Integrantes:

Nicolas Rodriguez Romero # 2266071

Jhon Alexis Ruiz Quiceno # 2266014

Michael Rodriguez Arana # 2266193

Pruebas con racketunit:

En las pruebas tenemos listas con elementos múltiples, el caso de lista vacía, caso de lista anidada y hasta una lista con una sublista y elementos

```
(define lista_anidada
  (scan&parse
    "cons(cons(1 cons(2 empty)) cons(3 empty))"
  )
)
(define expected_lista_anidada
  '((1 2) 3)
)
(check-equal? (evaluar-programa lista_anidada) expected_lista_anidada)

; Lista con sublista y elementos
(define lista_mixta
  (scan&parse
    "cons(1 cons(cons(2 cons(3 empty)) cons(4 empty)))"
  )
)
(define expected_lista_mixta
  '(1 (2 3) 4)
)
(check-equal? (evaluar-programa lista_mixta) expected_lista_mixta)
```

```
EMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   COMMENTS

\Users\ruizq\Desktop\taller2flp> racket  c:/Users/ruizq/Desktop/taller2flp/pruebas.rkt
\Users\ruizq\Desktop\taller2flp>
```

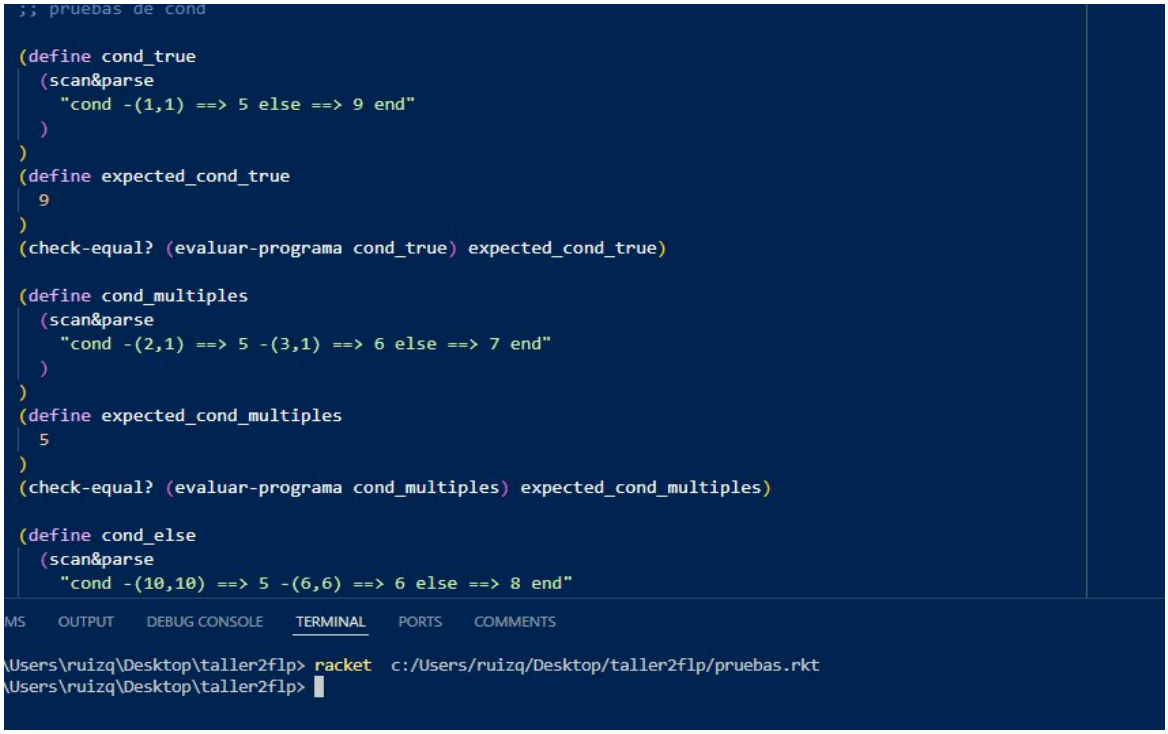
No muestra ningún error, si se cambia el esperado fallará:

```
32   (define lista_anidada
33     (scan&parse
34       "cons(cons(1 cons(2 empty)) cons(8 empty))"
35     )
36   )
37   (define expected_lista_anidada
38     '((1 2) 3)
39   )
40   (check-equal? (evaluar-programa lista_anidada) expected_lista_anidada)
41
42   ; Lista con sublista y elementos
43   (define lista_mixta
44     (scan&parse
45       "cons(1 cons(cons(2 cons(3 empty)) cons(4 empty)))"
46     )
```

