B5 - Advanced Functional Programming

B-FUN-501

Bootstrap

An introduction to Lisp

1.1

EPITECH.

Bootstrap

language: haskell, scheme
compilation: via Makefile, including re, clean and fclean rules

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- Error messages have to be written on the error output, and the program should then exit with the 84 error code (0 if there is no error).

Part 0 - Install Chez-Scheme

Chez-Scheme is the reference for our interpreter, it's a good idea to have it at hand to test the conformity of our functions and interpreters.
You can install Chez-Scheme from here: `https://cisco.github.io/ChezScheme/#get`

Part 1 - S-Expressions

S-Expressions are at the core of Lisp. It's important to understand how they work, how to represent them as data structures in memory, and how to parse them.

It may be useful to do some reseach first. `https://en.wikipedia.org/wiki/S-expression` may be a good starting point.

In the following exercices, it's important to store the parsed input into a suitable data structure before printing it again. Merely transforming the input to match the output totaly defuse the purpose of the exercice.

The parsing library you've made for the bootstrap of the EvalExpr should be very usefull now.

Exercise 1

Write (in Haskell) a program which takes a S-Expression composed of dotted pairs, and print them as a conventional S-Expression.

Terminal

```
~/B-FUN-501> ./from_pairs "’(1 . ( 2 . ( 3 . ()))))"
(1 2 3)
~/B-FUN-501> ./from_pairs "’((a . b) . ((foo . bar) . ((5 . 6) . ()))))"
((a . b) (foo . bar) (5 . 6))
```

If you type these expressions in chez-scheme, it will give you the same answer

Exercise 2

Write (in Haskell) a program which takes an S-Expression, and print it as dotted pairs.

Terminal

```
~/B-FUN-501> ./to_pairs "'(1 2 3)"
(1 . (2 . (3 . ())))
~/B-FUN-501> ./to_pairs "'((a . b) (foo . bar) (5 . 6))"
((a . b) . ((foo . bar) . ((5 . 6) . ())))
```

Here is a Chez-Scheme program which does exactly that:

```
(define (to-pairs l)
  (cond ((atom? l) (display l))
        (#t (display "(")
            (to-pairs (car l))
            (display " . ")
            (to-pairs (cdr l))
            (display ")"))))
```

Terminal

```
~/B-FUN-501> scheme ./to_pairs.scm
> (to-pairs `(1 2 3))
(1 . (2 . (3 . ())))
> (to-pairs '((a . b) (foo . bar) (5 . 6)))
((a . b) . ((foo . bar) . ((5 . 6) . ())))
```

Part 2 - Let's write some Lisp...

To implement a Lisp interpreter it's important to have at least a basic understanding of the language and how it can be used.

Here you will be asked to write a couple of usefull functions using only the procedures and special forms included in the HAL interpreter:

- lambda, define, let
- quote, cond
- cons, car, cdr
- eq?, atom?
- +, -, *, div, mod, <

Your functions must work when tested with Chez-Scheme. They should work with your HAL too, once it's done.

Most of these functions already exists in Chez-Scheme, be carefull to actually test your function and not Chez-Scheme's implementation.

Once you have implemented a function, you're free to use it to implement the other ones.

You may need auxiliary functions to implement some of the requested functions.

Math: >, <=, >=, abs

Extend your arythmetic capabilities:

Terminal

```
> (>= (abs (- 3)) 2)
#t
```

Boolean logic: not, or, and

Terminal

```
> (and (or (not #f) #f) #t)
#t
```

List manipulation: null? length, append, reverse

Terminal

```
> (null? '(1 2 3))
#f
> (null? (cdr '(1)))
#t
> (length '(a b c d))
4
> (append '(1 2 3) '(4 5 6))
(1 2 3 4 5 6)
> (reverse '(1 2 3))
(3 2 1)
```

Contrary to Scheme's **append**, our version takes exactly 2 arguments.

Functional primitives: map, filter, fold-left, fold-right

These higher-order functions are all taking other functions as parameters to make repetitive tasks easier:

map takes a function and a list in arguments and returns a new list where each element is the result of the function applied to each element of the list in the same order.

Terminal

```
> (map (lambda (x) (* x 2)) '(1 2 3 4 5))
(2 4 6 8 10)
> (map reverse '((1 2) (3 4) (5 6)))
((2 1) (4 3) (6 5))
```

filter takes a predicate (a function returning a boolean value) as first argument, and returns a list of the elements of the list for which the predicated has returned true.

```
> (filter (lambda (x) (eq? (mod x 2) 1)) '(1 2 3 4 5))
(1 3 5)
```

fold-left and **fold-right** take a function which "accumulate" values into a result, and take an initial value and a list as argument. **fold-left** does it from left to right, **fold-right** from right to left

```
> (fold-left (lambda (acc x) (- acc x)) 0 `(1 2 3))
-6
> (fold-right (lambda (x acc) (- x acc)) 0'(1 2 3))
2
> (fold-left + 0 '(1 2 3 4 5))
15
```

OPTIONAL

- **fact** : takes an integer as argument, returns it's factorial: (fact 10) -> 3628800
- **fib** : takes an integer as argument, returns the Nth number in the Fibonnacci sequence: (fib 30) -> 832040
- **sort** : takes a list of interger, returns this list sorted (merge-sort, quick-sort and tree-sort are good candidates...)
- **fizzbuzz**: the classic exercise of fizzbuzz

Part 3 - A recommanded read

A good starting point to think about how to implement a Lisp interpreter is to read Paul Graham's article Roots of Lisp: `http://languagelog.ldc.upenn.edu/myl/llog/jmc.pdf`

Be carefull, the dialect of Lisp used in this article has lots of difference with our HAL