Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación
Práctica 5
Semestre 2026-1
22 de Octubre de 2025

Profesora:

- Dra. Karla Ramírez Pulido

Ayudantes:

- Alan Alexis Martínez López

Ayudante de laboratorio:

- Karyme Ivette Azpeitia García

Integrantes del equipo:

- Gonzalez Castillo Patricio Salvador - 321142391
- Valencia Pérez Guillermo Emanuel - 321018689
- Rubio Resendiz Marco Antonio - 320209763
- Sautto Ramirez Seldon - 321084163

1 Objetivos

Analizar las implementaciones dadas en los archivos `grammars.rkt`, `parser.rkt`, `desugar.rkt` e `interp.rkt` para el lenguaje CFWBAE y realizar los siguientes ejercicios.

1.1 Gramática del lenguaje CFWBAE

La gramática del lenguaje CFWBAE se presenta a continuación:

```
<expr> ::= <id>
         | <num>
         | <bool>
         | {<op> <expr>+}
         | {if <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
<op> ::= + | - | * | / | modulo | expt | add1 | sub1
       | < | <= | = | > | >= | not | and | or | zero?
```

1.2 Definición de tipos en Racket

La gramática anterior se define mediante los siguientes tipos en Racket:

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

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

;; Definicion del tipo SCFWBAE
(define-type SCFWBAE
  [idS (i symbol?)]
  [numS (n number?)]
  [boolS (b boolean?)]
  [iFS (condicion 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

1. **(0.5 pts.) ¿Por qué es importante representar el programa como un árbol de sintaxis abstracta?**

2. **(1 pts.) Explica cómo el parser reconoce una expresión `if` y la diferencia respecto al antiguo `if0`.**

3. **(0.5 pts.) De acuerdo a la implementación en el archivo `parser.rkt` ¿Qué ocurre si el parser recibe una expresión con paréntesis mal colocados o una forma desconocida?**

2.2 Desugar

4. **(1 pts.) ¿Qué significa "desazucar" una expresión? Da un ejemplo.**

   Desazucar (desugar) significa transformar construcciones sintácticas convenientes (azúcar sintáctica) en construcciones más básicas del lenguaje núcleo. Es como traducir notación cómoda a su forma fundamental.

   Ejemplo: El `with` es azúcar sintáctica que se traduce a una aplicación de función:

```
; Azucar sintactica (SCFWBAE):
{with {{x 5}} {+ x 2}}

; Desazucarado (CFWBAE):
{{fun {x} {+ x 2}} 5}
```

   En `desugar.rkt`:

```
[withS (bindings body)
  (app (fun ids (desugar body)) vals)]
```

5. **(1 pts.) ¿Cómo traduce `cond` una cascada de condiciones en una serie de `if` anidados? ¿Qué ocurre si no hay un `else` final?**

   El `cond` se traduce recursivamente: cada caso se convierte en un `if` cuya rama `else` contiene el siguiente `cond`. El caso `else` final es la base de la recursión.

   Ejemplo conceptual:

```
; Original:
{cond
  {{< x 0} -1}
  {{= x 0} 0}
  {else 1}}

; Se desazucara a:
{if {< x 0}
    -1
    {if {= x 0} 0
        1}}
```

Si no hay `else` final, el parser debe generar un error (como se ve en `parse-cond`), porque el `cond` quedaría incompleto y no sabríamos qué devolver cuando todas las condiciones fallan.

6. **(1 pts.) ¿Qué pasaría si olvidas aplicar `desugar` recursivamente dentro de las subexpresiones (por ejemplo, en el cuerpo de un `with`)?**

   Las subexpresiones quedarían sin desazucar, causando que el intérprete reciba construcciones de SCFWBAE en lugar de CFWBAE, provocando errores de tipo.

