Functional Programming

- Programs are viewed as functions transforming input to output
- Complex transformations are achieved by composing simpler functions (i.e. applying functions to results of other functions)
- **Purely Functional Languages**: Values given to “variables” do not change when a program is evaluated
  - “Variables” are names for values, not names for storage locations.
  - Functions have *referential transparency*:
    - Value of a function depends solely on the values of its arguments
    - Functions do not have *side effects.*
    - Order of evaluation of arguments does not affect the value of a function’s output.
Features of Functional Programming Languages

- Usually support complex (recursive) data types
  ... with automatic allocation and deallocation of memory (e.g. garbage collection)
- No loops: recursion is the only way to structure repeated computations
- Functions themselves may be treated as values
  - Higher-order functions: Functions that functions as arguments.
  - Functions as first-class values: no arbitrary restrictions that distinguish functions from other data types (e.g. int)

History of Functional Programming

- LISP ('60)
- Scheme ('80s): a dialect of LISP; more uniform treatment of functions
- ML ('80s): Strong typing and type inference
  - Standard ML (SML, SML/NJ: '90s)
  - Categorical Abstract Machine Language (CAML, CAML Light, O'CAML: late '90s)
- Haskell, Gofer, HUGS, ... (late '90s): “Lazy” functional programming
Developed initially as a “meta language” for a theorem proving system (*Logic of Computable Functions*)

- The two main dialects, SML and CAML, have many features in common:
  - data type definition, type inference, interactive top-level, . . .
- SML and CAML have different syntax for expressing the same things. For example:
  - In SML: variables are defined using `val` and functions using `fun`
  - In CAML: both variables and functions defined using `equations`
- Both have multiple implementations (Moscow SML, SML/NJ; CAML, OCAML) with slightly different usage directives and module systems.

---

**OCAML**

- CAML with “object-oriented” features.
- Compiler and run-time system that makes OCAML programs run with performance comparable imperative programs!
- A complete development environment including libraries building UIs, networking (sockets), etc.
- *We will focus on the non-oo part of OCAML*
  - Standard ML (SML) has more familiar syntax.
  - CAML has better library and runtime support and has been used in more “real” systems.
The OCAML System

- OCAML interactive toplevel
  - Invocation:
    - UNIX: Run `ocaml` from command line
    - Windows: Run `ocaml.exe` from Command window or launch `ocamlwin.exe` from windows explorer.
  - OCAML prompts with “#”
  - User can enter new function/value definitions, evaluate expressions, or issue OCAML directives at the prompt.
  - Control-D to exit OCAML

- OCAML compiler:
  - `ocamlc` to compile OCAML programs to object bytecode.
  - `ocamlopt` to compile OCAML programs to native code.

Learning OCAML

- We will use OCAML interactive toplevel throughout for examples.
- What we type in can be entered into a file (i.e. made into a “program”) and executed.
- Read David Matuszek’s tutorial for a quick intro, then go to Jason Hickey’s tutorial. To clarify syntax etc. see OCAML manual. (http://caml.inria.fr/tutorials-eng.html)
Expression Evaluation

- Syntax: \textit{⟨expression⟩ ;;}
- Two semicolons indicate the end of expression

**Example:**

<table>
<thead>
<tr>
<th>User Input</th>
<th>OCAML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 * 3 ;;</td>
<td>- : int = 6</td>
</tr>
</tbody>
</table>

OCAML’s response:
- ‘-’ : The last value entered
- ‘.’ : is of type
- ‘int’ : integer
- ‘=’ : and the value is
- ‘6’ : 6

**Expressions (contd.)**

- More examples:

<table>
<thead>
<tr>
<th>User Input</th>
<th>OCAML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 + 3 * 4 ;;</td>
<td>- : int = 14</td>
</tr>
<tr>
<td>-2 + 3 * 4 ;;</td>
<td>- : int = 10</td>
</tr>
<tr>
<td>(-2 + 3) * 4 ;;</td>
<td>- : int = 4</td>
</tr>
<tr>
<td>4.4 ** 2.0 ;;</td>
<td>- : float = 19.36</td>
</tr>
<tr>
<td>2 + 2.2 ;;</td>
<td>... This expression has type float but is used here with type int</td>
</tr>
<tr>
<td>2.7 + 2.2 ;;</td>
<td>... This expression has type float but is used here with type int</td>
</tr>
<tr>
<td>2.7 +. 2.2 ;;</td>
<td>- : float = 4.9</td>
</tr>
</tbody>
</table>
Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Integer arithmetic</td>
</tr>
<tr>
<td>-</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td></td>
</tr>
<tr>
<td>/</td>
<td></td>
</tr>
<tr>
<td>mod</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>Floating point arithmetic</td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>**</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;,</td>
<td></td>
</tr>
</tbody>
</table>

