FUNCTIONAL PROGRAMMING

Variables and Functions

Prof. Clarkson
Summer 2015

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

Yesterday:
• What is a functional language?
• Why learn to program in a functional language?
• Lab: Simple OCaml expressions

Today:
• **Variables**: bindings, scope
• **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 well-known project) provide “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: produce a type or fail with an error message
  - Evaluation rules: 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
Let expressions

Syntax:

```
let x = e1 in e2
```

- `x` is an identifier
- `e1` and `e2` are expressions
- `let x = e1 in e2` is itself an expression
- `x = e1` is a binding

e.g.

```
let x = 2 in x+x
let inc x = x+1 in inc 10
let y = "zar" in (let z = "doz" in y^z)
```
Let expressions

\begin{equation}
\text{let } x = e_1 \text{ in } e_2
\end{equation}

Evaluation:

– Evaluate \( e_1 \) to a value \( v_1 \)
– Substitute \( v_1 \) for \( x \) in \( e_2 \), yielding a new expression \( e_2' \)
– Evaluate \( e_2' \) to \( v \)
– Result of evaluation is \( v \)
Let expressions

\[
\text{let } x = 1+4 \text{ in } x*3
\]

\[\rightarrow\] Evaluate \(e_1\) to a value \(v_1\)

\[
\text{let } x = 5 \text{ in } x*3
\]

\[\rightarrow\] Substitute \(v_1\) for \(x\) in \(e_2\), yielding a new expression \(e_2'\)

\[
5*3
\]

\[\rightarrow\] Evaluate \(e_2'\) to \(v\)

\[
15
\]

Result of evaluation is \(v\)
Let expressions in REPL

Syntax:

```
let x = e
```

Implicitly, “in rest of what you type”

E.g., you type:

```
let a="zar";;
let b="doz";;
let c=a^b;;
```

OCaml understands as

```
let a="zar" in
let b="doz" in
let c=a^b in...
```
Scope

Bindings are in effect only in the scope (the “block”) in which they occur.

```plaintext
let x=42 in
  (* y is not in scope here *)
x + (let y="3110" in
    (* y is in scope here *)
    int_of_string y)
```

Exactly what you’re used to from (e.g.) Java
Overlapping scope

Overlapping bindings of the same name is usually bad idiom (and darn confusing)

```
let x = 5 in ((let x = 6 in x) + x)
```

To what value does the above expression evaluate?
• 10
• 11
• 12
• None of the above
Substitution

```plaintext
let x = 5 in ((let x = 6 in x) + x)
```

-->

```plaintext
???
```

Not a choice:

```plaintext
let x = 5 in (6 + 6)
```

Two choices:

A. ```plaintext>((let x = 6 in x) + 5)```  
B. ```plaintext>((let x = 6 in 5) + 5)```
Substitution

```plaintext
let x = 5 in ((let x = 6 in x) + x)
```

-->

```plaintext
???
```

Not a choice:

```plaintext
let x = 5 in (6 + 6)
```

Two choices:

A. ```plaintext>((let x = 6 in x) + 5)```  
B. ```plaintext>((let x = 6 in 5) + 5)```  Why?
Principle of Name Irrelevance

The name of a variable should not matter.

In math, these are the same functions:
\[ f(x) = x^2 \]
\[ f(y) = y^2 \]

So in programming, these should be the same functions:
```plaintext
let f x = x*x
let f y = y*y
```

This principle is also called \textit{alpha equivalence}
Principle of Name Irrelevance

Likewise, these should be the same expressions:

\[
\text{(let } x = 6 \text{ in } x) \\
\text{(let } y = 6 \text{ in } y)
\]

So these should also be the same:

\[
\text{let } x = 5 \text{ in } ((\text{let } x = 6 \text{ in } x) + x) \\
\text{let } x = 5 \text{ in } ((\text{let } y = 6 \text{ in } y) + x)
\]

But if we substitute inside inner \texttt{let} expression, they will not be the same:

\[
\text{(let } x = 6 \text{ in } 5) + 5 \quad -----> \quad 10 \\
\text{(let } y = 6 \text{ in } y) + 5 \quad -----> \quad 11
\]
Back to substitution

```
let x = 5 in ((let x = 6 in x) + x)
```  

-->

````
???
```  

Not a choice:

```
let x = 5 in (6 + 6)
```  

Two choices:

A. `((let x = 6 in x) + 5)`

B. `((let x = 6 in 5) + 5)`

That's why!
Shadowing

A new binding *shadows* an older binding of the same name

```plaintext
let x = 5 in (let x = 6 in x) + x
```
Shadowing is not assignment

\[
\text{let } x = 5 \text{ in } ((\text{let } x = 6 \text{ in } x) + x)
\]
\[\rightarrow 11\]

\[
\text{let } x = 5 \text{ in } (x + (\text{let } x = 6 \text{ in } x))
\]
\[\rightarrow 11\]
Types

