HAPPY-)

```
(fun (main) (write (fact (read))))
(fun (fact x) (
    (if (= x 0) (return 1)  (return (* x (fact (- x 1)))))
))
```

La syntaxe du HAPPY-) est simple et épurée à l'image du LISP dont notre langage est inspiré. Tout est une fonction (ou une méthode) dans ce langage sauf le while et le if : le write renvoie l'objet qu'il vient d'écrire, le set la valeur qu'on vient d'assignée (ex (write (set a 5))). Mais rien n'oblige le programmeur à retourner quelquechose dans les fonctions qu'il écrit lui même. Les caractères autorisés pour définir les identifiants dépassent les simples caractère alphanumérique, on peut donc définir de nouveaux opérateurs pour nos données tel que -> ou : := ou encore «« voir :-> (la fonction happy). Mais puisqu'un exemple vaut mieux qu'un long discours, voici un programme travaillant avec une pile.

```
(fun (main) (
    (set :: (newStack 1))
    (>>> 7 (>>> 6 (>>> 5 (>>> 4 (>>> 3 (>>> 2 ::))))))
    (write (sum ::))
))
(fun (sum stack) (
    (set sum 0)
    (while ((set val (<<< stack))) ((set sum (+ sum val)))
    (return sum)
))
(fun (newStack x) (
    (set stack (new 1)) (set node (newNode)) (node:= x)
    (stack.@= (newNode))
    (retrun stack)
))
(fun (<<< stack) (
    (set node (stack.first))
    (if (node)
        ((stack.@= (node.next)) (return node.val))
        ((return node))
    )
))
(fun (>>> x stack) (
    (set node (newNode))
    (node.:= x)
    (node.-> (stack.first))
    (stack.@= node)
    (return stack)
))
(method.1 (first) (return this.1))
(method.1 (@= x) ((set this.1 x) (return this)))
(method.2 (next) (return this.1))
(method.2 (val) (return this.2))
(method.2 (-> x) ((set this.1 x) (return this)))
(method.2 (:= x) ((set this.2 x) (return this)))
(fun (newNode) (return (new 2)))
```