A Concise Introduction to Objective Caml

General

Caml is a dialect of ML, developed primarily in France. This paper describes Objective Caml version 3.01, or OCaml (pronounced "oh-camel") for short; it does not go into the object-oriented features of OCaml, however. Another dialect, Caml Lite 0.74, has almost identical syntax, but the modules and many of the functions in the modules differ to a greater or lesser extent.

OCaml is case-sensitive.

OCaml is interactive; you start it up, then you type expressions (at the prompt) to be evaluated. OCaml responds with both the value and the type of the result. Example:

```
# 3 * 4;;
- : int = 12
```

Every expression ends with with **two** semicolons.

The `#` sign is OCaml's usual input prompt. In this paper, I typically include the prompt in one-line examples, and leave it off longer examples that are best loaded from a file. Similarly, I use boldface to indicate user input that is typed in at a prompt.

To define a constant, use the form:

```
let pi = 3.1416;;
```

Most values in OCaml are *immutable* (constants). However, arrays and strings can be altered in place.

OCaml is an *expression-oriented* language, not a statement-oriented language. That is, everything in it is considered to be an expression and to have a value. A few expressions that are executed for their side effects rather than their value (mostly output expressions) return the *unit*, `()`, as their value--this is like `void` in C or Java.

To define a function, use one of the forms:

`let sum (x, y) = x + y;;`	`let add x y = x + y;;`

For reasons which will be explained later, if you use parentheses around the arguments in the definition, you must use parentheses in the function call. If you omit parentheses in the function definition, you must omit them in the function call. Hence:

```
# sum 3 5;;
- : int = 8
# add (3, 5);;
- : int = 8
```

OCaml is strongly typed; however, OCaml almost always figures out the types for itself. If you need to help it along, you can specify the type of any identifier by following it by a colon and the name of the type, enclosed in parentheses, anywhere the identifier occurs (not just in the parameter list). For example, in the function

```
let max (x, y) = if x > y then x else y;;
```

the variables `x` and `y` could be int, float, char, or string. To define this function to use only char values, give OCaml a hint by attaching `:char` to any one of the variables, anyplace in the function. For example, you could say

```
   let max (x:char) (y) = if x > y then x else y;;
or let max (x) (y:char) = if x > y then x else y;;
or let max (x) (y) = if x > y then (x:char) else y;;
```

or any of several other possibilities.

To execute the expressions in file `myFile.sml` (usually definitions that you are testing), use

```
use "myFile.sml";;
```

Predefined OCaml functions are in groups called *modules*, often organized around some particular data type. A module is like a function library. The collection of these modules is called the *common basis*. The most common functions are in the "`Pervasives` module," which means you don't have to do anything special to use them. For less commonly used functions, you have to either prefix the function name with the name of the module, or you have to "open" the module. Opening the module means making the contents visible to the program.

For example, the following sequence shows that the function `length` is not defined for strings until the `open String` command is issued:

```
# length "hello";;
Characters 0-6: Unbound value length
# open String;;
# length "hello";;
- : int = 5
```

Note: Caml Light uses a slightly different syntax-- `#open "string";;` -- where the `#` is typed by the user. The functions provided by each module are also different, often only in having a different name.

Finally, comments are indicated as:

```
(* This is a comment (* and comments may be nested. *) *)
```

Identifiers

Identifiers must begin with a *lowercase* letter or underscore, and may contain letters, digits, underscores, and single quotes. Most implementations also recognize accented characters, such as é.

Alphanumeric identifiers beginning with a `'` (single quote) are used only for type identifiers. Often OCaml will use `'a` to indicate a variable (unknown or arbitrary) type. For example, `'a list` means a list whose elements are of type `'a`. Additional variable types are indicated by `'b`, `'c`, and so on.

The variable `_` (an underscore all by itself) is used as a "wildcard" or "don't care" variable, for example,

```
let second (_, x) = x;;
```

Types

There are several primitive types in OCaml; the following table gives the most important ones.

