CMSC 330: Organization of Programming Languages

Working with OCaml

OCaml Compiler

- OCaml programs can be compiled using ocamlc
  - Produces .cmo ("compiled object") and .cmi ("compiled interface") files
    - We'll talk about interface files later
  - By default, also links to produce executable a.out
    - Use -o to set output file name
    - Use -c to compile only to .cmo/.cmi and not to link
- Can also compile with ocamlopt
  - Produces .cmx files, which contain native code
  - Faster, but not platform-independent (or as easily debugged)

OCaml Compiler

- Compiling and running the following small program:

hello.ml:

```
(* A small OCaml program *)
print_string "Hello world!\n";;
```

```
% ocamlc hello.ml
% ./a.out
Hello world!
%
```

OCaml Compiler: Multiple Files

main.ml:

```
let main () =
  print_int (Util.add 10 20);
  print_string "\n"

let () = main ()
```

util.ml:

```
let add x y = x+y
```

- Compile both together (produces `a.out`)

  ```
  ocamlc util.ml main.ml
  ```

- Or compile separately

  ```
  ocamlc -c util.ml
  ocamlc util.cmo main.ml
  ```

- To execute

  ```
  ./a.out
  ```

OCaml Top-level

- The *top-level* is a read-eval-print loop (REPL) for OCaml
  - Like Ruby's **`irb`**

- Start the top-level with the **`ocaml`** command:

```
ocaml
        OCaml version 4.07.0
# print_string "Hello world!\n";;
Hello world!
- : unit = ()
#
```

- To exit the top-level, type ^D (Control D) or call the exit 0

```
# exit 0;;
```

OCaml Top-level (cont'd)

Expressions can also be typed and evaluated at the top-level:

```
# 3 + 4;;
- : int = 7

# let x = 37;;
val x : int = 37

# x;;
- : int = 37

# let y = 5;;
val y : int = 5

# let z = 5 + x;;
val z : int = 42

# print_int z;;
42- : unit = ()

# print_string "Colorless green ideas sleep furiously";;
Colorless green ideas sleep furiously- : unit = ()

# print_int  "Colorless green ideas sleep furiously";;
This expression has type string but is here used with type int
```

gives type and value of each expr

"-" = "the expression you just typed"

unit = "no interesting value" (like void)

Loading Files in the Top-level

File **hello.ml** :

```
print_string "Hello world!\n";;
```

- Load a file into top-level

  #use *"filename.ml"*

- Example:


```
# #use "hello.ml";;
Hello world!
- : unit = ()
#
```

OPAM: OCaml Package Manager

- opam is the package manager for OCaml
  - Manages libraries and different compiler installations
- We recommend installing the following packages with opam
  - OUnit, a testing framework similar to minitest
  - Utop, a top-level interface similar to irb
  - Dune, a build system for larger projects

Ocamlbuild: Smart Project Building

- Use **`ocamlbuild`** to compile larger projects and automatically find dependencies
- Build a bytecode executable out of **`main.ml`** and its local dependencies

```
ocamlbuild main.byte
```

- The executable **`main.byte`** is in **`_build`** folder. To execute:

```
./main.byte
```

Dune: Smarter Project Building

- Use **`dune`** to compile larger projects and automatically find dependencies
- Define a dune file, similar to a Makefile:

dune:

```
(executable
 (name main))
```

Indicates that an executable (rather than a library) is to be built

Name of main file (entry point)

```
% dune build main.exe
% _build/default/main.exe
30
%
```

Check out https://medium.com/@bobbypriambodo/starting-an-ocaml-app-project-using-dune-d4f74e291de8

Dune commands

- If defined, run a project's test suite:

```
dune runtest
```

- Load the modules defined in src/ into the **`utop`** top-level interface:

```
dune utop src
```

- **`utop`** is a replacement for **`ocaml`** that includes dependent files, so they don't have be be **`#load`**ed

A Note on ;;

- ;; ends an expression in the top-level of OCaml
  - Use it to say: "Give me the value of this expression"
  - Not used in the body of a function
  - Not needed after each function definition
    - Though for now it won't hurt if used there
- There is also a single semi-colon ; in OCaml
  - But we won't need it for now
  - It's only useful when programming imperatively, i.e., with side effects
    - Which we won't do for a while