   Ejemplo del problema:

```
; Si desugar fuera:
[withS (bindings body)
  (app (fun ids body))] ; ERROR! body no esta desazucarado

{with {{x 5}} {with {{y 3}} {+ x y}}}; Internamente  with {{y 3}} ...
                                     ; no se traduce
```

   El interprete fallaria al no reconocer withS, pues no es del lenguaje CFBAE, por eso necesitamos: `(desugar body)` y `(map desugar args)` en todas las subexpresiones.

7. **(1 pts.) Si agregas un nuevo azúcar como `when`, ¿qué regla de traducción definirías en `desugar.rkt`?**

   El `when` ejecuta una expresión solo si la condición es verdadera, sin rama `else` explícita. Se traduce a un `if` con el valor `false` como rama else:

```
; En grammars.rkt agregar:
[whenS (test SCFWBAE?) (body SCFWBAE?)]

; En parser.rkt:
[(list 'when test body)
 (whenS (parse test) (parse body))]

; En desugar.rkt:
[whenS (test body)
  (if (desugar test)
      (desugar body)
      (bool #f))]
```

   Uso: `{when {> x 0} {+ x 1}}` se traduce a `{if {> x 0} {+ x 1} false}`

8. **(1 pts.) ¿Por qué `desugar` siempre debe producir una expresión válida del lenguaje base (CFWBAE)?**

   Porque el intérprete solo entiende CFWBAE, no SCFWBAE. La separación en dos capas (sintaxis concreta → sintaxis abstracta) simplifica el intérprete: no necesita conocer todas las variantes sintácticas, solo las construcciones fundamentales. Sería como si un compilador que traduce un lenguaje a lenguaje maquina generara instrucciones inválidas

   Si `desugar` produjera algo inválido:

   - El intérprete no sabría cómo evaluar la expresión
   - Los `type-case` en `interp.rkt` fallarían

Analogía: Es como un compilador que traduce de lenguaje alto nivel a lenguaje máquina; si la traducción genera instrucciones inválidas, el procesador no puede ejecutarlas.

2.3 Interp

9. **(1 pts.) ¿Qué papel cumple el ambiente (`DefrdSub`) en la evaluación de expresiones con variables?**
   Se trata de la memoria que requere cada aplicación de *interp* para evaluar con variables ligadas. De otro modo no es posible pasar de *CFWBAE* a *CFWBAE-Value*.

   Por ejemplo, para la siguiente expresión.

```
{with {x 5}
        {with {y 3}
                {+ x y}}}
```

   El *parser* hace lo siguiente:

```
(withS (list (binding 'x (numS 5)))
        (withS (list (binding 'y (numS 3)))
                (opS + (list (idS 'x) (idS 'y)))))
```

   Y mediante *desugar* hace lo siguiente:

```
(app (fun '(x) (app (fun '(y) (op + (list (id 'x) (id 'y))))
        (list (num 3))))
        (list (num 5)))
```

   Al final, se crean los ambientes necesarios para evaluar la función:

```
env0 = (mtSub)
env1 = (aSub 'x (numV 5) env0)
env2 = (aSub 'y (numV 3) env1)
```

   Lo cual nos permite evaluar la función mediante *lookup*:

```
(lookup 'x env2) -> (numV 5)
(lookup 'y env2) -> (numV 3)
(op + (list (id 'x) (id 'y))) == (+ x y) -> (+ 5 3) -> 8
```

10. **(1 pts.) ¿Qué diferencias encuentras entre evaluar `with` directamente y evaluar la versión desazucarada (como `app(fun ...)`)?**

    (a) **Ambientes**: El `with` utiliza un ambiente en la evaluación de funciones mientras la versión desazucarada crea una cerradura antes de crear el ambiente de la función. Se diría que `with` ocupa menos pasos y menos memoria que la versión desazucarada.

```
(aSub 'x (numV 5) env-actual)                    ;with
(aSub 'x (numV 5) (closure-env closure))         ;app(fun)
```

    (b) **Anidamiento**: El `with` puede manejar multiples asignaciones y crear un único ambiente para las mismas, mientras que la versión desazucarada, al crear una cerradura por función, tiene que crear multiples ambientes.

```
{with {x 1 y 2} {+ x y}}                    ;ambiente extendido.
{{fun {x} {{fun {y} {+ x y}} 2}} 1}         ;ambientes anidados.
```

(c) **Mantenimiento**: La versión desazucarada requiere menos constructores, por lo que es más fácil de mantener.