Primitive type	**Examples**	**Notes**
`int`	`0, 5, 42, -17, 0x00FF, 0o77, 0b1101`	`-` is used for unary minus; there is no unary + `0x` or `0X` starts a hexadecimal number; `0o` or `0O` starts an octal number; and `0b` or `0B` starts a binary number
`float`	`0.0, -5.3, 1.7e14, 1.7e+14, 1e-10`	Can't start with a decimal point
`bool`	`true, false`	These are the only bool values.

`string`	`""`, `"One\nTwo"`	`"\n"` is newline, `"\t"` is tab, `"\\"` is backslash
`char`	`'a'`, `'\n'`	Single quotes for chars, double quotes for strings.
`unit`	`()`	This **is** a value. It is the only value of its type, and is often used when the value isn't important (much like `void` in C).

There are three families of constructed types in OCaml: lists, tuples, and functions.

Lists are enclosed in brackets and the list elements are separated by semicolons. All elements in a list must be of the same type.

Tuples are usually enclosed in parentheses, and the elements are separated by commas. The elements of a tuple may be of different types.

Functions are *first-class* objects: they can be created, manipulated, passed as parameters, and otherwise treated like other kinds of values. (However, when OCaml prints the result of an expression, and that result is a function, OCaml doesn't print out the entire function; it just prints the word **`fn`**.) Every function takes exactly one parameter as input and returns one value as its result; however, that parameter and that result may each be of a constructed type, such as a tuple.

The following table gives a few examples of constructed types. **Pay special attention to the second column,** which shows how OCaml expresses type information.

Example expression	**Expression type**	**Notes**
`[2; 3; 5; 7]`	`int list`	Lists may be of any length, but all elements must be of the same type.
`[]`	`'a list`	The empty list can be represented by `[]`. The type of this list is allowed to be unknown.
`(5, "hello", ~16)`	`int * string * int`	The type of a tuple depends on its length and the types in each position.
`("abc", [1; 2; 3])`	`string * int list`	Tuples can contain lists, and vice versa.
`(3.5)`	`float`	A tuple with one element is the same as that one element.
`let double x = 2.0 *. x;;`	`float -> float`	All functions take exactly **one** parameter, and parentheses are optional.
`let sum (x, y) = x + y;;`	`int * int -> int`	In this example the **one** parameter is a tuple.
`let hi () = print_string "hello\n";;`	`unit -> unit`	In this example the **one** parameter is the "unit," and so is the result.

`(double, [sum])`	`(float -> float) * (int * int -> int) list`	Functions are values, and can be put into lists and tuples.

Built-In Functions

This section lists the most generally useful of the built-in functions; it is not a complete listing. See the appropriate reference manual for additional functions.

An *operator* is just a function with a special syntax. Syntax that is added just for the sake of convenience, and not for any technical reasons, is called *syntactic sugar.* In OCaml, operators can be "de-sugared" by enclosing them in parentheses, for example:

```
# (+) 3 5;;
- : int = 8
```

This also provides a handy way to peek at the type of an operator:

```
# (+);;
- : int -> int -> int = <fun>
```

Standard bool operators:

Function	Examples	Notes
`not : bool -> bool`	`not true, not (i = j)`	(Prefix) Unary negation.
`&& : bool * bool -> bool`	`(i = j) && (j = k)`	(Infix, left associative) Conjunction, with short-circuit evaluation.
`|| : bool * bool -> bool`	`(i = j) || (j = k)`	(Infix, left associative) Disjunction, with short-circuit evaluation.

Standard arithmetic operators on integers:

Function	Examples	Notes
`- : int -> int`	`-5, -limit`	(Prefix) Unary negation.
`* : int * int -> int`	`2 * limit`	(Infix, left associative) Multiplication; operands and result are all ints.
`/ : int * int -> int`	`7 / 3, score / average`	(Infix, left associative) Division; truncates fractional part.
`mod : int * int -> int`	`limit mod 2`	(Infix, left associative) Modulus; result has sign of first operand.

`+ : int * int -> int`	`2 + 2, limit + 1`	(Infix, left associative) Addition.
`- : int * int -> int`	`2 - 2, limit - 1`	(Infix, left associative) Subtraction.
`abs : int -> int`	`abs (-5)`	(Prefix) Absolute value.

