B5 - Advanced Functional Programming

B-FUN-501

HAL

An HAskell Lisp interpreter

1.1

HAL

binary name: hal
language: haskell
compilation: via Makefile, including re, clean and fclean rules
build tool: stack wrapped in a Makefile (see below)

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.
- 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).

The goal of this project is to implement an interpreter for a minimalist dialect of LISP in Haskell.

Our dialect of Lisp is functionnal (almost no side effects or mutable states), and a subset of Scheme. Therefore, an expression evaluated by your interpreter must give the same result as the same expression evaluated by a Scheme interpreter (the reference implementation being Chez-Scheme)

Invocation

Your interpreter **must** take a list of files as command line arguments and interpret them sequentially. Symbols defined in a file must be kept defined in subsequent files.

Terminal

```
~/B-FUN-501> cat foo.scm
(define foo 21)
~/B-FUN-501> cat bar.scm
(* foo 2)
~/B-FUN-501> ./hal foo.scm bar.scm
42
```

when more than one expression is evaluated in a file or multiple files, only the very last one must be printed on the standard output

Your interpreter **should** support an interactive mode (REPL). For convenience most of the examples in this subject uses this mode. See **REPL** section bellow for details.

An REPL is very useful to rapidly test your interpreter.

Error handling

You must stop the exectution as soon as an error occurs and return a 84 status code. You're free to display any meaninful information on the standard output or error output.

```
Terminal
~/B-FUN-501> ./hal error.scm
*** ERROR : variable foo is not bound.
~/B-FUN-501> echo $?
84
```

Types

Your interpreter must support the following types:

- Signed integers (64 bits or more)
- Symbols (unique identifiers)
- Lists as linked lists of cons cells, an empty list being represented by "**'()**"

You may want to add more types for internal use, or as bonuses

You may have missed it so let say it again: you should take the time to read https://en.wikipedia.org/wiki/Cons

Builtins

Here are the procedures you must implement in your interpreter as builtins. If in doubt they must perform as in Chez-Scheme. You are free to implement other builtins as long as they don't conflict with Chez-Scheme default library.

CONS

Takes two arguments, **cons**truct a new list cell with the first argument in the first place (car) and the second argument is the second place (cdr).

Terminal

```
> (cons 1 2)
(1 . 2)
> (cons 1 (cons 2 (cons 3 '())))
(1 2 3)
```

CAR

Takes a cons as argument, returns its first element (the car).

Terminal

```
> (car (cons 1 2))
1
```

CDR

Takes a cons as argument, returns its second element (the cdr).

Terminal

```
> (cdr (cons 1 2))
2
> (cdr '(1 2 3))
(2 3)
```

" ' " is syntactic sugar for the **quote** special form, documented bellow

EQ?

Returns "**#t**" if its first and second arguments are equal. Lists are never equals, except for the empty list. Symbols equals to themselves only. Intergers behave as expected.

Terminal

```
> (eq? 1 1)
#t
> (eq? (+ 1 1) 2)
#t
> (eq? 'foo (car '(foo bar)))
#t
> (eq? 'foo 'bar)
#f
> (eq? '() '())
#t
```

ATOM?

Returns "**#t**" if it's first argument is atomic, that is if it's not a non-empty list.

Terminal

```
> (atom? 'foo)
#t
> (atom? '(1 2 3))
#f
> (atom? '())
#t
```

ARITHMETICS BUILTINS

you also have to implement "**+**", "**-**", "*****", "**div**", "**mod**" and "**<**".

"**+**", "**-**", "*****" take a variable number of arguments, while **div**, **mod** and **<** take exactly two arguments.

Terminal

```
> (div (* 5 2) (- 3))
-3
> (< (* 2 2) 5)
#t
> (mod (+ 5 5) 3)
1
```

SPECIAL FORMS

Special forms are expressions where the arguments are not necessarily evaluated all the time (contrary to the case of a regular procedure call).

QUOTE

Takes one argument, returns it without evaluating it.

Terminal

```
> (quote toto)
toto
> (quote (+ 1 2))
(+ 1 2)
```

As syntactic sugar, quote can also be noted as a leading " ' " character:

Terminal

```
> 'toto
toto
>'(+ 1 2)
(+ 1 2)
```

LAMBDA

Takes a list of parameters as first argument, and an expresion to evaluate as second argument, returns a lambda (procedure) which can be subsequently called.

Terminal

```
> (lambda (a b) (+ a b))
#<procedure>
> ((lambda (a b) (+ a b)) 1 2)
3
```

DEFINE

If it's first argument is a symbol, associate the symbol to its second argument, and returns it's name.
If it's first argument is a list, defines a function which name is the first elemnt of the list, the rest of the list its parameters, and the second argument the function's body.

alternatively, it's acceptable to mimic Chez-Scheme behavior and to return / display nothing.

```
> foo
*** ERROR : variable foo is not bound.
> (define foo 42)
foo
> foo
42
> (define add (lambda (a b) (+ a b)))
add
> (add 1 3)
3
> (define (sub a b) (- a b))
sub
> (sub 3 1)
2
```

The second form of define is easily expressed as a rewrite of the expression using lambda: (define (name arg1 arg2 ...) body) => (define name (lambda (arg1 arg2 ...) body))

You only have to support **define** when placed at the top level. Supporting **define** inside functions and lambdas (like Chez-Scheme) is considered a bonus.