11. **(1 pts.) De acuerdo a la implementación dada en el archivo `interp.rkt` ¿Qué estrategia de evaluación usa el intérprete: estricta o perezosa? Justifica tu respuesta.**
Notemos los siguientes fragmentos del código:

- Líneas 24-26:

```
[op (f args)
        (let ([arg-vals (map (lambda (a) (interp a env)) args)])
```

- Líneas 41-43...52:

```
[app (fun-expr arg-exprs)
        (let ([fun-val (interp fun-expr env)]
                [arg-vals (map (lambda (a) (interp a env))
                   arg-exprs)])
                ...
                        (interp (closure-body fun-val) new-env))
```

En las observaciones anteriorees, el uso de `(map(lambda (a)...))` evalua todos los argumentos y operandos antes de aplicar la función o ejecutar una operación. Lo cual es congruente con la definición de *Evaluación Estricta*.

2.4 Extra (Opcionales)

12. **(1 pts.) Propón una extensión del lenguaje donde `with` acepte funciones anónimas múltiples (como parámetros simultáneos). Escribe la traducción en forma de desazucar.**
Habría que agregar una nueva construcción al tipo SCFWBAE:

```
[withFunS (bindings (listof binding?)) (body SCFWBAE?)]
```

withFunS se desazucara de manera similar a with, por lo que habría que agregar a desugar.rkt un nuevo caso:

```
[withFunS (bindings body)
   (let ([ids (map binding-id bindings)]
         [vals (map (lambda (b) (desugar (binding-value b))) bindings
            )])
     (app (fun ids (desugar body)) vals))]
```

13. **(0.5 pts.) Explica ¿Qué cambios serían necesarios en el `interp` si se decidiera eliminar el módulo `desugar` y manejar los azúcares directamente?**
El interp toma expresiones azúcaradas, las interpreta como sugars y luego las desazucara

en desugar, al eliminar desugar habría que cambiar interp para que trate directamente con expresiones SCFWBAE y no CFWBAE.

```
(type-case CFWBAE expr ...
```

y agregar todos los casos de constructores azucarados:
Primero el with

```
[withS (bindings body)
  (let* ([ids (map binding-id bindings)]
         [vals (map (lambda (b) (interp (binding-value b) env))
            bindings)]
         [new-env (foldr (lambda (id val acc) (aSub id val acc)) env
            ids vals)])
    (interp body new-env))]
```

with*:

```
[withS* (bindings body)
  (if (null? bindings)
      (interp body env)
      (let* ([first (first bindings)]
             [new-env (aSub (binding-id first)
                            (interp (binding-value first) env)
                            env)])
        (interp (withS* (rest bindings) body) new-env)))]
```

cond:

```
[condS (cases)
  (interp-cond cases env)]

(define (interp-cond cases env)
  (match cases
    [(list) (error 'interp "cond␣sin␣casos")]
    [(list (else-cond e)) (interp (else-expr e) env)]
    [(list (condition test then) rest ...)
     (let ([test-val (interp (test-expr test) env)])
       (if (and (boolV? test-val) (boolV-b test-val))
           (interp (then-expr then) env)
           (interp-cond rest env)))]))
```

Y el withFun anteriormente agregado:

```
[withFunS (bindings body)
  (let* ([ids (map binding-id bindings)]
         [vals (map (lambda (b) (interp (binding-value b) env))
            bindings)]
         [new-env (foldr (lambda (id val acc) (aSub id val acc)) env
            ids vals)])
    (interp body new-env))]
```