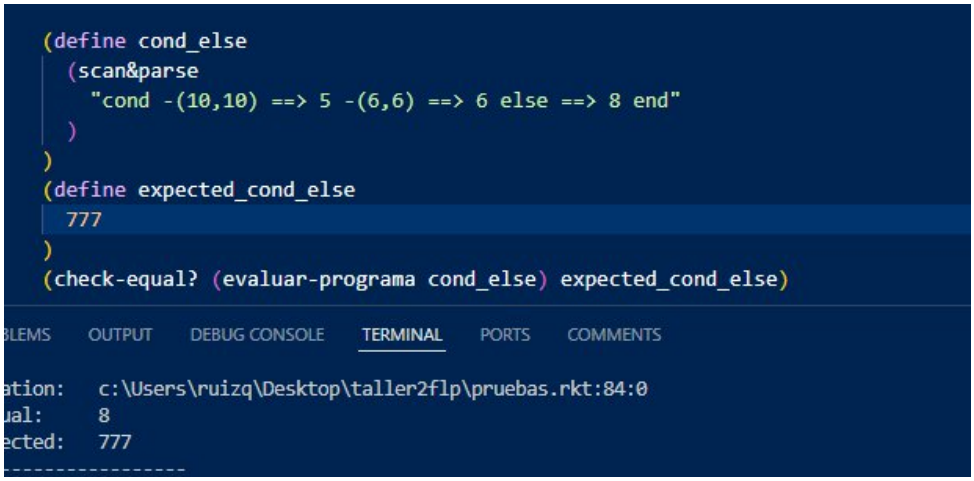
```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS   COMMENTS

location:   c:\Users\ruizq\Desktop\taller2flp\pruebas.rkt:40:0
actual:     '((1 2) 8)
expected:   '((1 2) 3)
--------------------
```

Para las condiciones también incluimos los casos e hicimos las pruebas con rackunit



Todas corresponden a lo esperado y si este se llegara a cambiar obtenemos el error:



Explicacion de las modificaciones en la gramatica:

Modificaciones en la gramatica para las expresiones: cons, empty y cond:

```
;;List-exp
(expresion ("cons" "("expresion expresion")") list-exp)
;List-empty-exp
(expresion ("empty") list-empty-exp)

;;condicionales
(expresion ("cond" (arbno expresion "==>" expresion ) "else" "==>" expresion "end") cond-exp)
```

En la List-exp se define la expresion list-exp la cual tiene como se llama con “cons” y recibe dos expresiones las cuales en la funcion de evaluacion de expresiones realiza la lista con los valores indicados, verifica el caso de exepcion y el empty para retornar vacio.

Por ejemplo un caso de uso puede ser la creacion de una lista con cons, en la cual se inserta en el interprete lo siguiente:

```
-->cons(1 cons(2 cons(3 empty)))
(1 2 3)
```

La expresion cons recibe 2 expresiones las cuales son 1 y otro cons que a su vez se le ingresan dos expreseiones, para asi construir una lista y como expresion terminal el empty.

En List-empty-exp sencillamente se define el nombre de la expresion que en la funcion de evaluar expresiones retorna una lista vacia, por ejemplo:

```
-->empty
()
```

La expresion empty retorna una lista vacia.

En Condicionales se crea la expresion con nombre cond-exp la cual recibe 0 o mas expresiones por eso el uso de arbno donde se indica que dada una expresion dependiendo si es verdadera o falsa retorna otra expresion o entra en el caso del else el cual retorna otra expresion y finalizando con el end, todas estas evaluaciones son realizadas en la funcion de evaluar expresiones.

```
-->cond +(1,1) ==> 2 +(1, 3) ==> 4 else ==> 9 end
2
```

La expresion cond recibe 0 o mas expresiones en este ejemplo en concreto recibe dos expresiones las cuales son verdaderas en este caso, retorna el valor de la primera expresion evaluada dado que es verdadera, la cual retorna 2.

```
-->cond -(1, 1) ==> 1 -(1, 1) ==> 3 else ==> 10 end
10
```

En este caso la expresion retorna el valor indicado en el else dado que las expresiones dadas en el cons no son verdaderas.

Modificaciones en la gramatica para las expresiones primitivas:

```
;;Nuevas primitivas
(primitiva ("length") length-prim)
(primitiva ("first") first-prim)
(primitiva ("rest") rest-prim)
(primitiva ("nth") nth-prim)
(primitiva ("cons1") cons-prim)
(primitiva ("empty1") empty-prim)
```

Se agrega en la defincion gramatical de las primitivas las nuevas primitivas las cuales se desean implementar, las cuales son: length, first, rest, nth, cons1, empty1.

Estas primitivas son evaluadas y cumplen su funcionalidad en la funcion de evaluar primitivas.

```
-->length (cons(1 cons(2 cons(3 empty))))
3
-->first (cons(1 cons(2 cons(3 empty))))
1
-->rest (cons(1 cons(2 cons(3 empty))))
(2 3)
-->nth (cons(1 cons(2 cons(3 empty))), 1)
2
```

En la primitiva length recibe una lista en la cual retorna de manera correcta su longitud, que en este caso es 3.

En la primitiva first recibe una lista y retorna el primer elemento, que en este caso es 1.