Write `colon` to indicate type of expression

As does the top-level:

```ocaml
# let x = 42;;
val x : int = 42
```

Type-checking of let expression:

If \( e_1 : t_1 \), and if \( e_2 : t_2 \) (assuming that \( x : t_1 \)), then \( (let \ x = e_1 \ in \ e_2) : t_2 \)
Let expressions (summary)

• **Syntax:**
  
  ```
  let x = e1 in e2
  ```

• **Type-checking:**
  If \( e1 : t1 \), and if \( e2 : t2 \) under the assumption that \( x : t1 \), then `let x = e1 in e2 : t2`

• **Evaluation:**
  
  – Evaluate \( e1 \) to \( v1 \)
  
  – Substitute \( v1 \) for \( x \) in \( e2 \) yielding new expression \( e2' \)
  
  – Evaluate \( e2' \) to \( v \)
  
  – Result of evaluation is \( v \)
Function declaration

Functions:
• Like Java methods, have arguments and result
• Unlike Java, no classes, **this**, **return**, etc.

Example function declaration:

\[
(* \text{ requires: } y \geq 0 *)
\]
\[
(* \text{ returns: } x \text{ to the power of } y *)
\]
\[
\text{let rec pow x y =}
\]
\[
\quad \text{if } y=0 \text{ then 1}
\]
\[
\quad \text{else } x \times \text{pow } x (y-1)
\]

Note: “**rec**” is required because the body includes a recursive function call: \(\text{pow}(x, y-1)\)
Function declaration

• **Syntax:**
  ```
  let f x1 x2 ... xn = e
  ```

• **Evaluation:**
  – No evaluation!
  – Just declaring the function
  – Will be evaluated when applied to arguments

• **Type-checking:**
  – Conclude that \( f : t_1 \to \ldots \to t_n \to t \) if \( e : t \) under assumptions:
    • \( x_1 : t_1, \ldots, x_n : t_n \) (arguments with their types)
    • \( f : t_1 \to \ldots \to t_n \to t \) (for recursion)
Writing argument types

Though types can be inferred, you can write them too:

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

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

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

Syntax: `e0 e1 ... en`

- Parentheses not strictly required around argument(s)
- If there is exactly one argument and you do use parentheses and you leave out the space, syntax looks like C function call: `e0(e1)`
Function application

Type-checking

if \( e_0 : t_1 \rightarrow \ldots \rightarrow t_n \rightarrow t \)
and \( e_1 : t_1, \ldots, e_n : t_n \)
then \( e_0 \ e_1 \ldots \ e_n : t \)

e.g., \( \text{pow} \ 2 \ 3 : \text{int} \)
Function application

Evaluation of \( e_0 \ e_1 \ldots \ e_n \)

1. Evaluate \( e_0 \) to a function
   
   \[
   \texttt{let } f \ x_1 \ldots x_n = e
   \]

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
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
  ```
Anonymous functions

Syntax: \( (\text{fun } x_1 \ldots x_n \rightarrow e) \)

Evaluation:
- A function is already a value: no further computation to do
- In particular, body \( e \) is not evaluated until function is applied

Type checking:
\((\text{fun } x_1 \ldots x_n \rightarrow e) : t_1\rightarrow\ldots\rightarrow t_n\rightarrow t\)
if \( e : t \) under assumptions \( x_1 : t_1, \ldots, x_n : t_n \)
Anonymous functions

These two declarations are syntactically different but semantically equivalent:

```
let inc = fun x -> x+1
let inc x = x+1
```
Anonymous functions

These two expressions are **syntactically different** but **semantically equivalent**:

\[
\text{let } x = 7 \text{ in } x + 1 \\
(f \text{un } x \rightarrow x + 1) \ 7
\]
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 "functions are values"
• But it is a feature with massive consequences

"A language that doesn't affect the way you think about programming is not worth knowing." --Alan Perlis
Alan Jay Perlis

First Winner of Turing Award (1966)

for his influence in the area of advanced programming techniques and compiler construction

1922-1990

Google "perlisisms" for great quotes about programming
Higher-order functions

(* some base function *)

let double x = 2*x

let square x = x*x

(* apply those functions twice *)

let quad x = double (double x)

let fourth x = square (square x)
Higher-order functions

(* higher order function that
 * applies f twice to x *)

let twice f x = f (f x)

val twice : ('a -> 'a) -> 'a -> 'a

'a is a type variable: could be any type
Higher-order functions

(* higher-order function that
  * applies f twice to x *)
let twice f x = f (f x)

(* define functions using twice *)
let quad x = twice double x
let fourth x = twice square x

(* even better definitions *)
let quad = twice double
let fourth = twice square