Programming Languages and Computation

Week 2: Primitive Operators

You should first download and read the short description of the Brischeme language and its interpreter, in the coursework brief: http://uob-coms20007.github.io/questions/cwk0.pdf. Then answer the following.

1. Write a Brischeme program to implement the *mod* function which takes two positive integers x and y and returns the remainder after dividing x by y, according to:

$$mod(x,y) = \begin{cases} x & \text{if } x < y \\ mod(x-y,y) & \text{otherwise} \end{cases}$$

and define the name *mod* for it. It will be useful to keep a copy of your definition in a text file. Then check that it behaves correctly on some examples:

```
> (mod 25 4)
1
> (mod 122 3)
2
> (mod 36 6)
0
```

Solution

```
(define mod (lambda (x y) (if (< x y) x (mod (- x y) y))))
```

2. Write a Brischeme program to implement the *gcd* function, which takes two positive integers x and y and returns their greatest common denominator, according to Euclid's algorithm:

$$gcd(x,y) = \begin{cases} x & \text{if } y = 0 \\ gcd(y, mod(x,y)) & \text{otherwise} \end{cases}$$

and define the name `gcd` for it. Check that your solution works:

```
> (gcd 20 30)
10
> (gcd 270 192)
6
> (gcd 1234 5243)
1
```

Solution

```
(define gcd (lambda (x y) (if (= y 0) x (gcd y (mod x y)))))
```

The REPL is defined by the function repl : store -> unit in the file *bin/main.ml*. It parses, evaluates and prints the expressions given to it by the user. It takes a store, e, as argument, which is the list of currently defined functions that are available in scope. When a user makes a new definition, using *define*, this will be added to the store.

3. Make sure you understand which part of the REPL source code is responsible for parsing and which is responsible for evaluating and printing. In this question you will extend Brischeme with a standard library.

   (a) Does parsing depend on the current store? Does evaluation and printing?

   (b) When the REPL is first initialised at the entry point of the program, what store is given?

   (c) According to the source code, which variable is used to hold the store that has been updated after parsing, evaluating and printing the user's input?

   (d) In the root directory of the development create a text file *prelude.bs* and put into it any definitions that you want to include in your standard library, for example *mod* and *gcd*.

   (e) In the entry point of the program, after printing Brischeme, read in the contents of the file *prelude.bs*, parse it and evaluate the forms (starting from the empty store) to create a new store. Finally, invoke the REPL with the new store.

   - The contents of the file, located in the directory where the interpreter is run, whose name is *foo*, can be obtained by the OCaml expression:

     ```
     In_channel.with_open_bin "foo" In_channel.input_all
     ```

     which returns a string. For more information, consult the documentation at https://ocaml.org/manual/5.2/api/In_channel.html.

Solution

(a) Parsing doesn't depend on the current store, but evaluation does. Taken in isolation, printing doesn't depend on the current store, but it is bundled in with the code for evaluation-and-printing ep_form, so in this sense it does depend on the current store.

(b) The empty store (list) [].

(c) e'

(d)
```
$ cat prelude.bs:
(define mod (lambda (x y) (if (< x y) x (mod (- x y) y))))
(define gcd (lambda (x y) (if (= y 0) x (gcd y (mod x y)))))
```

(e) The new entry point of the program:

```
(* This is effectively the entry point of the program. *)
let () =
  print_endline "Brischeme";
  (* Read in all the contents of file "prelude.bs" *)
  let prelude = In_channel.with_open_bin "prelude.bs" In_channel.input_all in
  (* Parse the contents to create a list of ASTs *)
  let forms = parse_prog prelude in
  (* Evaluate the prelude ASTs starting from the empty store *)
  let e = List.fold_left ep_form [] forms in
  (* Start the REPL from the store obtained by evaluating the prelude *)
  repl e
```

4. In this exercise, you will extend Brischeme so that the primitive addition operator can take any number of arguments and returns their sum.

```
> (+ 2 3 4 5 6)
20
> (+)
0
```

   (a) In the file *eval.ml*, after the function is_value, define a function sum_num_values of type sexp list -> int. This function should behave as follows, when given a list of Num AST nodes [Num n_1; Num n_2; ...; Num n_k] it should return the OCaml integer n_1 + n_2 + ... + n_k. Define the function recursively, by filling out the missing cases:

```
let rec sum_num_values (vs : sexp list) : int =
  match vs with
  | [] ->
  | (Num n) :: ws ->
  | _ -> failwith "Sexp in the list is not a number value."
```

   (b) Find the code for evaluating a call of the primitive operators, in the function step_sexp of the file lib/eval.ml. Replace the case for evaluating a call of the Plus operator, which currently requires exactly two number arguments, so that a list of arguments of any length can be provided instead.

Solution

(a)
```
let rec sum_num_values (vs: sexp list) : int =
  match vs with
  | [] -> 0
  | (Num n) :: ws -> n + (sum_num_values ws)
  | _ -> failwith "Non number value in list"
```

```
| Call (Plus, vs) -> Num (sum_num_values vs)
```