Tarea 2

Interpretes y Haskell

(*interp* (*parse* "*Helloworld*"))

Para la resolución de la tarea recuerde que

- Toda función debe estar acompañada de su firma, una breve descripción coloquial (en `P2.rkt`, `P3.rkt`) y un conjunto significativo de tests (en `test.rkt`).
- Todo datatype definido por el usuario (via `deftype`) debe estar acompañado de una breve descripción coloquial y de la gramática BNF que lo genera.

Si la función o el datatype no cumple con estas reglas, será ignorado.

Ejercicio 1 **18 Pt**

Responda las siguientes preguntas conceptuales acerca de los tipos de *scope*:

(a) [4 Pt] Explique brevemente los conceptos de *scope* estático y *scope* dinámico.

(b) [4 Pt] Explique por qué la gran mayoría de los lenguajes adoptan el *scope* estático por defecto. Además, indique motivos por los cuales en ciertas circunstancias es conveniente tener adicionalmente mecanismos de alcance dinámico.

(c) [7 Pt] Dado el siguiente programa:

```
(let ([x 3] [y 5] [z 7])
  (let ([f (λ (x) (+ x y))])
    (let ([g (λ (y) (+ y (f x) z))])
      (let ([x 17] [z 8])
        (g z)))))
```

¿Qué resultado produce con *scope* estático? ¿Y con *scope* dinámico? Justifique cada caso.

(d) [3 Pt] ¿Qué mecanismo se usa en los lenguajes para preservar el *scope* estático ante la presencia de funciones de primera clase?

Ejercicio 2 **16 Pt**

En este ejercicio consideraremos un lenguaje que corresponde a un subconjunto de la lógica proposicional. Los programas de este lenguaje siguen la siguiente sintaxis abstracta:

```
#| <logic> ::=
    | (bool <bool>)
    | (id <id>)
    | (band <logic> <logic>)
    | (bor <logic> <logic>)
    | (with <id> <logic> <logic>)
|#
```

Estas reglas corresponden a:

- Constantes (True, False)
- Proposiciones no interpretadas (A, B, C, ...)
- Conjunción y disyunción de otras dos subexpresiones
- Definiciones locales

(a) [2 Pt] Declare el tipo inductivo `Logic` que representa la sintaxis abstracta del lenguaje arriba descrita.

(b) [4 Pt] Escriba la gramática BNF que define la sintaxis concreta del lenguaje y escriba el parser correspondiente:

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

Para definir la sintaxis concreta, básese en los siguientes ejemplos:

```
> (parse 'True)
(bool #t)
> (parse '(A v False))
(bor (id 'A) (bool #f))
> (parse '(C ^ (A v B)))
(band (id 'C) (bor (id 'A) (id 'B)))
> (parse '(with (A True) (A ^ A)))
(with 'A (bool #t) (band (id 'A) (id 'A)))
```

(c) [1 Pt] Declare el tipo `LValue` que representa los valores de este lenguaje.

(d) [9 Pt] Escriba un interprete para el lenguaje utilizando substitución diferida:

```
interp :: Logic Env -> LValue
```

Para ello defina el ambiente en el que sustituirá identificadores. 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 -> LValue

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

Ejemplos de entradas válidas a interpretar:

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

Ejercicio 3 **16 Pt**

En este ejercicio consideraremos una variación del lenguaje visto en clases, que soporta números complejos (recuerde que un número complejo se expresa mediante dos números reales, parte real e imaginaria).

La sintaxis abstracta del lenguaje está dada por la siguiente gramática:

```
#|
<expr> ::= (real <num>)
         | (comp <num> <num>)
         | (add <expr> <expr>)
         | (id <id>)
         | (fun <sym> <expr>)
         | (app <expr> <expr>)
|#
```

(a) [1 Pt] Defina el tipo de datos recursivo `Expr` que representa la sintaxis abstracta del lenguaje.

(b) [4 Pt] Implemente el parser para este lenguaje, considerando los siguientes ejemplos de sintaxis concreta:

`8`	Número real.
`(+ 1 (2)i)`	Numero Complejo.
`(+ 3 6)`	Adición.
`'x`	Identificador.
`(fun (x) (+ 2 x))`	Función de primera clase.
`(f (+ 2 3))`	Aplicación de función.
`((+ y 5) where y = (+ 2 4))`	Definición local.

Las definiciones locales son azúcar sintáctico, y se representarán como una aplicación de función.
Escriba, además, la gramática (en forma de BNF) de la sintaxis concreta considerada.

(c) [4 Pt] Defina el datatype `Value` para que existan 3 valores posibles en este lenguaje (construidos a partir de los constructores `realV`, `compV` y `closureV`). Además, defina la función:

```
;; num+ :: Value Value -> Value
```

que se encargue de manejar las sumas.

Importante: Note que el operador de suma puede recibir cualquier combinación de números reales o complejos, donde la suma de un número complejo con otro cualquiera es siempre un complejo, y la suma de dos números reales es un real.

(d) [7 Pt] Construya un intérprete:

```
;; eval:: Expr Env -> Value
```

el cual debe implementar alcance estático. Adicionalmente, implemente la función:

```
;; run:: s-expr -> Value
```

que resuelva el *pipeline* completo.

```
> (run '((f (+ 2 (+ 1 (2)i))) where f = (fun (x) (+ x x))))
(compV 6 4)
```

Ejercicio 4 **10 Pt**

Los números de Stirling se definen como la cantidad de formas de particionar un conjunto de m elementos en k subconjuntos no vacíos. A continuación se muestra el triángulo de valores que forman los números de Stirling:

(m,k)	(0)	(1)	(2)	(3)	(4)	...
(0)	1					
(1)	0	1				
(2)	0	1	1			
(3)	0	1	3	1		
(4)	0	1	7	6	1	
...	...					

Los números de Stirling obedecen la siguiente relación de recurrencia:

$$
\begin{aligned}
S(0,0) &= 1 \\
S(m,0) &= 0, \qquad m > 0 \\
S(m,k) &= S(m-1,k-1) + k * S(m-1,k)
\end{aligned}
$$

Defina en Haskell la lista infinita

```
triangulo_stirling :: [[Integer]]
```

de manera que el elemento n-ésimo de la lista sea el nivel n del triángulo de Stirling. Por ejemplo:

```
*Main> (take 4 triangulo_stirling)
[[1],[0,1],[0,1,1],[0,1,3,1]]
```