Auxiliar 5

Sustitución y funciones de primer orden

P1. *Free-ocurrences:* El archivo `base_aux5.rkt` contiene una definición inicial del tipo de datos recursivo `Expr` con constantes numéricas, adición, resta, if0 y un with. Además contiene su parser e interpretación con sustitución directa.

Programe la función **free-vars** que recibe una expresion del lenguaje y retorna una lista de los simbolos que son free-ocurrences/identificadores libres en la expresión.

P2. *With*:* En el lenguaje actual el programador solo puede hacer un binding en un with, es decir, solo puede añadir un solo identificador con su valor para ser usado en el cuerpo, la única forma de tener múltiples bindings es haciendo withs concatenados, pero es bastante verboso para el programador.

Así que el programador usara un **with*** que soporta **n** bindings y usted deberá progamar la función **unrolling-with*** que reciba una sintaxis concreta del with* y la pase a una concatenación de with también en sintaxis concreta, es decir, usted programará un syntactic-sugar.

P3. *Funciones n-arias de primer orden:* Considere que tiene el lenguaje visto en clases (código en `base_aux5_p3.rkt`).

```
<expr> ::= (num <num>)
         | (add <expr> <expr>)
         | (sub <expr> <expr>)
         | ( if0  <expr> <expr> <expr>)
         | (with <id> <expr> <expr>)
         | (id <sym>)
         | (app <sym> <expr>)

<fundef> ::= (fundef <sym> <sym> <expr>)
```

Extienda el lenguaje para que sea posible:

a) Definir funciones con n argumentos (0, 1, 2, etc). e invocar funciones que reciban varios argumentos. Considere los siguientes ejemplos:

```
> (run '{const5}
       ( list  (fundef 'const5 (  list  ) (parse `5))))
5
> (run '{add 1 2 3 4}
       ( list  (fundef 'add ( list  'e1 'e2 'e3 'e4) (parse `(+ e1 (+ e2
       (+ e3 e4)))))))
10
```

b) Agregar nuevas funciones en medio de la ejecución del programa, considere el siguiente ejemplo:

```
> (run    {with-fun {define add1 x {+ 1 x}}
    {add1 5}})
6
```