Standard arithmetic operators on real numbers:

Function	Examples	Notes
`** : float *. float -> float`	`15.5 ** 2.0`	(Infix, right associative) Exponentiation.
`sqrt : float -> float`	`sqrt 8.0`	(Prefix) Square root.
`-. : float -> float`	`-1e10, -average`	(Prefix) Unary negation.
`*. : float *. float -> float`	`3.1416 *. r *. r`	(Infix, left associative) Multiplication; operands and result are all real numbers.
`/. : float * float -> float`	`7.0 /. 3.5, score /. average`	(Infix, left associative) Division of real numbers.
`+. : float * float -> float`	`score +. 1.0`	(Infix, left associative) Addition of real numbers.
`-. : float * float -> float`	`score -. 1.0`	(Infix, left associative) Subtraction of real numbers.
`** : float *. float -> float`	`15.5 ** 2.0`	(Infix, right associative) Exponentiation.
`sqrt : float -> float`	`sqrt 8.0`	Square root.
`ceil : float -> float`	`ceil 9.5`	Round up to nearest integer (but result is still a real number).
`floor : float -> float`	`floor 9.5`	Round down to nearest integer (but result is still a real number).
`exp, log, log10, cos, sin, tan, acos, ... : float -> float`	`exp 10.0`	The usual transcendental functions.

Coercions

Function	Examples	Notes

`float : int -> float`	`float 5, float (5)`	Convert integer to real.
`truncate : float -> int`	`truncate average`	Fractional part is discarded.
`int_of_char : char -> int`	`int_of_char 'a'`	ASCII value of character.
`char_of_int: int -> char`	`char_of_int 97`	Character corresponding to ASCII value; argument must be in range 0..255.
`int_of_string : string -> int`	`int_of_string "54"`	Convert string to integer.
`string_of_int : int -> string`	`string_of_int 54`	Convert integer to string.
`float_of_string : string -> float`	`float_of_string "3.78"`	Convert string to float.
`string_of_float : float -> string`	`string_of_float 3.78`	Convert float to string.
`bool_of_string : string -> bool`	`bool_of_string "true"`	Convert string to bool.
`string_of_bool : bool -> string`	`string_of_bool true`	Convert bool to string.

Comparisons

Function	**Examples**	**Notes**
`< : 'a * 'a -> bool`	`i < 0`	Less than. `a'` can be `int, float, char,` or `string`.
`<= : 'a * 'a -> bool`	`x <= 0.0`	Less than or equal to. `a'` can be `int, float, char,` or `string`.
`= : 'a * 'a -> bool`	`s = "abc"`	Equals. `a'` can be `int, char,` or `string,` but not `float`.
`<> : 'a * 'a -> bool`	`ch <> '\n'`	Not equal. `a'` can be `int, char,` or `string,` but not `float`.
`>= : 'a * 'a -> bool`	`i >= j`	Greater than or equal to. `a'` can be `int, float, char,` or `string`.
`> : 'a * 'a -> bool`	`x > y`	Greater than. `a'` can be `int, float, char,` or `string`.
`== : 'a -> 'a -> bool`	`x == y`	Physical equality; meaning is somewhat implementation-dependent.

`!= : 'a -> 'a -> bool`	`x != y`	Physical inequality; meaning is somewhat implementation-dependent.
`max : 'a -> 'a -> 'a`	`max 'a' 'v',` `max 0 n`	Returns the larger of the two arguments.
`min : 'a -> 'a -> 'a`	`min ch1 ch2`	Returns the smaller of the two arguments.

Operations with strings

The operators `<  <=  =  !=  >=  >` can be applied to strings for lexical comparisons. Many of the following operations can be used only by opening the `String` module or prefixing the name of the operation with `String` (e.g. `String.length "hello"`).