Value definitions

- Syntax: `let ⟨name⟩ = ⟨expression⟩ ;;`
- Examples:

<table>
<thead>
<tr>
<th>User Input</th>
<th>OCAML's Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>let x = 1;;</code></td>
<td><code>val x : int = 1</code></td>
</tr>
<tr>
<td><code>let y = x + 1;;</code></td>
<td><code>val y : int = 2</code></td>
</tr>
<tr>
<td><code>let x = x + 1;;</code></td>
<td><code>val x : int = 3</code></td>
</tr>
<tr>
<td><code>let z = &quot;OCAML rocks!&quot;;;</code></td>
<td><code>val z : string = &quot;OCAML rocks!&quot;</code></td>
</tr>
<tr>
<td><code>let w = &quot;21&quot;;;</code></td>
<td><code>val w : string = &quot;21&quot;</code></td>
</tr>
<tr>
<td><code>let v = int_of_string(w) ;;</code></td>
<td><code>val v : int = 21</code></td>
</tr>
</tbody>
</table>
Functions

- Syntax: `let ⟨name⟩ {⟨argument⟩} = ⟨expression⟩ ;;`
- Examples:

<table>
<thead>
<tr>
<th>User Input</th>
<th>OCAML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>let f x = 1;;</td>
<td>val f : 'a -&gt; int = &lt;fun&gt;</td>
</tr>
<tr>
<td>let g x = x;;</td>
<td>val g : 'a -&gt; 'a = &lt;fun&gt;</td>
</tr>
<tr>
<td>let inc x = x + 1;;</td>
<td>val inc : int -&gt; int = &lt;fun&gt;</td>
</tr>
<tr>
<td>let sum(x,y) = x+y;;</td>
<td>val sum : int * int -&gt; int = &lt;fun&gt;</td>
</tr>
<tr>
<td>let add x y = x+y;;</td>
<td>val add : int -&gt; int -&gt; int = &lt;fun&gt;</td>
</tr>
</tbody>
</table>

Note the use of *parametric polymorphism* in functions `f` and `g`

Expressions (contd.)

More examples:

<table>
<thead>
<tr>
<th>User Input</th>
<th>OCAML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>let max(x, y) =</td>
<td>val max : 'a * 'a -&gt; 'a = &lt;fun&gt;</td>
</tr>
<tr>
<td>if x &lt; y</td>
<td></td>
</tr>
<tr>
<td>then y</td>
<td></td>
</tr>
<tr>
<td>else x;;</td>
<td></td>
</tr>
<tr>
<td>let mul(x, y) =</td>
<td>Unbound value mul</td>
</tr>
<tr>
<td>if x = 0</td>
<td></td>
</tr>
<tr>
<td>then 0</td>
<td></td>
</tr>
<tr>
<td>else y+mul(x-1,y);;</td>
<td></td>
</tr>
<tr>
<td>let rec mul(x, y) =</td>
<td>val mul : int * int -&gt; int = &lt;fun&gt;</td>
</tr>
<tr>
<td>if x = 0</td>
<td></td>
</tr>
<tr>
<td>then 0</td>
<td></td>
</tr>
<tr>
<td>else y+mul(x-1,y);;</td>
<td></td>
</tr>
<tr>
<td>let rec mul(x, y) =</td>
<td>val mul : int * int -&gt; int = &lt;fun&gt;</td>
</tr>
<tr>
<td>if x = 0</td>
<td></td>
</tr>
<tr>
<td>then 0</td>
<td></td>
</tr>
<tr>
<td>else let i = mul(x-1,y) in y+i;;</td>
<td>}</td>
</tr>
</tbody>
</table>
An Aside on Polymorphism

- Java supports two kinds of polymorphism:
  - *Subtype* polymorphism: a method defined for objects of class A can be applied to objects of A’s subclasses.
  - *Ad-hoc* polymorphism: a method name can be overloaded, with same name representing many different methods.

