CSE 307: Principles of Programming Languages

Syntax

R. Sekar

Topics

1. Introduction
2. Basics
3. Functions
4. Data Structures
5. Overview
6. OCAML Performance

Section 1

Introduction
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.

Functional Programming (Contd.)

- 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

- 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.

---

**Section 2**

**Basics**

---

**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](http://caml.inria.fr/tutorials-eng.html)

Expression Evaluation

- Syntax: `⟨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><code>2 * 3;;</code></td>
<td><code>- : int = 6</code></td>
</tr>
</tbody>
</table>

OCAML's response:
- `'-'` : The last value entered
- `':'` : is of type
- `'int'` : integer
- `'='` : and the value is
- `'6'` : 6
Expression Evaluation (Contd.)

<table>
<thead>
<tr>
<th>User Input</th>
<th>OCaml's Response</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>2 * 3 * 4;;</code></td>
<td><code>- : int = 14</code></td>
</tr>
<tr>
<td><code>-2 + 3 * 4;;</code></td>
<td><code>- : int = 10</code></td>
</tr>
<tr>
<td><code>(2 + 3) * 4;;</code></td>
<td><code>- : int = 4</code></td>
</tr>
<tr>
<td><code>4.4 ** 2.0;;</code></td>
<td><code>- : float = 19.36</code></td>
</tr>
<tr>
<td><code>2 + 2.2;;</code></td>
<td></td>
</tr>
<tr>
<td><code>2.7 + 2.2;;</code></td>
<td></td>
</tr>
<tr>
<td><code>2.7 * 2.2;;</code></td>
<td></td>
</tr>
<tr>
<td><code>2.7 +. 2.2;;</code></td>
<td><code>- : float = 4.9</code></td>
</tr>
</tbody>
</table>

More 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 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>

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>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td></td>
</tr>
</tbody>
</table>

Value definitions

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

<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>
Section 3

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`

- More example functions

  | let max(x, y) =         | val max : 'a * 'a -> 'a = <fun> |
  | if x < y                |                                          |
  | then y                  |                                          |
  | else x;;                |                                          |
  | let mul(x, y) =         | Unbound value mul                       |
  | if x = 0                |                                          |
  | then 0                  |                                          |
  | else y+mul(x-1,y);;     |                                          |
  | let rec mul(x, y) =     | val mul : int * int -> int = <fun>      |
  | if x = 0                |                                          |
  | then 0                  |                                          |
  | else y+mul(x-1,y);;     |                                          |
  | let rec mul(x, y) =     | val mul : int * int -> int = <fun>      |
  | if x = 0                |                                          |
  | then 0                  |                                          |
  | else let i = mul(x-1,y) |                                          |
  | in y+i;                 |                                          |
Currying

- Named after H.B. Curry
- Curried functions take arguments one at a time, as opposed to taking a single tuple argument
- When provided with number of arguments less than the requisite number, result in a closure
- When additional arguments are provided to the closure, it can be evaluated

Currying Example

- Tuple version of a function
  
  ```ocaml
  fun add(x,y) = x+y:int;
  val add = fn int * int -> int
  ```

- Curried version of the same function
  
  ```ocaml
  fun addc x y = x+y:int;
  val addc = fn : int -> int -> int
  ```

- When addc is given one argument, it yields a function with type `int -> int`
  
  ```ocaml
  - add 2 3;
  - add 2;
  it = 5 : int;
  it = fn : int->int
  - it 3;
  it = 5 : int
  ```

Recursion

- Recursion is the means for iteration
- Consider the following examples
  
  ```ocaml
  fun f(0) = 0
  | f(n) = 2*f(n-1);
  ```

  ```ocaml
  fun g(0) = 1
  | g(1) = 1
  | g(n) = g(n-1)+g(n-2);
  ```

  ```ocaml
  fun h(0) = 1
  | h(n) = 2*h(n div 2);
  ```
Section 4

Data Structures

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 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>

Tuples

(2,"Andrew") : int * string
(true,3.5,"x") : bool * real * string
((4,2),(7,3)) : (int * int) * (int * int)

• Tuple components can be of different types, but lists must contain elements of same type

[1,2,3] : int list
["Andrew","Ben"] : string list
[(2,3),(2,2),(9,1)] : (int * int) list
[[],[1],[1,2]] : int list list
Pattern Matching

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

```ocaml
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])`

Pattern Matching (Contd.)

- `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