LET

Takes a list of key/values as first argument, and an expression as a second argument, evaluate this second argument within an environement where the key / value pairs are bound.

```
> (let ((a 2) (b (+ 1 2))) (+ a b))
5
```

Let can also be expressed as a rewrite involving lambda: (let ((n1 v1) (n2 v2) (n3 v3) ...) body) => ((lambda (n1 n2 n3) body) v1 v2 v3)

COND

Allows to conditionally evaluate expressions. It takes a variable number of arguments. Each argument is a list. “**cond**” successively evaluates the first element of each list. If its return value is true, it evaluates the second element of the list and returns it’s value. Otherwise, it tries the next expression.

Terminal

```
> (cond (#f 1) (#t (+ 1 1)))
2
> (cond ((eq? 'foo (car '(foo bar))) 'here) ((eq? 1 2) 'there) (#t 'nope))
here
```

Contrary to Chez-Scheme's behavior, your interpreter can consider as invalid a cond which doesn't have a catch-all condition in last position (**#t**). Both behaviors are accepted.

REPL

Your interprer should implement a REPL (as all LISPs do).
In this case, if invoked without arguments your interpreter must launch the REPL. Additionally, the REPL can be launched using **"-i"** as argument, in conjunction with files.

```
Terminal
~/B-FUN-501> cat foo.scm
(define foo 21)
~/B-FUN-501> ./hal foo.scm -i
> (* foo 2)
42
>
```

You can simply use getLine, or you're allowed to use the package **haskeline**

If an error occurs in interactive mode, you may give control back to the user instead of terminating the execution, or enter a debug mode.

EXAMPLES

Your lisp interpreter should be able to process the following programs:

FACTORIAL (FACT.LISP)

```
(define (fact x)
  (cond ((eq? x 1) 1)
        (#t (* x (fact (- x 1))))))
```

```
Terminal
~/B-FUN-501> hal fact.scm -i
> (fact 10)
3628800
```

FIND THE N'TH NUMBER IN THE FIBONACCI SEQUENCE (FIB.LISP)

```
(define (fib x)
  (cond ((eq? x 0) 0)
        ((eq? x 1) 1)
        (#t (+ (fib (- x 1)) (fib (- x 2))))))
```

```
~/B-FUN-501> hal fib.scm -i
> (fib 21)
10946
```

Merge sort (sort.scm)

```
(define (null? l) (eq? l '()))

(define (merge-lists l1 l2)
  (cond ((null? l1) l2)
        ((null? l2) l1)
        ((< (car l1) (car l2)) (cons (car l1) (merge-lists (cdr l1) l2)))
        (#t                    (cons (car l2) (merge-lists l1 (cdr l2))))))

(define (split-half l l1 l2)
  (cond ((null? l) (cons l1 l2))
        ((null? (cdr l)) (split-half (cdr l) (cons (car l) l1) l2))
        (#t (split-half (cdr (cdr l))
                        (cons (car l) l1)
                        (cons (car (cdr l)) l2)))))

(define (merge-sort lst)
  (cond ((null? lst) '())
        ((null? (cdr lst)) lst)
        (#t (let ((lsts (split-half lst '() '())))
              (merge-lists (merge-sort (car lsts))
                           (merge-sort (cdr lsts)))))))
```

```
~/B-FUN-501> hal sort.scm -i
> (merge-sort '(39 16 22 24 17 29 18 26 27 3 34 25 10 6 7 12 8 30 2 21 13 36 14 38
32 41 40 4 35 19 5 33 23 9 15 31 28 20 42 37 11 1))
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42)
```

Bonus

Writting a substancial (50 lines or more) program in Lisp executable by your interpreter is a nice bonus to have. Other examples of interesting bonuses include:

- Extend your parser (for example: to support square brackets like Scheme).
- Add more special forms (like **if**...)
- A debugger.
- Add more types (strings, floats, rationals (try (/ 10 3) in Scheme)...) and builtins to manipulate them.
- Builtins to interact with the file system.
- Bindings for a graphical library (SFML?), or a network API maybe?
- Special forms to implement Lisp macros.
- Tail call recurtion optimisation.
- Anything cool you may think of...

Stack

Stack is a convenient build tool/package manager for Haskell.
Its use is required for this project, with **version 2.1.3 at least**.

It wraps a build tool, either **Cabal** or **hpack**.
You are required to use the hpack variant (package.yaml file in your project, autogenerated .cabal file).

> This is what stack generates by default with `stack new`.

Stack is based on a package repository, **stackage**, that provides consistent snapshots of packages.
The version you use must be in the **LTS 18** series (`resolver: 'lts-18.10'` in stack.yaml).

> In `stack.yaml`, extra-dependencies cannot be used.

base, **haskeline**, **containers** and **mtl** are the only dependencies allowed in the `lib` and `executable` sections of your project (package.yaml).
There is no restriction on the dependencies of the `tests` sections.

> You must provide a **Makefile** that builds your stack project (i.e. it should at some point call 'stack build').

'stack build' puts your executable in a directory that is **system-dependent**, which you may want to copy.
A useful command to learn this path in a system-**independent** way is:
`stack path --local-install-root`.

Of course using any parsing library is forbidden, even if present in
base