# Lenguajes de Programación

## Práctica 6

Semestre 2022-1
Facultad de Ciencias, UNAM


| | | |
|---|---|---|
| **Profesora** | Karla Ramírez Pulido | **Fecha de inicio:** 12 de Enero de 2022 |
| **Ayud. Lab** | Silvia Díaz Gómez | **Fecha de entrega:** 19 de Enero de 2022 |
| **Ayud. Lab** | Fhernanda Montserrat Romo Olea | |


## Descripción

La práctica consiste en implementar un intérprete sencillo para el lenguaje **RCFWBAEL**. Para esto, se debe completar el cuerpo de las funciones faltantes dentro de los archivos **grammars.rkt**, **parser.rkt**, **desugar.rkt** e **interp.rkt** hasta que pasen las pruebas unitarias incluidas en el archivo **test-practica6.rkt** y se ejecute correctamente el archivo **practica6.rkt**.

La gramática del lenguaje **RCFWBAEL** se presenta a continuación:

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

<id> ::= a | b | c | ...

<num> ::= 1 | 2 | 3 | ...

<bool> ::= true | false

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

<op> ::= + | - | * | / | modulo | expt | add1 | sub1
         | < | <= | = | > | >= | not | and | or | zero?
         | empty? | car | cdr
```

## Ejercicios

1. (2 pts.) Completar el cuerpo de la función **`(parse sexp)`** dentro del archivo **`parser.rkt`** el cual recibe una expresión simbólica[1], realiza el análisis sintáctico correspondiente es decir, construir el Árbol de Sintaxis Abstracta[2] (ASA).

```
;; Definicion del tipo SBinding.
(define-type SBinding
[sbinding (id symbol?) (value SAST?)])

;; Definicion del tipo Condition.
(define-type Condition
[condition (test-expr SAST?) (then-expr SAST?)]
[else-cond (else-expr SAST?)])

;; Definicion del tipo SAST.
(define-type SAST
[idS    (i symbol?)]
[numS   (n number?)]
[boolS  (b boolean?)]
[listS  (elems (listof SAST?))]
[opS    (f procedure?) (args (listof SAST?))]
[ifS    (test-expr SAST?) (then-expr SAST?) (else-expr SAST?)]
[condS  (cases (listof Condition?))]
[withS  (bindings (listof sbinding?)) (body SAST?)]
[withS* (bindings (listof sbinding?)) (body SAST?)]
[recS   (bindings (listof sbinding?)) (body SAST?)]
[funS   (params (listof symbol?)) (body SAST?)]
[appS   (fun-expr SAST?) (args (listof SAST?))])

;; parse: s-expression → AST
(define (parse sexp) ...)
```

2. (4 pts.) Completar el cuerpo de la función **`(desugar expr)`** dentro del archivo **`desugar.rkt`**. Esta función recibe una expresión en forma de Árbol de Sintaxis Abstracta Endulzado[3], eliminando el azúcar sintáctica de las expresiones correspondientes. Y regresará un nuevo árbol. Las expresiones con azúcar sintáctica son:

   - **Asignaciones locales**

     Las expresiones **`with`** se convierten en aplicaciones de función. Expresiones de la forma:

[1] Del inglés, *s-expression*. Puede ser un número, un símbolo o una lista de expresiones simbólicas.
[2] Del inglés, *Abstract Syntax Tree (AST)*.
[3] *Sugared Abstract Syntax Tree*.

```
{with {{<id> <value>}}
  <body>}
```

se transforma en una aplicación de función respectivamente. El **<id>** y el **<body>** del **with** forman el parámetro y cuerpo de la función, mientras que el campo **<value>** representa el argumento a aplicar.

```
{{fun {<id>} <body>} <value>}
```

- **Asignaciones locales anidadas**

  Las expresiones **with*** se convierten en expresiones **with** anidadas. Expresiones de la forma:

```
{with* {{<id> <value>} {<id> <value} ...}
   <body>}
```

  las cuales se transforman en expresiones **with** anidadas de tal forma que cada pareja de identificador y valor **{<id> <value>}** representa una nueva asignación local.

