OCAML basics

OCAML is a statically typed programming language that is part of the ML family. It emphasizes functional programming with a fairly rich type system and an interesting module system.

Expressions, Values, Types

Everything in OCAML is an expression. Expressions are evaluated by successively performing replacements based on the language rules, until we arrive at an expression that cannot be further evaluated. These are called **values**. Here is a simple example of such a sequence of evaluations:

```
(4 + 5) + (2 * 3)
9 + (2 * 3)
9 + 6
15
```

Every value has a **type**. The main types in OCAML are:

- int for integers values
- float for double-precision floating point values
- char for single characters, e.g. 'x'
- string for strings, e.g. "fgrw"
- bool for the two boolean values true and false
- unit is a special type with only one value, ()

In OCAML there is no automatic coercion between types. We will learn about other more complex types as we move on with the material.

The three key questions

For each language construct we examine three questions:

1. What is the syntax for it?
2. What are the typechecking rules for it?
3. What are the evaluation rules (semantics) for it?

With that in mind, let's consider various constructs in turn:

Arithmetic Operators

Most of these operators are defined in what is known as the **Pervasives** module, a library that is automatically included when you start OCAML. We will only discuss addition here, but you can as exercise write down similar instructions for other operators:

Syntax $e_1 + e_2$ where e_1 and e_2 are expressions.

Typechecking Both e_1 and e_2 must have type int. The whole expression has type int.

Evaluation We first evaluate e_1 to an integer v_1, then we evaluate e_2 to an integer v_2, then we add those two values.

Conditional

There is one conditional expression, and unlike most conditional constructs you might have seen in other languages, it is an *expression*, and thus evaluates to a value.

Syntax if e_1 **then** e_2 **else** e_3 where e_1, e_2, e_3 are expressions.

Typechecking e_1 must be of type bool, and e_2 and e_3 must be of the same type. That type is then the type of the expression.

Evaluation We first evaluate e_1. If it evaluates to true then the expression evaluates to the result of evaluating e_2. If e_1 evaluated to false, then the expression evaluates to the result of evaluating e_3.

Bindings, Environments

An OCAML program is a sequence of **bindings**, that look like this:

```
let x = 2 + 3
```

Each binding is evaluated by evaluating the expression on the right side of the equal sign, and adding to the **dynamic environment** the "binding" $x \mapsto 5$. Later bindings can use the variable "x". There is also a **static environment**, used during the typechecking process, which holds a "binding" for the type of x.

So we have two new language constructs to consider: bindings, and variables.

Bindings:

Syntax let $s = e$ where s is an identifier and e is an expression.

Typechecking We typecheck e, and say it has a type t. Then this is the type of the whole expression, and furthermore the static environment is extended with a binding $x \mapsto t$.

Evaluation We evaluate the expression e, and say it produces a value v. This is then the value of the whole expression, and furthermore the dynamic environment is extended with a binding $x \mapsto v$.

Variables:

Syntax s where s is an identifier.

Typechecking We look for a binding for s in the static environment and return what we find.

Evaluation We look for a binding for s in the dynamic environment and return what we find.

Local Bindings

Local bindings have a very temporal nature. Instead of extending the environment for all future computations, they only extend it for a specific computation. So we are basically saying "let x stand for this value while you compute this expression".

Syntax **let** $s = e_1$ **in** e_2 where s is an identifier and e_1 and e_2 are expressions.

Typechecking Compute the type of e_1 in the current static environment, and suppose this results in a type t_1. Then temporarily extend that environment with a binding $s \mapsto t_1$, and compute the type of e_2 is this extended environment. The resulting type is the type of the expression.

Evaluation Evaluate e_1 in the current dynamic environment, and suppose this results in a value v_1. Then temporarily extend that environment with a binding $s \to v_1$ and evaluate e_2 in this extended environment. The resulting value is the value of the expression.

As a little teaser, determine the value of the following expression:

```
let x = (let x = 3 in x + 1)
in (let x = x + 8 in 2 * x) + x
```