $$\text{fun} \ x \rightarrow x+1$$
Function application operator

• Infix operator for function application
• Instead of \( f \ e \) can write \( e \ |> f \)
• Run a value through several functions
  \( 5 \ |> \text{inc} \ |> \text{square} \quad (* \ 36 \ *) \)
• "pipeline" operator
Functions are values

• Can use them anywhere we use values
• Functions can take functions as arguments
• Functions can return functions as results
  ...so functions are higher-order
• This is not a new language feature; just a consequence of "a functions is a value"
• But it is a feature with massive consequences!