**Universidad Nacional Autónoma de México**
**Facultad de Ciencias**
**Lenguajes de Programación**

Práctica 6

**Karla Ramírez Pulido** karla@ciencias.unam.mx
**J. Ricardo Rodríguez Abreu** ricardo_rodab@ciencias.unam.mx
**Manuel Soto Romero** manu@ciencias.unam.mx

**Fecha de inicio:** 14 de abril de 2018
**Fecha de término:** 27 de abril de 2018
Semestre 2018-2

# Objetivo

Agregar listas, asignaciones locales recursivas y nuevos operadores al lenguaje implementado en la Práctica 5, así como repasar el concepto de recursividad y su implementación mediante el uso de cajas[1] y ambientes recursivos.

# Antecedentes

En las sesiones previas de laboratorio se revisó el concepto de recursividad y se realizaron actividades para implementar un pequeño intérprete para una versión simplificada del lenguaje de programación `RCFWAE` que implementa recursividad. Se recomienda revisar dicha actividad junto con las Notas 5 del curso.

# Repositorio

El material necesario para completar esta práctica se encuentra en el repositorio de *GitHub Classroom* del curso: `https://classroom.github.com/g/mj1HwdNf`.

# Desarrollo de la práctica

La gramática en EBNF para las expresiones del lenguaje `RCFWBAEL` (*Recursive, Conditionals, Functions, With, Booleans, Arithmetic Expressions and Lists)* que se implementa en esta práctica es la siguiente:

[1] *Boxes*

```
<expr> ::= <id>
         | <num>
         | <bool>
         | <list>
         | {<op> <expr>+}
         | {if <expr> <expr> <expr>}
         | {cond {<expr> <expr>}+ {else <expr>}}
         | {with {{<id> <expr>+}
         | {with* {{<id> <expr>}+} <expr>}
         | {rec {{<id> <expr>}+} <expr>}
         | {fun {<id>*} <expr>}
         | {<expr> <expr>*}

 <id> ::= a | ... | z | A | Z | aa | ab | ab | ... | aaa | ...
        (Cualquier combinación de caracteres alfanuméricos
         con al menos uno alfabético)

 <num> ::= ... | -1 | 0 | 1 | 2 | ...

<bool> ::= true | false

 <list> ::= empty
          | {list <expr>+}

 <op> ::= + | - | * | / | % | min | max | pow | sqrt | inc | dec
        | < | <= | = | /= | > | >= | zero?
        | not | and | or
        | head | tail | append | empty?
```

En equipos de **tres integrantes** se deben completar las funciones faltantes de los archivos `grammars.rkt`, `parser.rkt`, `desugar.rkt` e `interp.rkt` hasta que logren pasar todas las pruebas unitarias que se incluyen en el archivo `pruebas_practica6.rkt` y se ejecute correctamente el archivo `pratica6.rkt`[2].

**Ejercicio 6.1 (1 pt.)** Completar el cuerpo de la función `(parse sexp)` del archivo `parser.rkt` que realiza el análisis sintáctico correspondiente, es decir, construye expresiones del TDA `RCFWBAEL/L` incluido en el archivo `grammars.rkt`.

```
;; parse: s-expression -> RCFWBAEL/L
(define (parse sexp) ...)
```

```
> (parse '{+ 1 2})
(opS + (list (numS 1) (numS 2))
```

[2]Para tener derecho a calificación, los archivos deben ejecutarse sin errores.

**Ejercicio 6.2 (1 pt.)** Completar el cuerpo de la función `(desugar expr)` del archivo `desugar.rkt` que elimina azúcar sintáctica[3] de las expresiones de `RCFWBAEL/L`, es decir, las convierte en expresiones del TDA `RCFBAEL/L` incluido en el archivo `grammars.rkt`[4].

```
;; desugar: RCFWBAEL/L -> RCFBAEL/L
(define (desugar expr) ...)
```

```
> (desugar (parse '{with {{a 3}} {+ a 4}}))
(app (fun '(a) (op + (id 'a) (num 4))) (list (num 3)))
```

**Ejercicio 6.3 (8 pts.)** Completar el cuerpo de la función `(interp expr env)` del archivo `interp.rkt` que realiza el análisis semántico correspondiente, es decir, evalúa expresiones de `RCFBAEL/L`. Para evaluar las asignaciones locales recursivas, se debe modificar la función `cyclically-bind-and-interp` vista en clase para que procese una lista de identificadores en lugar de uno solo.

Además de las especificaciones que se tomaron en cuenta para resolver la Práctica 5, se deben considerar ahora, los siguientes puntos para implementar la función `interp`:

- Las listas se evalúan a valores de tipo `listV`. Ejemplo:

  ```
  > (interp (desugar (parse '{list 1 2 3})) (mtSub))
  (listV (list (numV 1) (numV 2) (numV 3)))
  ```

- En esta práctica se agregan nuevos operadores:

  | | |
  |---|---|
  | `zero?` | Indica si un número es cero. |
  | `head` | Obtiene el primer elemento de una lista. |
  | `tail` | Obtiene el resto de una lista. |
  | `append` | Concatena dos listas. |
  | `empty?` | Indica si una lista es vacía. |
  | `inc` | Incrementa el valor de un número en 1. |
  | `dec` | Decrementa el valor de un número en 1. |

  ```
  > (interp (desugar (parse '{zero? 10})) (mtSub))
  (boolV #f)
  ```

- Las expresiones `rec` presentan un comportamiento parecido al de la primitiva `with*`, sin embargo, estas expresiones permiten definir identificadores recursivos, es decir, que se definen en términos de sí mismos. Por ejemplo `{rec {{fac {fun{n} {if {zero? n} 1 {* n {fac {dec n}}}}}} {n 5}} {fac n}}`. Se interpreta similar a `with*`, la diferencia es que se usan ambientes recursivos que hacen uso de cajas para almacenar el cuerpo de la función. Ejemplo:

[3] Tipo de sintaxis que hace que un programa sea más "dulce" o fácil de escribir.
[4] Para esta práctica no se agregan nuevas expresiones endulzadas.

```
> (define expr
   '{rec {
      {fac {fun {n}
         {if {zero? n}
             1
             {* n {fac {dec n}}}}}}
      {n 5}}
      {fac n}})
> (interp (desugar (parse expr)) (mtSub))
(numV 120)
```

```
;; interp: RCFBAEL/L -> RCFBAEL/L-Value
 (define (interp expr env) ...)
```

```
> (interp (op + (list (num 1) (num 2))))
3
```

# Referencias

Algunas referencias de consulta:


[1] Karla Ramírez, Manuel Soto, *Notas de laboratorio del curso de Lenguajes de Programación*, Semestre 2018-2, Facultad de Ciencias, UNAM. Disponibles en: [http://lenguajesfc.com/notas.html].

[2] Rodrigo Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.

[3] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, Brown University, 2007.