```ocaml
let emptyList l =
  match l with
  [] -> true
  _ -> false;
```

Pattern Syntax

- 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);
let rec sumlist =
  function
  [] -> 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 @

Functions on Lists (Contd.)

- 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

User-defined Types

- *Enumerated types*:
  - A finite set of values
  - Two values can be compared for equality
  - There is no order among values
  - Example:
    ```ocaml
    type primaryColor = RED | GREEN | BLUE;;
    type status = Freshman | Sophomore | Junior | Senior;;
    
    Syntax: type `<name>` = `<name>` | `<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.
Record types

- Used to define structures with named fields. Example:
  
  ```ocaml
  type student = {name:string; gpa:float; year:status};;
  ```

- Syntax: `type ⟨name⟩ = { { ⟨name⟩ : ⟨name⟩ ; } } ;;`

- Usage:
  - Creating records:
    ```ocaml
    let joe = {name="Joe"; gpa=2.67; year=Sophomore};;;
    ```
  - Accessing fields:
    ```ocaml
    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:
  
  ```ocaml
  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)`

- 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:
  ```ocaml
  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;;
  ```

Union Types (Contd.)
Recursive Types

- Direct definition of recursive types is supported in SML using datatype declarations.
  - datatype intBtree =
    LEAF of int
    | NODE of int * intBtree * intBtree;

datatype intBtree =
    LEAF of int
    | NODE of int * intBtree * intBtree

- We are defining a binary tree type inductively:
  - Base case: a binary tree with one node, called a LEAF
  - Induction case: construct a binary tree by constructing a new node that stores an integer value, and has two other binary trees as children

Recursive Types (Contd.)

- We may construct values of this type as follows:
  - val l = LEAF(1);
  - val r = LEAF(3);
  - val n = NODE(2, l, r);

Recursive Types (Contd.)

- Types can be mutually recursive. Consider:
  - datatype expr = PLUS of expr * expr |
  = PROD of expr * expr |
  = FUN of (string * exprs) |
  = IVAL of int 
  = and 
  = exprs = EMPTY
  = | LIST of expr * exprs; 
  datatype expr = FUN of string * exprs
  | PLUS of expr * expr
  | PROD of expr * expr 
  datatype exprs = EMPTY | LIST of expr * exprs

- The key word and is used for mutually recursive type definitions.
We could also have defined expressions using the predefined list type:

```ocaml
datatype expr = PLUS of expr * expr | PROD of expr * expr | FUN of string * expr list;
```

Examples: The expression $3 + (4 \times 5)$ can be represented as a value of the above datatype `expr` as follows:

```
val v3 = IVAL(3);
val v5 = IVAL(5);
val v4 = IVAL(4);
val pr = PROD(v5, v4);
val pl = PLUS(v3, pr);
```

Similarly, $f(2,4,1)$ can be represented as:

```
val a1 = EMPTY;
val a2 = ARG(IVAL(4), a1);
val a3 = ARG(IVAL(2), a2);
val fv = FUN("f", a3);
```

Note the use of `expr list` to refer to a list that consists of elements of type `expr`.

The following picture illustrates the structure of the value `pl` and how it is constructed from other values.
Polymorphic Data Structures

- Structures whose components may be of arbitrary types. Example:

  ```ocaml```
  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:

```ocaml```
let rec preorder tree =
  match tree with
    Node(i, lst, rst) -> i::(preorder lst)@(preorder rst)
  | Empty -> [];;
```n

Parameterized Types

```ocaml```

```ocaml```

```ocaml```

Example Functions and their Type

```ocaml```
- fun leftmost(LEAF(x)) = x
  =
  | leftmost(NODE(y, l, r)) = leftmost(l);
  val leftmost = fn : ('a,'b) Btree -> 'a

- fun discriminants(LEAF(x)) = nil
  =
  | discriminants(NODE(y, l, r)) =
  =
  =
  =
  =
  let
  =
  =
  =
  =
  =
```
Example Functions (Contd.)

- fun append(x::xs, y) = x::append(xs, y)
  =       | append(nil, y) = y;
val append = fn : 'a list * 'a list -> 'a list
- fun f(x::xs, y) = x::f(xs, y)
  =       | f(nil, y) = nil;
val f = fn : 'a list * 'b -> 'a list

- SML Operators that restrict polymorphism:
  - Arithmetic, relational, boolean, string, type conversion operators
- SML Operators that allow polymorphism
  - tuple, projection, list, equality (= and <>)

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: /, List.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`
  (* Exception to signal no elements in a list *)
  exception NoElements;;
  let rec last = function
     []  -> raise NoElements
     | x::[]  -> x
     | _::xs  -> last xs;;
Exceptions (Contd.)