Function	Examples	Notes
`^ : string * string -> string`	`"Hello, " ^ name`	Infix concatenation of two strings.
`String.concat : string -> string list -> string`	`String.concat " and " ["ab"; "c"; "de"]`	Concatenates the strings of the list with the first argument inserted between each pair.
`String.length : string -> int`	`String.length "hello"`	Number of characters in string.
`String.get : string -> int -> char`	`String.get "hello" 0`	Return a given character of the string, counting from 0.
`String.set : string -> int -> char -> unit`	`String.set name 4 's'`	Modifies the original string by changing the character at the given location.
`String.index : string -> char -> int`	`String.index "radar" 'a'`	Returns the position of the first occurrence of the char in the string.
`String.rindex : string -> char -> int`	`String.rindex "radar" 'a'`	Returns the position of the last occurrence of the char in the string.
`String.contains : string -> char -> bool`	`String.contains "radar" 'a'`	Tests whether the char occurs in the string.
`String.sub : string -> int -> int -> string`	`String.sub "abcdefg" p n`	Returns a substring of length `n` starting at position `p`.
`String.make : int -> char -> string`	`String.make n c`	Returns a string consisting of `n` copies of character `c`.
`String.uppercase : string -> string`	`String.uppercase "OCaml"`	Returns a copy of the string with all letters translated to uppercase.

`String.lowercase : string -> string`	`String.lowercase "OCaml"`	Returns a copy of the string with all letters translated to lowercase.
`String.capitalize : string -> string`	`String.capitalize "OCaml"`	Returns a copy of the string with the first character translated to uppercase.
`String.uncapitalize : string -> string`	`String.uncapitalize "OCaml"`	Returns a copy of the string with the first character translated to lowercase.

Special syntactic sugar for accessing characters of a string:

`s.[i]`	Returns the `i`th character of string `s`.
`s.[i] <- c`	Sets the `i`th character of string `s` to `c`.

Operations on characters

The operators `<` `<=` `=` `!=` `>=` `>` can be applied to characters.

The following functions are in the `Char` structure. To use these functions without typing `Char.` each time, enter **`open Char;;`**.

Function	Notes
`Char.uppercase : char -> char`	Given a lowercase letter, returns the corresponding capital letter. Given any other character, returns that same character.
`Char.lowercase : char -> char`	Given a capital letter, returns the corresponding lowercase letter. Given any other character, returns that same character.
`Char.escaped : char -> string`	Returns a string consisting of the single character. The name refers to the fact that the character may be escaped (quoted).

Operations on lists

A list is a set of elements, all of the same type, enclosed in brackets and separated by semicolons. Example: `["hello"; "bonjour"; "guten Tag"]`. The type of this example is `string list`.

The empty list is represented by `[]`.

Only the `::` operator (LISP cons) and `@` operator (list concatenation) can be used without opening the `List` module or prefixing the function name with `List.` .

Function	Examples	Notes

`:: : 'a -> 'a list -> 'a list`	`5 :: [6; 7]`	Add an element to the front of the list. This operator is right associative.
`@ : 'a list -> 'a list -> 'a list`	`[5] @ [6; 7]`	List concatenation.
`List.length : 'a list -> int`	`List.length [5; 6; 7]`	Number of elements in the list
`List.hd : 'a list -> 'a`	`List.hd [3; 5; 7]`	The "head" of a list is its first element. Same as `car` in LISP.
`List.tl : 'a list -> 'a list`	`List.tl [3; 5; 7]`	The "tail" of a list is the list with its first element removed. Same as `cdr` in LISP.
`List.nth : 'a list -> int -> 'a`	`List.nth [3; 5; 7] 2`	Returns the nth element of a list, counting from zero.
`List.rev : 'a list -> 'a list`	`List.rev [1; 2; 3]`	Reverse the list.

Operations on tuples

Remember that a tuple consists of zero or more values, separated by commas and enclosed in parentheses. The parentheses can usually (but not always) be omitted.

If `T` is a pair (a tuple of two elements), then `fst(T)` is the first element and `snd(T)` is the second element. Standard ML defines additional operations on tuples, but OCaml does not.

The type of a tuple describes the number and type of each element in the tuple. For example,

```
# ("John Q. Student", 97, 'A');;
- : string * int * char = "John Q. Student", 97, 'A'
```

Sometimes you may want a function to return more than one value. That isn't possible, but the next best thing is to return a tuple of values.

```
# let divide x y = x / y, x mod y;;
val divide : int -> int -> int * int = <fun>
# divide 20 3;;
- : int * int = 6, 2
```

You can easily define functions to work with tuples, by using patterns. For example:

```
# let third_of_four (_, _, x, _) = x;;
val third_of_four : 'a * 'b * 'c * 'd -> 'c = <fun>
# third_of_four ('a', 'b', 'c', 'd');;
 - : char = 'c'
```

Functions with side effects

The most useful functions with side effects are the I/O functions. Since OCaml automatically prints the values of expressions entered at the prompt, you can often avoid doing any explicit I/O.