```
{with {{<id> <value>}}
   {with {{<id> <value>}}
         ...
         <body>}...}
```

- **Asignaciones locales recursivas**

  Las expresiones **rec** se transforman en expresiones **with** anidadas donde a cada valor (**<value>**) se le aplica el combinador **Y**. Expresiones de la forma:

```
{rec {{<id> <value>} {<id> <value} ...}
   <body>}
```

  se transforman en:

```
{with {{<id> {fun {<id>} {Y <value>}}}}
    {with {{<id> {fun {<id>} {Y <value>}}}}
          ...
          <body>}...}
```

- **Funciones**

  Las expresiones de tipo función **fun** con $n$ parámetros se deberán transformar en funciones con un único parámetro, es decir, expresiones currificadas. Si se tiene una expresión:

```
{fun {<param1> <param2> ...} <body>}
```

  se deberá transformar en una expresión **fun** anidada de tal forma que cada función regrese una nueva función como resultado.

```
{fun {<param1>} {fun {<param2>} ... <body>} ...}
```

- **Aplicaciones de función**
  Dado que las aplicaciones de función reciben $n$ argumentos se deberán transformar en aplicaciones de función con un único argumento, tomando en cuenta que la asociación se hace a la izquierda. Además el orden de asociación de los argumentos es de izquierda a derecha. Por lo tanto, las expresiones de la forma:

  ```
  {<fun-expr> <arg1> <arg2> ...}
  ```

  se transformarán en aplicaciones de función argumento por argumento.

  ```
  {{<fun-expr> <arg1>} <arg2>} ...}...}
  ```

- **Condicionales**
  Las expresiones **cond** deberán transformarse a expresiones **if** anidadas. Si se tiene una expresión:

  ```
  {cond {<test-expr> <then-expr>} ... {else <else-expr>}}
  ```

  se transformará en una expresión anidada **if**. Es importante observar que la parte **else** es obligatoria.

  ```
  {if <test-expr> <then-expr> {if ... <else-expr>}...}
  ```

```
;; Definicion del tipo Binding.
(define-type Binding
   [binding (id symbol?) (value AST?)])

;; Definicion del tipo AST.
(define-type AST
   [id   (i symbol?)]
   [num  (n number?)]
   [bool (b boolean?)]
   [listT (elems (listof AST?))]
   [op   (f procedure?) (args (listof AST?))]
   [iF   (test-expr AST?) (then-expr AST?) (else-expr AST?)]
   [fun  (param symbol?) (body AST?)]
   [app  (fun-expr AST?) (arg AST?)])

;; desugar: SAST → AST
(define (desugar expr) ...)
```

3. (4 pts.) Completar el cuerpo de la función **(interp expr env)** dentro del archivo **interp.rkt** que dada una expresión, regresa la evaluación correspondiente. Para la evaluación de expresiones debe usarse alcance estático y evaluación perezosa. Tomar en consideración:

- **Identificadores**

  Se debe lanzar un error indicando que se trata de una variable libre.

  ```
  (interp (desugar (parse 'foo)) (mtSub)) => error: Variable libre
  ```

- **Números**

  Al ser un valor atómico, los números se evalúan así mismos.

  ```
  (interp (desugar (parse 1729)) (mtSub)) => (numV 1729)
  ```

- **Booleanos**

  Al ser un valor atómico, las constantes lógicas se evalúan así mismas.

  ```
  (interp (desugar (parse 'true)) (mtSub)) => (boolV #t)
  ```

- **Listas**

  Al ser un valor atómico, las listas se evalúan a sí mismas, evaluando cada uno de sus elementos.

