Programming Languages and Computation

Week 4: Atoms and Lists

1. In this exercise you will extend the language to include Elixir/Erlang-style atom literals (which are a simple version of Scheme quoted symbols). The syntax of our atom literals is simply an identifier prefixed with an apostrophe, for example 'foo or 'x2.

```
> 'foo
'foo
> 'foo = 'bar
#f
> 'foo = 'foo
#t
```

Atoms are, as their name implies, atomic, that is, they have no internal structure as far as the programmer is concerned. The only thing you can do to an atom is to test it for equality with something else (typically another atom). Hence, you can think of an atom as a string that you can never take a apart, or a nullary datatype constructor (as in Haskell or OCaml).

The following refer to the lexer in *lib/lexer.ml*.

(a) Extend the lexer with a new kind of literal constructor called LAtom suitable for representing an atom literal.

(b) Explain how to output an atom literal as a string in string_of_lit.

(c) Write a function lex_atom which, when called in a state where peek() = '\'', consumes an atom literal and returns an appropriate token.

(d) Modify the lex_init function to dispatch to lex_atom when appropriate.

The following refer to the AST in *lib/ast.ml*.

(e) Extend the AST datatype sexp with a new constructor Atom **of** string.

(f) Explain an appropriate way to output an atom as a string in string_of_sexp.

The following refer to the parser in *lib/parser.ml*.

(g) Extend the eat_lit function to return an appropriate AST node when encountering an atom literal token.

The following refer to the evaluator in *lib/eval.ml*.

(h) Extend the is_value function to label atom literals as values (i.e. they are already fully evaluated).

Solution

In *lib/lexer.ml*:

```ocaml
type literal =
  ...
  | LAtom of string

let lex_atom () : token =
  (* assumes [peek () = '\''] *)
  drop ();
  let lexeme = ref "" in
  while is_more () && is_id_char (peek ()) do
    let c = peek () in
    drop ();
    lexeme := !lexeme ^ String.make 1 c
  done;
  TkLit (LAtom !lexeme)

let lex_init () : token =
  match peek () with
  ...
  | '\'' -> lex_atom ()

let string_of_lit l =
  match l with
  ...
  | LAtom s -> "'" ^ s
```

In *lib/ast.ml*:

```ocaml
and sexp =
  ...
  | Atom of string

and string_of_sexp (e:sexp) : string =
  match e with
  ...
  | Atom s -> "'" ^ s
```

In *lib/parser.ml*:

```ocaml
let eat_lit () : sexp =
  match peek () with
  ...
  | TkLit (LAtom s) -> drop (); Atom s
```

In *lib/eval.ml*:

```ocaml
let rec is_value (s:sexp) : bool =
  match s with
  | Num _ | Bool _ | Lambda _ | Atom _ -> true
  ...
```

2. In this exercise, you will extend Brischeme with lists, which can be done entirely within the standard library once pairs and atoms are available. The idea is that we will use the atom literal

'nil to represent the empty list, and a list with head *s* and tail *t* will be represented by a pair (cons s t). Consequently, the head of a given list can be extracted using car and the tail using cdr. For example:

```
> (define mylist (cons 2 (cons 3 (cons 4 'nil))))
> (car mylist)
2
> (cdr mylist)
(cons 3 (cons 4 'nil))
```

Add each of the following to your standard library.

(a) Define the Brischeme function *empty?* which returns *#t* if its single argument is *'()* and *#f* otherwise.

(b) Define the Brischeme function *map* which given a function *f* and a list *xs*, returns the list obtained from *xs* by applying *f* to every element.

```
> (map (lambda (x) (+ x 1)) (cons 2 (cons 3 (cons 4 '()))))
(cons 3 (cons 4 (cons 5 '())))
```

Solution

```
$ 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)))))
(define empty? (lambda (x) (= x '())))
(define map (lambda (f xs) (if (empty? xs) xs (cons (f (car xs)) (map f (cdr
```

3. Define a new primitive operator *list* which takes any number of arguments and returns the list containing exactly those arguments as its elements, in the same order.

```
> (list 2 3 4 5)
(cons 2 (cons 3 (cons 4 (cons 5 '()))))
> (list)
'()
```

Solution

In *lib/ast.ml*:

```
type primops =
  ...
  | List

...

let string_of_primop (p:primop) : string =
  match p with
```

```
...
| List -> "list"
```

At the end of lex_kw_or_id of *lib/lexer.ml*:

```
match !lexeme with
...
| "list"   -> TkPrimOp List
```

For evaluating the primitive operator call in step_sexp of *lib/eval.ml*:

```
| Call (List, vs) -> List.fold_right (fun v acc -> Call (Cons, [v; acc])) vs Nil
```

4. Using lists, pairs and atoms, it is straightforward to define a dictionary data structure.

    (a) Define the Brischeme function *lookup* which, given a key *k* list of pairs *xs*, returns the second component of the first pair in *xs* whose first component is equal to *k* if such a pair exists, and returns the atom 'not_found otherwise.

    ```
    > (define age (list (cons 'starmer 63) (cons 'jinping 72) (cons 'trump
    > (lookup 'trump age)
    79
    > (lookup 'modi age)
    'not_found
    ```

Solution

```
(define lookup
  (lambda (k xs)
    (if (empty? xs)
      'not_found
      (if (= (car (car xs)) k) (cdr (car xs)) (lookup k (cdr xs)))
    )
  )
)
```