Function	Examples	Notes
`print_char : char -> unit`	`print_char 'a'`	Prints one character
`print_int : int -> unit`	`print_int (x + 1)`	Prints an integer.
`print_float : float -> unit`	`print_float 5.3`	Prints a real number.
`print_string : string -> unit`	`print_string mystring`	Prints a string.
`print_endline : string -> unit`	`print_endline "Hello"`	Prints a string followed by a newline.
`print_newline : unit -> unit`	`print_newline "Hello"`	Prints a string followed by a newline, then flushes the buffer.

The printing functions all return the unit, `()`, which is OCaml's way of saying that nothing important is returned.

Function	Examples	Notes
`read_line : unit -> string`	`read_line ()`	Reads a string from standard input.
`read_int : unit -> int`	`read_int ()`	Reads an integer from standard input.
`read_float : unit -> float`	`read_float ()`	Reads a real number from standard input.

Expressions (but not statements)

OCaml is a purely functional language: everything is built from functions. Functions return values. Statements don't return values, they have side effects. There are no statements in OCaml. Things in OCaml that look like statements are actually expressions, and return values.

The *state* of a program is the collection of values in the environment. Statements, especially assignment statements, have *side effects*--that is, they change the state of a program by altering values in the environment. To understand what a program is doing, you have to know the state of the program. Since states can be very large, understanding a program can be difficult.

OCaml claims to be *stateless.* That is, you don't need to know about any environmental variables in order to understand a program. Everything you need is either in the function parameters or in the execution stack (the set of function calls that got you to this point). In theory, this should make an OCaml program easier to understand than a program in a nonfunctional language. There is actually some truth in this point of view.

Because you can open structures and define and redefine values and functions at the prompt, the argument that OCaml is stateless is a little difficult to justify. However, if all your functions are defined in a file, and you just read in that file and use the functions, the stateless nature of OCaml is more obvious.

Match expressions

The *match expression* looks like this:

```
match <expression> with <match>
```

where a `<match>` has the form:

```
<pattern_1> -> <expression_1> |
<pattern_2> -> <expression_2> |
. . .
<pattern_N> -> <expression_N>
```

First, the initial `<expression>` is evaluated, then its value is compared against each of the `<patterns>`. When a matching `<pattern_i>` is found, the corresponding `<expression_i>` is evaluated and becomes the value of the case expression.

The most common patterns are

- a variable (matches anything),
- a tuple, such as `(x, y)`,
- a literal (constant) value, such as `5` or `"abc"`,
- an expression `x::xs`, to match the head and tail of a nonempty list, and
- an `as` expression to have a name for the entire actual parameter as well as its parts; for example, `L as x::xs`, which matches the same things as `x::xs`, but also assigns to `L` the entire list.

Examples:

```
match (n + 1) with 1 -> "a" | 2 -> "b" | 3 -> "c" | 4 -> "d";;

match myList with [] -> "empty" | x::xs -> "nonempty";;
```

Because every expression must have a value, OCaml will warn you if it does not think you have a pattern for every possible case.

When expressions are nested, it can be difficult to tell exactly where the case expression ends. As with any expression, it doesn't hurt to put parentheses around it, as for example,

```
(match (n + 1) with 1 -> "a" | 2 -> "b" | 3 -> "c" | 4 -> "d");;
```

If expressions

The *if expression* looks like this:

```
if <bool expression> then <expression_1> else <expression_2>
```

If the `<bool expression>` is true, then `<expression_1>` is evaluated and is the value of the expression, otherwise `<expression_2>` is evaluated and is the value of the expression.

Since the if expression is an expression and not a statement, it must have a value; therefore, the `else` part is required.

