1 Expressions

Every expression has exactly one type.

\[
\begin{align*}
\text{let } & \text{name = expr ; ;} \\
\text{let } & \text{name arg1 arg2 \ldots argn = expr ; ;} \\
\text{let } & \text{name = fun arg1 -> fun arg2 -> \ldots -> fun argn -> expr ; ;} \\
\text{let } & \text{name = expr1 in expr2 ; ;}
\end{align*}
\]

2 Basic types and operators

<table>
<thead>
<tr>
<th>type</th>
<th>operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>+, /, -, *</td>
</tr>
</tbody>
</table>
| float    | +., /., -. | *.
| string   | ^ (concatenation) |
| bool     | && (and), || (or) |
| 'a * 'b  | fst, snd |
| 'a list  | @(concatenation), :: (append) |

Many useful functions convert between types.

\[
\begin{align*}
\# \text{int_of_string } & \text{"100" ; ;} \\
- : & \text{int = 100} \\
\# \text{string_of_int} & \text{23 ; ;} \\
- : & \text{string = "23"} \\
\# \text{float_of_int} & \text{4 ; ;} \\
- : & \text{float = 4.} \\
\# \text{int_of_float} & \text{2.34 ; ;} \\
- : & \text{int = 2}
\end{align*}
\]

Many more functions for these types can be found in the Pervasives module.

Exercise 1 Write a function that takes a pair of integers as input and returns a string representation of the pair with surrounding parenthesis and a comma between them.
3 Modules String and List

Ocaml support modules. Modules are a package of functions and types that serve some coherant purpose. The standard library has modules that serve many purposes.

A function in a module is called by typing the module name, a period, and then the function.

```ocaml
String.contains "hello" 'a';
String.length "hello" 'a';
List.head [2;3;4;5];
List.tail [6;7;8;9];
```

4 Conditionals

```ocaml
if expr1 then expr2 ;;
if expr1 then expr2 else expr3 ;;
```

The following constraints must be observed:

1. `expr1` must evaluate to type `bool`.

2. In the former usage, `expr2` must evaluate to type `unit`.

3. In the latter usage, `expr2` and `expr3` must evaluate to the same type.

Exercise 2 What is wrong with this function?

```ocaml
let is_less_than3 x =
  if x < 3
  then print_endline "yes it is!"
  else "no it is not!";;
```

5 Pattern matching

Pattern matching is a more effective way of handling multiple cases.

```ocaml
match expr with
  | pattern1 -> expr1
  | pattern2 -> expr2
  | pattern3 -> expr3
```
A pattern is an expression composed of constants and variable names. When a pattern matches an argument, the variables are bound to values of the corresponding expressions. Patterns may contain arbitrary data structures (tuples, records, variant types, lists and arrays) and, in particular, the operator `::` may be used in a pattern to decapitate a list. Also, the pattern `_` matches any value without assigning a name to it.¹

```
let gimme_info n =
  match n with
  | 0    -> "it's zero!"
  | 2    -> "it's two!"
  | 100  -> "it's 100!"
  | _    -> "Oh I don't care about this number."

let rec factorial n =
  if n < 1 then 1 else n * factorial (n-1);;

let rec pointwise_add_one list =
  match list with
  | []    -> []
  | h::t  -> (h+1)::(pointwise_add_one t);;
```

6 Recursive functions

Recursive functions can call themselves. It is necessary to use the keyword `rec` after `let` when defining recursive functions.

```
let rec factorial n =
  if n < 1 then 1 else n * factorial (n-1);;

let rec pointwise_add_one list =
  match list with
  | []    -> []
  | h::t  -> (h+1)::(pointwise_add_one t);;
```

Exercise 3 Write functions which like `pointwise_add_one` which

1. multiplies every number in the list times 2
2. concatenates the string "\n" after each string in the list. ("\n" is the newline character.)
3. maps each string in the list to its length.

Exercise 4 Write a function like `pointwise_add_one` which take two arguments. The first argument is a function which is to be applied to every element in the list. Examine the function you wrote. What is its type?

Exercise 5 Write a recursive function which takes a list of integers and adds them to find their sum. Use pattern matching like above. What should happen if an empty list is given to this function?

¹This passage is from page 17 of *Ocaml for Scientists* by Jon Harrop.
Exercise 6  Write a recursive function which takes a list of strings and concatenates them together. Write a similar function which puts a space between them. Write another which puts parentheses around the string each time it is concatenated.

Exercise 7  How would you generalize the functions above in the same way we generalized the pointwise_add_one function?