- *Templates* in C++ support an additional kind of polymorphism:
  ```cpp
  template <typename T>
  int f(T x) { return 1; }
  
  template <typename T>
  T g(T x) { return x; }
  ```

Data structures

Examples of built-in data structures (lists and tuples):

<table>
<thead>
<tr>
<th>User Input</th>
<th>OCAML’s Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1];;</td>
<td>- : int list = [1]</td>
</tr>
<tr>
<td>[4.1; 2.7; 3.1];;</td>
<td>- : float list = [4.1; 2.7; 3.1]</td>
</tr>
<tr>
<td>[4.1; 2];;;</td>
<td>... This expression has type int but is</td>
</tr>
<tr>
<td></td>
<td>used here with type float</td>
</tr>
<tr>
<td>[[1;2]; [4;8;16]];;</td>
<td>- : int list list = [[1;2], [4;8;16]]</td>
</tr>
<tr>
<td>1::2::[]</td>
<td>- : int list = [1; 2]</td>
</tr>
<tr>
<td>1::(2::[])</td>
<td>- : int list = [1; 2]</td>
</tr>
<tr>
<td>(1,2);;</td>
<td>- : int * int = (1, 2)</td>
</tr>
<tr>
<td>();;</td>
<td>- : unit = ()</td>
</tr>
<tr>
<td>let (x,y) = (3,7);;</td>
<td>val x : int = 3</td>
</tr>
<tr>
<td></td>
<td>val y : int = 7</td>
</tr>
</tbody>
</table>
Pattern Matching

- Used to “deconstruct” data structures.
- Example:

```ml
let rec sumlist l =
    match l with
    | [] -> 0
    | x::xs -> x + sumlist(xs);
```

- When evaluating `sumlist [2; 5]`
  - The argument `[2; 5]` matches the pattern `x::xs`,
  - ... setting `x` to `2` and `xs` to `[5]`
  - ... then evaluates `2 + sumlist([5])`

Match statements

- `match` is analogous to a “switch” statement
  - Each case describes
    - a pattern (lhs of ‘->’) and
    - an expression to be evaluated if that pattern is matched (rhs of ‘->’)
    - patterns can be constants, or terms made up of constants and variables
  - The different cases are separated by ‘|’
  - A matching pattern is found by searching in order (first case to last case)
  - The first matching case is selected; others are discarded

```ml
let emptyList l =
    match l with
    | [] -> true
    | _ -> false;;
```
Patterns

- Pattern syntax:
  - Patterns may contain “wildcards” (i.e. ‘_’); each occurrence of a wildcard is treated as a new anonymous variable.
  - Patterns are linear: any variable in a pattern can occur at most once.
- Pattern matching is used very often in OCAML programs.
- OCAML gives a shortcut for defining pattern matching in functions with one argument.

Example:

```ocaml
let rec sumlist l = 
  match l with 
  | [] -> 0 
  | x::xs -> x + 
  sumlist(xs);;
```

Functions on Lists

- Add one list to the end of another:

```ocaml
let rec append v1 v2 = 
  match v1 with 
  | [] -> v2 
  | x::xs -> x::(append xs v2);;
```

- Note that this function has type
  append: 'a list -> 'a list -> 'a list
  and hence can be used to concatenate arbitrary lists, as long as the list elements are of the same type.
- This function is implemented by builtin operator @

- Many list-processing functions are available in module Lists.
  Examples:
  - Lists.hd: get the first element of the given list
  - Lists.rev: reverse the given list
## Enumerated Types

- A finite set of values
- Two values can be compared for equality
- There is no order among values
- Example:
  ```haskell
  type primaryColor = RED | GREEN | BLUE;;
  type status = Freshman | Sophomore | Junior | Senior;;
  ```
- Syntax: `type ⟨name⟩ = ⟨name⟩ { | ⟨name⟩ } ;;`
- A note about names:
  - Names of constants must begin with an *uppercase* letter.
  - Names of types, functions and variables must begin with a *lowercase* letter.
  - Names of constants are global within a module and not local to its type.

## Records

- Used to define structures with named fields.
- Example:
  ```haskell
  type student = {name:string; gpa:float; year:status;};
  ```
- Syntax: `type ⟨name⟩ = { ⟨name⟩ { : ⟨name⟩ } } ;;`
- Usage:
  - Creating records:
    ```haskell
    let joe = {name="Joe"; gpa=2.67; year=Sophomore;};
    ```
  - Accessing fields:
    ```haskell
    let x = joe.gpa;; (* using "." operator *)
    let {id=x} = joe;; (* using pattern matching *)
    ```
- Field names are global within a module and not local to its type.
Union Types

- Used to define (possibly recursive) structured data with tags.
- Example:
  ```haskell
type iTree = Node of int * iTree * iTree | Empty;;
```
- The empty tree is denoted by `Empty`.
- The tree with one node, with integer 2, is denoted by `Node(2,Empty,Empty)`.