The if expression is really shorthand for the following match expression:

```
match <bool expression> with true -> <expression1> | false ->
<expression2>
```

When the OCaml compiler detects an error in an if expression, it reports the error as though it occurred in the corresponding case expression.

Sequence of expressions

We can execute a sequence of expressions by separating them with semicolons and enclosing the group in parentheses; the value of the sequence is the value of the last expression in the sequence. Any values produced by earlier expressions are discarded. This is only useful with expressions that are evaluated for their side effects.

```
let name = "Dave" in
  (print_string "Hello, "; print_string name; print_string "\n");;
```

OCaml is supposed to be a purely functional language, which means that it has no expressions with side effects. However, printing output *is* a side effect, and output is the primary use of side effects in OCaml. Output operations produce the unit, `()`, as their value. OCaml therefore expects all the expressions (except the last) in a semicolon-separated list to have a unit value, and will warn you if that isn't the case.

Exceptions

An "exception" is an error. You can declare new types of exceptions, with or without a parameter, as follows:

```
exception <Name> ;;
exception <Name> of <type> ;;
```

The name of an exception must begin with a capital letter. For example:

```
exception FormatError of int * string;;
```

To signal that one of these exceptions has occurred, use

```
raise (<name> arguments);;
```

The way you use exceptions is as follows:

```
try <expression> with <match>
```

If no exception is raised, the result of the `try` is the result of the *`<expression>`*. If an exception is raised, the first rule in *`<match>`* that matches the expression will be executed. If you raise an exception and fail to handle it, OCaml will give you an "uncaught exception" error.

Note that the results of the match must all have the same type as the result of a correct *`<expression>`*. However, it's okay to use a sequence of expressions for printing, ended with a value of the correct type, as in the following example.

Example:

```
exception WrongOrder of int*int;;
```

```
let rec fromTo (m, n) =
  if m > n then raise (WrongOrder(m, n))
  else if m = n then [m]
       else m::fromTo(m + 1, n);;

let makeList (m, n) =
  try fromTo (m, n)
  with WrongOrder(m, n) ->
    (print_int m;
     print_string " is greater than ";
     print_int n;
     print_newline ();
     []);;
```

Functions

The usual form of a function definition is

```
let <name> <parameter> = <expression> ;;
```

A function always has exactly one parameter, and returns one result. It often *appears* that a function takes more than one argument or returns more than one result, but appearances can be misleading. For instance,

```
let swap (x, y) = (y, x);;
```

In this case, the one argument is a tuple, and the one result is a tuple.

```
print_newline ();;
```

Here, the one argument is the *unit*, `()`, which *is* a value--it is *not* syntax to indicate an empty parameter list. Similarly, the unit is returned as a result. Finally,

```
let max x y = if x > y then x else y;;
```

This is an operation called *currying*, in which `max` takes the single argument `x`, and returns a *new function* which takes the single argument `y`. Currying is explained in more detail later in this paper.

It doesn't hurt to use a tuple of length 1 in place of a single parameter; the following are equivalent definitions:

```
let score  x   = if x < 0 then 0 else x;;
let score (x)  = if x < 0 then 0 else x;;
```

Functions that don't use their parameter must still have one; they can be given the unit as the parameter:

```
let tab () = print_char '\t';;
```

Similarly, a function can only return one value, but that value can easily be a tuple. For example,

```
let vectorAdd ((x1, y1), (x2, y2)) = (x1 + x2, y1 + y2);;
```

Recursive functions must include the keyword `rec`, for example:

```
let rec firstDigit (x: int) =
        if x < 10 then x else firstDigit (x / 10);;
```

Patterns in functions

Simple functions consist of only a single case, but more complex functions typically use pattern matching to separate cases. Consider the well-known Fibonacci function,

```
fib(0) = fib(1) = 1
fib(n) = fib(n-1) + fib(n-2), for n > 1
```

We can implement this in OCaml as follows:

```
let rec fibonacci x =
  match x with
    0 -> 1
  | 1 -> 1
  | n -> fibonacci (x - 1) + fibonacci (x - 2);;
```

Using a `match` in this way is so common that there is a special syntax for it, as follows:

```
let rec fibonacci2 = function
    0 -> 1
  | 1 -> 1
  | n -> fibonacci2 (n - 1) + fibonacci2 (n - 2);;
```

