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
```

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:

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

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

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

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.

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 -> sexp. 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.

5. In this exercise, you will extend Brischeme with a new primitive operator for less-than-or-equal.

   (a) Add a new nullary constructor Leq to the type primops in the file *lib/ast.ml*.

   (b) Explain that this operator looks like "<=" as a string, for the purposes of printing (e.g. in debug messages), by adding an extra case to the function string_of_primop in file *lib/ast.ml*.

   (c) Find and understand the function lex_bool from the file *lib/lexer.ml*. Using this for inspiration, write a function lex_le_or_leq which, when invoked in a state where peek () = '<', returns either PrimOp Leq or PrimOp Less depending on whether the next character in stream is '=' or not. Replace the right-hand-side of the '<' case in the function lex_init by a call to this function.

   (d) Find the code for evaluating a call of the primitive operators, in the function step_sexp of the file lib/eval.ml. Add a case for the less-than-or-equal primitive operator, taking the existing code for the less-than operator as inspiration.

6. In this exercise, you will extend Brischeme with pairs (tuples of length 2). The idea is that pairs will be constructed using a new primitive binary operator *cons*, and new primitive unary operators *car* and *cdr* can be used to project out the first and second components, as in:

```
> (define x (cons 2 6))
> (car x)
2
> (cdr x)
6
```

   (a) Add new nullary constructors Cons, Car and Cdr to the type primops in the file *lib/ast.ml*, and explain what they look like as strings (for output purposes) in string_of_primop.

   (b) Modify the function lex_kw_or_id to return TkPrimOp Cons, TkPrimOp Car and TkPrimOp Cdr when the lexeme is "cons", "car" and "cdr" respectively.

   (c) Explain that Call (Cons, [v1; v2]) is already a value (and therefore does not make any execution step) whenever v1 and v2 are values, by adding a case Call (Cons, [v1; v2]) **when** is_value v1 && is_value v2 -> **true** to the function is_value of file *lib/eval.ml*. This will require that is_value is now defined as a recursive function, using **let rec**.

   (d) Explain how calls Call (Car, [Call (Cons, [v1; _])]) and Call (Cdr, [Call (Cons, [_; v2])]) are evaluated, by modifying the code for primitive operations in step_sexp.