<table>
<thead>
<tr>
<th>Tree</th>
<th>Denoted by</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Node(1, Node(2, Node(4, Empty, Empty), Node(5, Empty, Empty)), Node(3, Empty, Node(6, Empty, Empty))))</td>
</tr>
<tr>
<td>2</td>
<td>Node(2,</td>
</tr>
<tr>
<td>3</td>
<td>Node(4, Empty, Empty), Node(5, Empty, Empty))</td>
</tr>
<tr>
<td>4</td>
<td>Node(3,</td>
</tr>
<tr>
<td>5</td>
<td>Empty,</td>
</tr>
<tr>
<td>6</td>
<td>Node(6, Empty, Empty))</td>
</tr>
</tbody>
</table>

- Generalizes enumerated types.
- Constants that tag the different structures in an union (e.g. `Node` and `Empty`) are called `data constructors`.
- Usage example: counting the number of elements in a tree:
  ```haskell```
  ```haskell
  let rec nelems tree =
  match tree with
  Node(i, lst, rst) ->
  (* 'i' is the value of the node; 'lst' is the left sub tree; and 'rst' is the right sub tree *)
  1 + nelems lst + nelems rst
  | Empty -> 0;;
  ```
Polymorphic Data Structures

- Structures whose components may be of arbitrary types.
- Example:
  
  ```
  type 'a tree = Node of 'a * 'a tree * 'a tree | Empty;;
  ```

  - `'a` in the above example is a *type variable* ... analogous to the *typename* parameters of a C++ template

- Parameteric polymorphism enforces that all elements of the tree are of the same type.
- Usage example: traversing a tree in preorder:
  
  ```
  let rec preorder tree =
    match tree with
      Node(i, lst, rst) -> i::(preorder lst)@(preorder rst)
    | Empty -> [];;
  ```

Exceptions

- **Total function**: function is defined for every argument value.
  - Examples: +, length, etc.
- **Partial function**: function is defined only for a subset of argument values.
  - Examples: /, Lists.hd, etc.
  - Another example:
    
    ```
    (* find the last element in a list *)
    let rec last = function
      x::[]    -> x
    | _::xs    -> last xs;;
    ```

- Exceptions can be used to signal invalid arguments.
- Failed pattern matching (due to incomplete matches) is signalled with (predefined) Match_failure exception.
- Exceptions also signal unexpected conditions (e.g. I/O errors)
Exceptions (contd.)

- Users can define their own exceptions.
- Exceptions can be thrown using `raise`.

```plaintext
(* Exception to signal no elements in a list *)
exception NoElements;;

let rec last = function
  | [] -> raise NoElements
  | x::[] -> x
  | _::xs -> last xs;;
```

- Exceptions can be handled using `try ... with`.

```plaintext
exception DumbCall;;
let test l y =
  try (last l) / y
  with
  | NoElements -> 0
  | Division_by_zero -> raise DumbCall;;
```

Functions of functions

- Add 1 to every element in list:

  ```plaintext
  let rec add_one = function
    | [] -> []
    | x::xs -> (x+1)::(add_one xs);
  ```

- Multiply every element in list by 2:

  ```plaintext
  let rec double = function
    | [] -> []
    | x::xs -> (x*2)::(double xs);
  ```

- Perform function `f` on every element in list:

  ```plaintext
  let rec map f = function
    | [] -> []
    | x::xs -> (f x)::(map f xs);
  ```

- Now we can write `add_one` and `double` as:

  ```plaintext
  let add_one = map ((+) 1);; let double = map (( *) 2);;
  ```
More examples of higher-order functions

<table>
<thead>
<tr>
<th>Sum all elements in a list</th>
<th>Multiply all elements in a list</th>
</tr>
</thead>
<tbody>
<tr>
<td>let rec sumlist = function</td>
<td></td>
</tr>
<tr>
<td>[]   -&gt; 0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>x::xs -&gt; x + sumlist xs;</td>
</tr>
<tr>
<td></td>
<td>[]   -&gt; 1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Accumulate over a list:

let rec foldr f b = function
(* f is the function to apply at element;  
   b is the base case value *)
   []   -> b
   | x::xs -> f x (foldr f b xs);;

Using foldr:

<table>
<thead>
<tr>
<th>Sum all elements in a list</th>
<th>Multiply all elements in a list</th>
</tr>
</thead>
<tbody>
<tr>
<td>let sumlist = foldr (+) 0;</td>
<td>let prodlist = foldr ( * ) 1;</td>
</tr>
</tbody>
</table>

Summary

- OCAML definitions have the following syntax:

  $\langle def \rangle ::= \text{let [rec] } \langle letlhs \rangle = \langle expr \rangle$
  
  (value definitions)
  
  | $\text{type } \langle typelhs \rangle = \langle typeexpr \rangle$
  
  (type definitions)
  
  | exception definitions ...

  $\langle letlhs \rangle ::= \langle id \rangle \{\langle pattern \rangle\}$
  
  (patterns specify “parameters”)

  $\langle typelhs \rangle ::= \{\langle typevar \rangle\}\langle id \rangle$
  
  (typevars specify “parameters”)

- OCAML programs are a sequence of definitions separated by \text{;;}
OCAML expressions have the following syntax:

\[
\langle \text{expr} \rangle ::= \langle \text{const} \rangle \\
\text{(constants)} \\
\mid \langle \text{id} \rangle \\
\text{(value identifiers)} \\
\mid \langle \text{expr} \rangle \langle \text{op} \rangle \langle \text{expr} \rangle \\
\text{(expressions with binary operators)} \\
\mid \langle \text{expr} \rangle \langle \text{expr} \rangle \\
\text{(function application)} \\
\mid \text{let } [\text{rec}] \{ \langle \text{letlhs} \rangle = \langle \text{expr} \rangle ; ; \} \text{in } \langle \text{expr} \rangle \\
\text{(let definitions)} \\
\mid \text{raise } \langle \text{expr} \rangle \\
\text{(throw exception)}
\]

Expressions (contd.)

\[
\langle \text{case} \rangle ::= \langle \text{pattern} \rangle \rightarrow \langle \text{expr} \rangle \\
\text{(pattern matching case)}
\]
Writing Efficient OCAML Programs

- Using recursion to sum all elements in a list:

<table>
<thead>
<tr>
<th>OCAML</th>
<th>C</th>
</tr>
</thead>
</table>
| let rec sumlist = function
  | int sumlist(List l) {
  |     [] -> 0
  |     | x::xs -> x + sumlist xs;; |
  |     if (l == NULL)
  |     | return 0; |
  |     else
  |     | return (l->element) + |
  |     | sumlist(l->next); |

- Iteratively summing all elements in a list (C):

```c
int acc = 0;
for(l=list; l!=NULL; l = l->next)
```
```c
    acc += l->element;
```

Iteration vs. Recursion

- Recursive summation takes stack space proportional to the length of the list

- Iterative summation takes constant stack space.
Tail Recursion

- let rec last = function
  | [] -> raise NoElements
  | x::[] -> x
  | _::xs -> last xs;;
- Evaluation of last [1;2;3];;

\[
\begin{array}{c}
\text{last([1;2;3])} \\
\text{⇒ last([2;3])} \\
\text{⇒ last([3])} \\
\downarrow \\
3
\end{array}
\]

- let rec last = function
  | [] -> raise NoElements
  | x::[] -> x
  | _::xs -> last xs;;
- Note that when the 3rd pattern matches, the result of last is whatever is the result of last xs.
  Such calls are known as tail recursive calls.

Taking Efficiency by the Tail

An efficient recursive function for summing all elements:

**C**

```c
int acc_sumlist(int acc, List l) {
    if (l == NULL)
        return acc;
    else
        return acc_sumlist(acc + (l->element), l->next);
}
int sumlist(List l) {
    return acc_sumlist(0, l);
}
```

**OCAML**

```ocaml
let rec acc_sumlist acc =
    function
    | [] -> acc
    | x::xs -> acc_sumlist(acc+x) xs;;
let sumlist l =
    acc_sumlist 0 l;;
```

\[
\begin{array}{c}
\text{acc_sumlist(0,[1;2])} \\
\text{⇒ acc_sumlist(1,[2])} \\
\text{⇒ acc_sumlist(3,[])} \\
\downarrow \\
3
\end{array}
\]
Obligatory XKCD

http://xkcd.com/1270/