

# Lenguajes de Programación

## Proyecto final: Parte 1

Karla Ramírez Pulido

Manuel Soto Romero

Alejandro Hernández Mora

Silvia Díaz Gómez

Pedro Ulises Cervantes González

Semestre 2021-2

Facultad de Ciencias, UNAM

**Fecha de inicio: 20 de julio de 2021**

**Fecha de entrega: 13 de agosto de 2021**

## 1. Objetivos

Implementar el parser e intérprete para el lenguaje **CFWBAE**. Para llevar a cabo esta tarea, se debe completar el cuerpo de las funciones faltantes dentro de los archivos **grammars.rkt**, **parser.rkt**, **desugar.rkt** e **interp.rkt**<sup>1</sup>.

La gramática del lenguaje CFWBAE se presenta a continuación:

```
<expr> ::= <id>
          | <num>
          | <bool>
          | <char>
          | <string>
          | <list>
          | {<op> <expr>+}
          | {if <expr> <expr> <expr>}
          | {if0 <expr> <expr> <expr>}
          | {cond {<expr> <expr>}+ {else <expr>}}
          | {with {{<id> <expr>}+} <expr>}
          | {with* {{<id> <expr>}+} <expr>}
          | {fun {<id>+} <expr>}
          | {<expr>}{<expr>*}

<id>   ::= a | b | c | ...
<num>  ::= 1 | 2 | 3 | ...
<bool> ::= true | false
<char> ::= 'a' | 'b' | 'c' | ...
<string> ::= "a" | "aa" | "ab" | ...
<list>  ::= empty | {lst <expr>*}
```

---

<sup>1</sup>Puedes encontrar el esqueleto de estos archivos en la especificación del proyecto en classroom. También puedes tomar como base los archivos de los ejercicios de programación vistos previamente.

```

<op>    ::= + | - | * | / | modulo | expt | add1 | sub1
        | < | <= | = | > | >= | not | and | or | zero?
        | num? | bool? | char? | string? | list?
        | cons | car | cdr | append | length | empty?
        | string-append | string-length

```

En *Racket*, se define la gramática anterior por medio de los siguientes tipos abstractos de datos.

---

```

;; Definición del tipo Binding
(define-type Binding
  [binding (id symbol?) (value SCFWBAE?)])  
  

;; Definición del tipo condition para la definición de cond.
(define-type Condition
  [condition (test-expr SCFWBAE?) (then-expr SCFWBAE?)]
  [else-cond (else-expr SCFWBAE?)])  
  

;; Definición del tipo SCFWBAEL
(define-type SCFWBAE
  [idS   (i symbol?)]
  [numS  (n number?)]
  [boolS (b boolean?)]
  [charS (c char?)]
  [stringS (s string?)]
  [listS (l (listof SCFWBAE?))]
  [iFS   (test-expr SCFWBAE?) (then SCFWBAE?) (else SCFWBAE?)]
  [iF0   (test-expr SCFWBAE?) (then SCFWBAE?) (else SCFWBAE?)]
  [opS   (f procedure?) (args (listof SCFWBAE?))]
  [conds (cases (listof Condition?))]
  [withS (bindings (listof binding?)) (body SCFWBAE?)]
  [withS* (bindings (listof binding?)) (body SCFWBAE?)]
  [funS   (params (listof symbol?)) (body SCFWBAE?)]
  [appS   (fun SCFWBAE?) (args (listof SCFWBAE?))])

```

---

## 2. Ejercicios

### 2.1. Parser.rkt

1. (3 pts.) Completar el cuerpo de la función (parse SEXP) dentro del archivo parser.rkt. La cual recibe una expresión simbólica<sup>2</sup> y realiza el análisis sintáctico correspondiente. Esto es, construye un Árbol de Sintaxis Abstracta<sup>3</sup> (ASA). Para el análisis sintáctico de las operaciones aritméticas se debe hacer un mapeo entre los símbolos de función en sintaxis concreta y las funciones nativas de Racket.

---

```
;; parse: s-expression -> CFWBAE
(define (parse SEXP) ...)
```

---

#### Aridad:

El parser verifica que la aridad de las funciones y las instrucciones del lenguaje sea la correcta:

- a) Para los predicados<sup>4</sup>, las funciones and, sub1, add1, not, length, car, cdr y string-length la aridad debe ser 1.
- b) Para las funciones modulo, expt y cons la aridad debe ser 2.
- c) Para el resto de funciones la aridad es arbitraria, pero mayor a 1.

#### Funciones and, or:

Las funciones and y or no están definidas como procedimientos en Racket, por lo que lo recomendable es definir tus propias funciones. Te recomendamos los nombres and y or.

### 2.2. Desugar.rkt

2. (3.0 pts) Completar el cuerpo de la función (desugar SEXP) dentro del archivo desugar.rkt.

