```
$Id: asg2-ocaml-interp.mm,v 1.21 2019-01-24 19:02:56-08 - - $
PWD: /afs/cats.ucsc.edu/courses/cmps112-wm/Assignments/asg2-ocaml-interp
URL: http://www2.ucsc.edu/courses/cmps112-wm/:/Assignments/asg2-ocaml-interp/
```

1. Overview

This project will repeat the Silly Basic interpreter, except this time the program will be written in Ocaml but with Silly Basic programs untranslated from the original. See the **.score/** directory for sample input files. Output should be the same as for the Scheme version of the program, except for minor variations in output due to differences between the Scheme and Ocaml languages. Any results which would produce a complex value in Scheme should produce **nan** in this project.

2. Running **ocaml** interactively

Ocaml may be run interactively from the command line or as a compiled program. The compiled program version, created using **make** is required for all submitted programs.

To run **ocaml** interactively, add the following to your **$HOME/.bash_profile**:

```
export PATH=$PATH:/afs/cats.ucsc.edu/courses/cmps112-wm/usr/ocaml/bin
```

When running **ocaml** interactively, use the command **rlwrap** to gain access to the readline arrow keys so that you can recover earlier typed lines. Example:

```
-bash-$ rlwrap ocaml
        OCaml version 4.02.1
# let f x y = x +. y;;
val f : float -> float -> float = <fun>
# f 3.;;
- : float -> float = <fun>
# f 3. 4.;;
- : float = 7.
# ^D
```

To simplify typing, the following line might be added to your **$HOME/.bash_profile**:

```
alias ocaml="rlwrap ocaml"
```

The suggestions above assume you are using **bash** as your login shell. If not, use the syntax appropriate for whatever shell you are using.

Some files that are useful when running interactively are:

using
: A set of **#use** directives which can be used for interactive testing of the functions. This file is not used in compilation. After starting Ocaml, type in the following command to load your source code interactively:

```
#use "using";;
```

.ocamlinit
: As an alternative to the **using** file, create the file **.ocamlinit** containing the same information. The file **.ocamlinit** in the current directory is automatically sourced when **ocaml** starts.

As an alternative, start up **ocaml** with the line

```
rlwrap ocaml -init using
```

which will start up the init file when needed, but avoid the automatic startup when you don't want it. If you have a **.ocamlinit** and want to ocassionally suppress it,

you can use

```
rlwrap ocaml -init /dev/null
```

3. Source code

The following files and modules are provided in the `code/` subdirectory :

`etc.mli`, `etc.ml`
: Interface and implementation of the `Etc` module, which contains miscellaneous functions not specifically tied to other purposes.

`absyn.mli`
: Definition of the abstract syntax used by the interpreter. No implementation file is needed.

`tables.mli`, `tables.ml`
: Module for maintaining the five tables needed by the program. The interface file is automatically generated from the implemenation, not entered manually. The required tables and their types are :

  `label_table`
  : Labels with pointers to the list of program statements.
    `type label_table_t = (string, Absyn.program) Hashtbl.t`

  `unary_fn_table`
  : The unary functions.
    `type unary_fn_table_t = (string, float -> float) Hashtbl.t`

  `binary_fn_table`
  : The binary functions.
    `type binary_fn_table_t = (string, float -> float -> float) Hashtbl.t`
    Because Ocaml is strongly typed, the unary and binary functions need to be be in separate tables.

  `variable_table`
  : The simple variables used by the program.
    `type variable_table_t = (string, float) Hashtbl.t`

  `array_table`
  : The arrays used by the program.
    `type array_table_t = (string, float array) Hashtbl.t`

`interp.mli`, `interp.ml`
: The interface and implementation of the interpreter. This is the major project of this program and must be extensively modified.

`main.ml`
: The main function which behaves differently, depending on whether the program is run interactively or from the command line. Does the parsing to create the abstract syntax structure, then calls the interpreter.

`parser.mly`
: The parser reads a Silly Basic program, verify syntax, and create the abstract syntax. Specifies the exact syntax of the language.

`scanner.mll`
: The lexical specification for the language, and reads tokens from the source file.

`Makefile`

Since the Ocaml project is compiled into interpreted bytecodes, a **`Makefile`** is needed, as is required in any C, C++, or Java project.

4. What to submit

Submit all of the necessary source files so that the grader may perform the build. That means submit **`Makefile`**, **`parser.mly`**, **`scanner.mll`**, and all **`*.mli`** and **`*.ml`** files. If you are doing pair programming, also submit the files required by the **`pair-programming`** document. Verify the grading criteria from the **`.score/`** subdirectory.