Notice that:

- The parameter (in this case, `x`) is omitted
- The word `function` replaces `match x with`
- Instead of using parameter `x`, the parameter is matched against a variable `n`, which can then be used in place of `x`

Examples:

```
let rec length = function
    [] -> 0
  | (x :: xs) -> 1 + length xs;;

let rec member = function
    (e, []) -> false
  | (e, x::xs) ->
      if (e = x)
        then true
        else member (e, xs) ;;
```

Mutually recursive functions

Functions must be defined before they are used. To define mutually recursive functions (functions that call one another), use `let rec...and...and...`. The following example (to return every other element of a list) is adapted from *Elements of Caml Programming* by Jeffrey D. Ullman:

```
let rec
   take (ls) =
      if ls = [] then []
      else List.hd(ls) :: skip(List.tl(ls))
and
   skip (ls) =
      if ls = [] then []
      else take(List.tl(ls)) ;;
```

Local variables in functions

It is sometimes convenient to declare local variables and functions using `let...in...` For example,

```
# let circleData (radius:float) =
    let pi = 3.1415926536 in
    let circumference = 2.0 *. pi *. radius in
    let area radius = pi *. radius *. radius in
      (circumference, area radius)
;;
          val circleData : float -> float * float = <fun>
# circleData 10.0;;
- : float * float = 62.831853072, 314.15926536
```

The `let` and `let...in` expressions work with patterns, so it is possible to do "multiple assignment" by assigning one tuple to another, as for example

```
# let x, y, z = 5, 10, "hello";;
val x : int = 5
val y : int = 10
val z : string = "hello"
# let (a, b) = (6, 7) in (b, a, b);;
- : int * int * int = 7, 6, 7
```

It is useful to know that multiple assignments happen simultaneously rather than one after another, so that the following code correctly swaps two values:

```
# let swap (x, y) = y, x;;
val swap : 'a * 'b -> 'b * 'a =
# swap (3, 5);;
- : int * int = 5, 3
```

Anonymous functions

Matches can also be used to define anonymous functions:

```
(fun x -> x + 1) 3
```

(Result is 4.) Anonymous functions cannot be recursive because, being anonymous, they have no name by which you can call them. However, an anonymous function can be given a name in the usual way, by using `let`:

```
# let incr = fun x -> x + 1;;
val incr : int -> int = <fun>
# incr 5;;
- : int = 6
```

Polymorphic functions

In other languages, "polymorphic" means that you have two or more functions with the same name; in OCaml it means that a single function can handle more than one type of parameter.

An example of a built-in polymorphic function is the list function `List.hd`; it returns the first element of any type of list.

OCaml functions that you write will be polymorphic if their parameters are used only with polymorphic functions and operators. For example, the following function to reverse the two components of a 2-tuple is polymorphic:

```
let revPair (x, y) = (y, x);;
```

To write a polymorphic function, you need to *avoid:*

- arithmetic operators (because OCaml needs to know whether the arithmetic is integer or floating point),
- string concatenation
- boolean operators
- type conversion operators.

You *can* use:

- lists and the list operators `hd`, `tl`, `::`, `@`, along with the constant `[]`
- the equality tests `=` and `!=`.

Higher-order functions

A higher-order function is one which takes a function (or a function-containing structure) as an argument, or produces a function (or function-containing structure) as its result, or both. For example,

```
# let test (f, x) = f x;;
val test : ('a -> 'b) * 'a -> 'b = <fun>
# test (List.hd, [1;2;3]);;
- : int = 1
# test (List.tl, [1;2;3]);;
# - : int list = [2; 3]
```

Notice in particular the type returned when `test` is defined: `('a -> 'b) * 'a -> 'b`. Here, `('a -> 'b)` is a function from type `'a` to type `'b`, type `'a` is the needed parameter type, and `'b` is the result type.

Curried functions

As was stated earlier, every function takes exactly one argument and produces one result.