En la primitiva rest recibe una lista y retorna el resto de la lista sin el primer elemento, en este caso retorna (2 3) lo cual es el resto de la lista ingresada.

En la primitiva nth recibe una lista y un indice para retornar el valor que se encuentra en el indice indicado de la lista, en este caso se da la lista y el indice 1 y retorna el valor de 2.

Explicacion de las nuevas funciones dentro de la funcion evaluar expresiones:

Funcion para la evaluacion de la expresion cons la cual es el list-exp

```
;;cons
(list-exp (exp1 exp2)
    (if (not (list? (evaluar-expresion exp2 amb)))
        (eopl:error (string-append "Error: " (number->string (evaluar-expresion exp2 amb))  " no es una lista"))
        (append (list (evaluar-expresion exp1 amb)) (evaluar-expresion exp2 amb))
    )
)
```

La funcion list-exp evalua dos expresiones exp1 y exp2 con la finalidad de construir una nueva lista con el valor de exp1 como primer elemento seguidos de los elementos dados en la exp2, pero para eso primero se debe verificar que exp2 sea una lista, de caso contrario se muestra un error.

Primero se evalua que efecitvamente la evaluacion de la expresion exp2 en el ambiente amb sea una lista, lo verifica mediante la funcion list?, si no es una lista envia un error el cual es un mensaje indicando que la exp2 no es una lista, de caso contrario donde exp2 si sea una lista se procede a construir la lista utilizando (evaluar-expresion exp1 amb) y lo convierte en un elemento de una lista con (list(evaluar-expresion exp1 amb)), luego concatena esta lista con el resultado de la evaluacion de la exp2 usando la funcion (append), esto nos permite crear una nueva lista con primer valor exp1 y el resto de los elementos son los de la expresion exp2.

Ejemplos:

```
-->cons (1 cons(2 cons(3 cons (4 empty))))
(1 2 3 4)
```

Se le pasa como exp1 el elemento 1 y la exp2 es un cons que retorna una lista en su evaluacion, lo cual permite la creacion de la lista como se ve en el resultado (1 2 3 4)

```
-->cons (1 cons(2 2))
Error: 2 no es una lista
```

En este caso se presenta un error, ya que la exp2 no llega a ser una lista, dado que en la evaluacion de exp2 la segunda expresion entregada no es una lista, es un numero por ende no se puede crear la lista y retorna el mensaje de error el cual indica que el numero 2 no es una lista.

Funcion para la evaluacion de la expresion empty la cual es el list-empty-exp

```
;;empty
(list-empty-exp ()
                empty
                )
```

La funcion sencillamente retorna una lista vacia.

Funcion para la evaluacion de la expresion cond la cual es cond-exp

```
;;cond
(cond-exp (exp-cond exp-true exp-false)
          (letrec
          (
          (ev-cond (lambda (exp-cond exp-true)
                     (cond
                         [(null? exp-cond) (evaluar-expresion exp-false amb)]
                         [(not (= (evaluar-expresion (car exp-cond) amb) 0)) (evaluar-expresion (car exp-true) amb)]
                         [else (ev-cond (cdr exp-cond) (cdr exp-true))]
                         ))
          )
          )
          (ev-cond exp-cond exp-true)
          )

)
```

La función cond-exp evalúa una estructura condicional conformada por tres expresiones las cuales son: exp-cond, que es una lista de condiciones a evaluar, exp-true, que es una lista de expresiones asociadas a cada condición y exp-false, que es la expresión que se evalúa si ninguna condición es verdadera.

Se utiliza un letrec para definir la función recursiva (ev-cond), la cual recibe exp-cond y exp-true, primero verifica si exp-cond está vacio con null?, en cuyo caso evalúa exp-false en el ambiente amb. Si no está vacío, evalúa la primera condición (car exp-cond), si es verdadera lo cual sabemos dado que da diferente de 0 se evalúa y retorna la expresión correspondiente en exp-true (car exp-true), de lo contrario, llama recursivamente a sí misma con el resto de las listas (cdr exp-cond y cdr exp-true).

Por último, se realiza el llamado inicial a ev-cond con los valores completos de exp-cond y exp-true, esto para procesar las condiciones de forma secuencial y retornar el valor de la primera condición verdadera o, en su defecto, el de exp-false.

Por ejemplo:

```
-->cond +(0, 1) ==> 1 else ==> 10 end
1
```

En este caso se hace el llamado a (ev-cond(exp-cond exp-true)) exp-cond que es +(0,1) lo cual es 1 (diferente de 0) dado que es verdadero se evalua el exp-true que es 1 y retorna 1

Ejemplo:

```
-->cond -(1, 1) ==> 1 else ==> 10 end
10
```

En este caso se hace el llamado a (ev-cond (exp-cond exp-true)) exp-cond que es -(1,1) lo cual es 0 dado que retorna falso, lo cual lleva a evaluar el exp-false que es 10 y este es el valor que se retorna.