---

```
;; desugar SCFWBAE-> CFWBAE
(define (desugar SEXP) ...)
```

---

El azúcar sintáctica sirve para hacer algunas expresiones del lenguaje más fáciles de leer para el programador, ayudando a entender mejor algunas instrucciones o reduciendo el código escrito. Sin embargo las expresiones no cambian la semántica ni la funcionalidad de la instrucción modificada.

La función desugar toma una expresión del tipo **SCFWBAE**, que es la sintaxis abstracta intermedia del lenguaje con azúcar sintáctica<sup>5</sup> y la transforma en una expresión del tipo **CFWBAE**; que es la sintaxis abstracta final del lenguaje, es decir, sin azúcar sintáctica.

La sintaxis abstracta final deja de considerar las expresiones de tipo **if0**, **cond**, **with** y **with\***. Dicha sintaxis se define mediante los siguientes tipos abstractos de datos: data-type CFWBAE

---

<sup>2</sup>Del inglés, *s-expression*. Puede ser un número, un símbolo o una lista de expresiones simbólicas.

<sup>3</sup>Del inglés, *Abstract Syntax Tree (AST)*.

<sup>4</sup>Las funciones cuyo nombre termina con ?

<sup>5</sup>La *S* viene del inglés *sugar*, que significa azúcar.

---

```
;; Definición del tipo CFWBAEL
(define-type CFWBAE
  [id   (i symbol?)]
  [num  (n number?)]
  [bool (b boolean?)]
  [chaR (c char?)]
  [strinG (s string?)]
  [lisT (l (listof CFWBAE?))]
  [iF   (condicion CFWBAE?) (then CFWBAE?) (else CFWBAE?)]
  [op   (f procedure?) (args (listof CFWBAE?))]
  [fun  (params (listof symbol?)) (body CFWBAE?)]
  [app  (fun CFWBAE?) (args (listof CFWBAE?))])
```

---

La función **desugar** considera el manejo de las siguientes expresiones como se indica a continuación.

- **Condicional *if0*:**

La expresión (*if0 num then-expr else-expr*) es una expresión con azúcar sintáctica para la instrucción (*if (zero? num) then-expr else-expr*). Por lo que la instrucción es eliminada de la sintaxis abstracta y sustituida por una condicional simple. Por ejemplo:

```
> (desugar (parse '{if0 3 4 5}))
(iF (op #<procedure:zero?> (list (num 3))) (num 4) (num 5))
```

- **Condicionales (*cond*):**

Recordemos que la expresión condicional es azúcar sintáctica para evitar escribir expresiones *iF* anidadas, por lo que en esta función se elimina el azúcar sintáctica. Por Ejemplo:

```
>(desugar (parse '{cond {#t 1} {#f 2} {else 3}}))
(iF (bool #t) (num 1) (iF (bool #f) (num 2) (num 3)))
>(interp (desugar (parse '{cond {#t 1} {#f 2} {else 3}})) (mtSub))
(numV 1)
>(interp (desugar (parse '{cond {#f 1} {#f 2} {else 3}})) (mtSub))
(numV 3)
> (interp (desugar (parse '{cond {#f 1} {#t 2} {else 3}})) (mtSub))
(numV 2)
```

- **Asignaciones locales simples (*with*):**

Las expresiones del tipo *with* se transforman en una aplicación de función. Por ejemplo:

```
'{with {[x 2]
          [y 1]{}}
    {- 3 y x}}
```

Se transforma en:

```
(app
  (fun '(x y)
    (op #<procedure:-> (list (num 3) (id 'y) (id 'x))))
  (list (num 2) (num 1)))
```

- **Asignaciones locales anidadas (*with\**):**

Como ya mencionamos antes, ***with\**** también deja de formar parte de la sintaxis abstracta. Por lo que la función desugar se encarga de transformar las instancias de ***with\**** en instancias anidadas de ***with***, que posteriormente son eliminadas como se menciona anteriormente. Por ejemplo, la expresión.

```
'(with* ([x 1]
          [y 2])
        (+ y x))
```

se transforma en<sup>6</sup>:

```
'(with ([x 1])
      (with ([y 2])
        (+ y x)))
```

y a su vez esto se transforma en:

```
'(app
  (fun '(x)
    (app
      (fun '(y)
        (op #<procedure:+> (list (id 'y) (id 'x))))
      (list (num 2))))
    (list (num 1)))
```

## 2.3. Interp.rkt

### 2.3.1. Ambientes

Deberás implementar ambientes con evaluación de alcance estático para nuestro lenguaje de programación. Para esto, deberás utilizar un caché de sustitución diferida. En el caché de sustituciones se guardan los ambientes en los que son interpretadas las variables que sean utilizadas. Un ambiente es una lista que almacena parejas de variables (identificadores que son símbolos) con su respectivo valor que debe de tener. Los valores que almacena el caché de sustituciones a cada identificador corresponden al resultado de la evaluación de una expresión del lenguaje<sup>7</sup>.