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

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

Higher Order Functions

- Functions that take other functions as arguments, or return newly constructed functions

```ocaml
fun map f nil = nil
  | map f x::xs=(f x)::(map f xs);
```

- Map applies a function to every element of a list

```ocaml
fun filter f nil = nil
  | filter f x::xs=
    if (f x) then x::(filter f xs)
    else (filter f xs)
```

Higher Order Functions (Contd.)

```ocaml
fun zip f nil nil = nil
  | zip f (x::xs) (y::ys)=(f x,y)::(zip f xs ys);
fun reduce f b nil = b
  | reduce f b x::xs = f(x, (reduce f b xs));
```
Examples of Higher Order Functions

- Add 1 to every element in list:
  ```ocaml
  let rec add_one = function
    [] -> []
    | x::xs -> (x+1)::(add_one xs);
  ```

- Multiply every element in list by 2:
  ```ocaml
  let rec double = function
    [] -> []
    | x::xs -> (x*2)::(double xs);
  ```

More Examples

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

- Accumulate over a list:
  ```ocaml
  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);
  ```
More Examples (Contd.)

- Using \texttt{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>\texttt{let sumlist = foldr (+) 0;;}</td>
<td>\texttt{let prodlist = foldr (*) 1;;}</td>
</tr>
</tbody>
</table>

Anonymous Functions

- You can define an unnamed function

  \(-((\texttt{fn \ } x \Rightarrow \ 2*\ } x) \ 5);\)

  \texttt{val it=10 : int}

- Is handy with higher order functions

Section 5

Overview
### Summary

- **OCAML definitions** have the following syntax:

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

  \[
  \langle \text{letlhs} \rangle ::= \langle \text{id} \rangle [[(\text{pattern})]] \\
  \text{(patterns specify "parameters")}
  \]

  \[
  \langle \text{typelhs} \rangle ::= [[(\text{typevar})]]\langle \text{id} \rangle \\
  \text{(typevars specify "parameters")}
  \]

- **OCAML programs** are a sequence of definitions separated by `;;`

### Summary (Contd.)

- **OCAML expressions** have the following syntax:

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

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

- Using recursion to sum all elements in a list:

  OCAML
  ```ocaml
  let rec sumlist = function
    | [] -> 0
    | x::xs -> x + sumlist xs;
  ```

  C
  ```c
  int sumlist(List l) {
    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)
    acc += l->element;
  ```

Writing Efficient OCAML Programs (Contd.)

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

  ```
  sumlist([1;2])  \Rightarrow  sumlist([2])  \Rightarrow  2
  |
  \Downarrow
  |
  sumlist([1;2])  \Rightarrow  sumlist([1;2])  \Rightarrow  0
  |
  |
  3  \Leftarrow  sumlist([1;2])  \Leftarrow  sumlist([1;2])
  ```

- 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];

\[
\begin{align*}
\text{last([1;2;3])} & \Rightarrow \text{last([2;3])} \\
\text{last([2;3])} & \Rightarrow \text{last([1;2;3])} \\
\downarrow & \\
3 & \Rightarrow \text{3}
\end{align*}
\]

- Tail Recursion (Contd.)

- 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**.

- Tail recursive calls can be evaluated without extra stack:

\[
\begin{align*}
\text{last([1;2;3])} & \Rightarrow \text{last([2;3])} \\
\text{last([2;3])} & \Rightarrow \text{last([3])} \\
\downarrow & \\
3 & \Rightarrow \text{3}
\end{align*}
\]

Taking Efficiency by the Tail

- An efficient recursive function for summing all elements:

\[
\begin{align*}
\text{C} & \quad \text{OCAML} \\
\begin{array}{l}
\text{int acc\_sumlist(int acc, List l) \{} \\
\quad \text{if (l == NULL)} \\
\quad \quad \text{return acc;} \\
\quad \text{else} \\
\quad \quad \text{return acc\_sumlist(acc + (l->element), l->next);} \\
\text{\};} \\
\end{array} & \quad \begin{array}{l}
\text{let rec acc\_sumlist acc = function} \\
\quad [] \rightarrow acc \\
\quad | x::xs \rightarrow acc\_sumlist(acc+x) \\
\text{xs;;} \\
\text{let sumlist l = acc\_sumlist 0 l;;}
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{acc\_sumlist(0,[1;2])} & \Rightarrow \text{acc\_sumlist(1,[2])} \\
\text{acc\_sumlist(1,[2])} & \Rightarrow \text{acc\_sumlist(3,[1])} \\
\downarrow & \\
3 & \Rightarrow \text{3}
\end{align*}
\]