  ```
  (interp (desugar (parse '{list 1 {+ 2 3}})) (mtSub)) => (listV '(1 4))
  ```

- **Operaciones**

  Se debe aplicar el operador correspondiente a la lista de operandos indicada.

  ```
  ;; Operaciones unarias
  (interp (desugar (parse '{add1 18})) (mtSub))     => (numV 19)
  (interp (desugar (parse '{sub1 35})) (mtSub))     => (numV 34)
  (interp (desugar (parse '{not true})) (mtSub))    => (boolV #f)
  (interp (desugar (parse '{zero? 1})) (mtSub))     => (boolV #f)

  ;; Operaciones binarias
  (interp (desugar (parse '{modulo 10 2})) (mtSub)) => (numV 0)
  (interp (desugar (parse '{expt 2 3})) (mtSub))    => (numV 8)

  ;; Operaciones n-arias
  (interp (desugar (parse '{+ 1 2 3})) (mtSub))     => (numV 6)
  (interp (desugar (parse '{- 3 2 1})) (mtSub))     => (numV 0)
  (interp (desugar (parse '{* 1 2 3})) (mtSub))     => (numV 6)
  (interp (desugar (parse '{/ 8 2 2})) (mtSub))     => (numV 2)
  (interp (desugar (parse '{< 1 2 3})) (mtSub))     => (boolV #t)
  (interp (desugar (parse '{<= 1 2 3})) (mtSub))    => (boolV #t)
  (interp (desugar (parse '{> 1 2 3})) (mtSub))     => (boolV #f)
  (interp (desugar (parse '{>= 1 2 3})) (mtSub))    => (boolV #f)
  (interp (desugar (parse '{= 1 2 3})) (mtSub))     => (boolV #f)
  ```

```
(interp (desugar (parse '{and true true false}))
      (mtSub))                                    => (boolV #f)
(interp (desugar (parse '{or true true false}))
      (mtSub))                                    => (boolV #t)
```

- **Condicionales**

  Debe evaluarse la condición y en caso de regresar un valor de verdadero entonces debe interpretarse la expresión *then*, en caso contrario (cuando la condición se evalúa a la constante falso) se debe entonces interpretar la expresión de la rama *else*.

  ```
  (interp (desugar (parse '{if true 1 2})) (mtSub)) => (numV 1)
  ```

- **Funciones**

  Cuando una expresión es de tipo función, se genera una cerradura (*closure*) la cual almacena el ambiente donde fue definida la función, lo cual permite implementar alcance estático.

  ```
  (interp (desugar (parse '{fun {x} x}))) =>
        (closureV 'x (id 'x) (mtSub))
  ```

- **Aplicación de función**

  Cuando se tienen aplicaciones de función deberás aplicar la sustitución del parámetro formal por el parámetro real en el cuerpo de la misma. Para esto debe buscarse el valor de cada variable en el ambiente correspondiente.

  ```
  (interp (desugar (parse '{{fun {x} x} 2})) (mtSub)) => (numV 2)
  ```

Al usar el tipo de evaluación perezosa, se deben aplicar los puntos estrictos correspondientes mediante la función **`strict`** previamente definida:

- Argumentos de las operaciones aritméticas y lógicas.
- Condición de la primitiva **`if`**.
- La función en una aplicación.

Adicionalmente se debe completar el cuerpo de la función **`call-interp`** que dada una expresión, llama a la función **`interp`** con el ambiente inicial correspondiente.

El ambiente inicial para esta práctica debe contener la definición del Combinador de punto fijo $Y$. La definición del Combinador $Y$ (en el Cálculo Lambda) es la siguiente:

$$Y =_{def} \lambda f.\,(\lambda x.f\,(xx))\,(\lambda x.f\,(xx))$$

```
;; interp: AST → AST-Value
(define (interp expr) ...)
```

# Referencias

[1] Shriram Krishamurthi, *Programming Languages: Application and Interpretation,* First Edition, Brown University, 2007.