---

```
;;Data-type que representa la sintaxis abstracta de CFWBAE-Value
(define-type CFWBAE-Value
  [closure (param (listof symbol?)) (body CFWBAE?) (env DefrdSub?)]
  [numV   (n number?)]
  [boolV  (b boolean?)]
  [charV  (c char?)]
  [stringV (s string?)]
  [listV   (l (listof CFWBAE-Value?))])
```

---

<sup>6</sup>Es opcional implementar esto en el desugar, podemos hacer el cambio directo desde el parser aunque puede ser más truculento.

<sup>7</sup>Por esta razón es que se llama así el data-type CFBWAE-Value.

Como se menciona anteriormente, el caché de sustituciones es una lista de parejas de variables y valores, pero además es una estructura recursiva. El caché puede ser vacío<sup>8</sup> o estar conformado por la tripleta id, valor y ds, que es otro caché de sustituciones<sup>9</sup>. El caché de sustituciones se define con el siguiente tipo abstracto de dato, data-type DefrdSub.

---

```
;; Data-type que representa un caché de sustituciones
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value CFWBAE-Value?) (ds DefrdSub?)])
```

---

3. (1 pts.) Completar el cuerpo de la función (lookup name ds) dentro del archivo interp.rkt la cual busca el nombre de la variable name dentro del caché de sustitución ds, Regresa el valor correspondiente o arroja un error en caso de que no encuentre el identificador.

```
;; lookup: symbol DefrdSub -> CFWBAE
(define (lookup name ds) ...)
```

---

4. (3 pts.) Completar el cuerpo de la función (interp expr ds) dentro del archivo interp.rkt; el cual recibe una expresión, un caché de sustituciones y regresa la evaluación correspondiente, de acuerdo con las variables definidas en el caché.

```
;; interp: CFWBAE DefrdSub-> CFWBAE-Value
(define (interp expr ds))
```

---

Las funciones en este lenguaje son de primera clase, lo que significa que pueden ser pasadas como parámetros a otras funciones, ser almacenadas como valores y ser devueltas como resultado de un procedimiento. Por esta razón los valores del lenguaje consideran el tipo closure, que contiene los valores necesarios para instanciar una función. Un closure contiene esencialmente la definición de una función ya interpretada; esto comprende a los argumentos de la función, el cuerpo de la función y el ambiente donde se fue evaluada<sup>10</sup>.

Algunos ejemplos de los resultados esperados de la interpretación para algunas expresiones son:

```
> (interp (desugar( parse '3)) (mtSub))
(numV 3)
> (interp (desugar( parse #t)) (mtSub))
(boolV #t)
> (interp (desugar( parse 'x)) (mtSub))
Error lookup: Variable libre: 'x
> (interp (desugar (parse '{cond {#t 1} {#f 2} {else 3}})) (mtSub))
(numV 1)
> (interp (desugar (parse '{cond {#f 1} {#t 2} {else 3}})) (mtSub))
(numV 2)
```

<sup>8</sup>Caso en el que el caché está representado por mtSub.

<sup>9</sup>Este caso está representado por el constructor aSub.

<sup>10</sup>Esto es necesario para poder preservar la evaluación estática. De otro modo los valores de las variables involucradas en la definición de la función pueden cambiar después de la definición de ésta y alterar la semántica (o contexto) en el que fue definida.

```
> (interp (desugar (parse '{cond {#t 1} {#t 2} {else 3}})) (mtSub))
  (numV 1)
>(interp (desugar (parse '{cond {#f 1} {#f 2} {else 3}})) (mtSub))
  (numV 3)
> (interp (desugar (parse '(+ 1 1 1 (- 3 4 1) (sub1 1)))) (mtSub))
  (numV 1)
>(interp (desugar (parse '{string-append "ho" "la"})) (mtSub))
  (stringV "hola")
> (interp (desugar (parse '{cons 3 (lst 1 2)})) (mtSub))
  (listV (list (numV 3) (numV 1) (numV 2)))
> (interp (desugar (parse '{append (lst 1 2) (lst 1 2 3)})) (mtSub))
  (listV (list (numV 1) (numV 2) (numV 1) (numV 2) (numV 3)))
> (interp (desugar (parse '{length (lst 1 2)})) (mtSub))
  (numV 2)
```

### 3. Requerimientos

Se deben subir los archivos requeridos a la plataforma Google Classroom antes de las 23:59 hrs. del día de la fecha de entrega, conforme lo como lo indican los lineamientos de entrega. No olvides incluir el archivo ReadMe.txt con los datos de tus compañeros de equipo. Sólo es necesario que una persona suba la práctica, los demás compañeros deberán subir como tarea el contenido del archivo ReadMe.txt, de manera que se pueda ver el equipo que integran desde la plataforma google classroom.

El orden en el que aparezcan las funciones en los archivos solicitados, debe ser el orden especificado en este archivo PDF, de lo contrario podrán penalizarse algunos puntos.

La idea es que reutilices funciones de los ejercicios que ya se programaron anteriormente.

¡Que tengas éxito en tu proyecto!