Guía de estudio
Control 1

P1. Considere un lenguaje correspondiente a un subconjunto de la lógica proposicional. Los programas válidos en este lenguajes están descritos con la siguiente BNF:

```
#| <formula> ::=
    | (Bool <Bool>)
    | (id <sym>)
    | (^ <formula> <formula>)
    | (v <formula> <formula>)
    | (with <sym> <formula> <formula>)
|#
```

corresponden a:

- Constantes (True,False)
- Proposiciones no interpretadas (A,B,C, ...)
- Conectivos lógicos conjunción ($\wedge$) y disyunción ($\vee$)
- `with` para asociar un id con una expresión lógica y poder hacer referencia a el en el cuerpo.

a) Escriba utilizando **`deftype`** el tipo inductivo `Formula` que genera los AST de los programas de este lenguaje.

b) Escriba utilizando **`deftype`** el tipo `FValue` que representa los valores de este lenguaje.

c) Escriba un parser que tome una *expresión estructurada* bien formada y retorne un `AST Formula`. Para ello:

d) Escriba la gramática BNF de las `s-expr` que representan la sintaxis concreta del lenguaje.

e) Escriba el parser

```
;; parse :: s-expr -> Formula
```

Ejemplos de entradas válidas a parsear:

```
> (parse 'True)
> (parse '(v A False))
> (parse '(^ C (v A B)))
> (parse '(with (A True) (^ A A)))
```

Escriba un interprete para este lenguaje utilizando substitución diferida y asegurando alcance léxico de sus identificadores. Para ello:

- **e.1** Defina el ambiente en el que sustituirá sus variables. Su *tipo de dato abstracto* debe seguir la siguiente interfaz.

```
#|    ----------------------------
Environment abstract data type

empty-env  ::  Env
extend-env  ::  Sym LValue Env -> Env
env-lookup  ::  Sym Env -> FValue

representation BNF:
<env> ::= (mtEnv)
        | (aEnv <id> <FValue> <env>)
|#
```

- **e.2** Defina la función `subst :: sym x Formula x Formula -> Formula` para hacer el reemplazo de identificadores

Escriba un interprete para el lenguaje.

```
;; interp  ::  Formula Env -> FValue
```

Ejemplos de entradas válidas a interpretar:

```
> (interp (parse 'True) empty-env)
 (BoolV #t)
> (interp (parse '(^ A False)) empty-env)
env-lookup: Variable A no definida
> (interp (parse '(^ C (v A B))) empty-env)
env-lookup: "Variable B no definida"
> (interp (parse '(with (A True) (^ A False))) empty-env)
(BoolV #f)
```

P2. a) Implemente en Racket la función `filterTR` tal que sea recursiva por la cola.

b) ¿Qué ventaja tiene la definición de `filterTR` que acaba de implementar con respecto a una que no sea recursiva por la cola?

c) Considere una versión recursiva por la cola de `filter` en Java. ¿Qué sucede con la ventaja del punto anterior?

P3. Disponemos de un dispositivo liviano cuyo procesador es basado en una pila y que solamente soporta las siguientes instrucciones:

- `push num`: pone el número `num` en la pila
- `add`: adiciona los dos números encima de la pila
- `sub`: subtrae los dos números encima de la pila

`add` y `sum` consumen los números y ponen el resultado en la pila.

Definan en PLAI Scheme un compilador (parser + generador de código) del lenguaje AE al procesador basado en pila. El output del compilador es la lista (plana) de instrucciones a ejecutar por el procesador. Por ejemplo:

```
> (compile '(+ 3 5))
(push 3 push 5 add)
```