A curried function is a higher-order function that takes an argument and produces as result a new function with that argument embedded in the function. For example,

```
# let incr x y = x + y;;
val incr : int -> int -> int = <fun>
# incr 5 3;;
- : int = 8
# (incr 5) 3;;
- : int = 8
# let incr5 = incr 5;;
val incr5 : int -> int = <fun>
# incr5 3;;
- : int = 8
```

Notice the way the function `incr` is defined, as if it had two blank-separated arguments. In fact, `incr` takes one argument, `x`, and produces the curried function `(incr x)`, which then has `y` as an argument. Now `incr` can be called as `incr 5 3` or as `(incr 5) 3`, but *cannot* be called as `incr (5, 3)`. This is because `incr` takes one argument, an integer, and returns a function as a result; it does *not* take a tuple as an argument.

The fact that `incr 5` returns a function as a result is further emphasized in the assignment of this value (a function that adds 5 to its argument) to the variable `incr5`,. Note also that we don't need to specify a parameter; a function is just another kind of value. We could, if we wished, have defined `incr5` by `let incr5 x = incr 5 x;;` with exactly the same result.

`map`

`List.map` is a curried function that takes a function that applies to one thing of type `'a` and produces a function that applies to a list `'a`. For example,

```
# truncate 5.8;;
- : int = 5
# List.map truncate [2.7; 3.1; 3.8; 9.4; 6.5];;
- : int list = [2; 3; 3; 9; 6]
# (List.map truncate) [2.7; 3.1; 3.8; 9.4; 6.5];;
- : int list = [2; 3; 3; 9; 6]
# let listTrunc = List.map truncate;;
val listTrunc : float list -> int list = <fun>
# listTrunc [2.7; 3.1; 3.8; 9.4; 6.5];;
- : int list = [2; 3; 3; 9; 6]
```

`List.filter`

The function `List.filter` takes a bool test and returns a function that will extract from a list those elements that pass the test.

```
# List.filter (fun x -> x > 0) [3; 0; 2; -5; -8; 4];;
- : int list = [3; 2; 4]
```

Redefining functions

Important: When you define a function, you may use other values (including other functions) in the definition. Later changes to those values do not change the meaning of the function you have defined. For example, look at the following sequence (for clarity, only some of OCaml's responses are shown):

```
# let x = 3;;
# let aa () = 5;;
# let bb () = aa () + x;;
# bb ();;
- : int = 8
# let x = 17;;
# let aa () = 83;;
# bb ();;
- : int = 8
```

Here, `bb()` was defined in terms of `aa()` and `x`. Calling `bb()` gave the result `8`. Later changes to `aa()` and `x` *did not affect* the definition of `bb()`.

The above may seem strange, but there is a parallel in algorithmic languages. Consider the sequence of statements `a := 5; b := a; a := 10;` You would not expect the value of `b` to change just because the value of `a` changed. In OCaml, functions are values, and they are treated just like any other values.

A definition is said to have *referential transparency* if its meaning does not depend on the context it is in. Functions in OCaml have referential transparency, that is, changing the context (other variables and other functions) does not change the meaning of any functions you have already defined. This fact can be crucial when you are debugging a program, because you are likely to be redefining functions fairly frequently.

Omissions

This is a brief document, and a great deal has been omitted. Among the more important omissions are records, arrays, the input/output system, most of the module system, and practically everything having to do with object-oriented programming. Loops, also, have been omitted, but they are not terribly useful in a purely functional language, anyway. More information (and more trustworthy information!) can be obtained from the documents mentioned in the next section.

Resources

The Objective Caml system release 3.01, Documentation and user's manual. Xavier Leroy (with Damien Doligez, Jacques Garrigue, Didier R�my and J�r�me Vouillon), March 8, 2001. This is available at http://caml.inria.fr/ocaml/htmlman/index.html and on the Burks 5 CD.

An introduction to OCAML for CSE 120. Peter Buneman, October 1, 2000. This is available as an Acrobat document at http://www.seas.upenn.edu:8080/~cse120/resources/primer/primer.pdf and as a postscript document at http://www.seas.upenn.edu:8080/~cse120/resources/primer/primer.ps.

An OCaml book, apparently nameless, by Jason Hickey, at http://www.cs.caltech.edu/cs134/cs134b/book.pdf.