Juan Carlos Rojas Quintero 2359358 - 3743
Juan Miguel Palacios Doncel 2359321 - 3743
Yenifer Ronaldo Muñoz Valencia 2278665 - 3743

Modificaciones a la gramática

Listas

La gramática para las listas incluye dos nuevos tipos de expresiones en el lenguaje

- Construcción de listas con `"cons"`.
- Representación de listas vacías con `"empty"`.

```
(expresión ("cons" "(" expresion expresion ")") list-exp)
(expresion ("empty") list-empty-exp)
```

Estos cambios permiten manejar estructuras de datos dinámicas en el lenguaje, esenciales para aplicaciones como algoritmos funcionales y manipulación de datos estructurados.

La estructura sintáctica de las listas se divide en:

1. La expresión comienza con la palabra clave `"cons"`.
2. Entre paréntesis se colocan dos expresiones `"(" expresion expresion ")"`, las cuales representan el valor y una lista respectivamente. Esta representación se debe a que la listas se deben crear recursivamente.

Ejemplos de listas

```
cons(1 cons(2 empty))
```

En este caso se crea una lista donde la primera expresión es 1 y la segunda es nuevamente una lista donde la primera expresión es 2 y la segunda es una lista vacía, lo cual genera una lista (1 2).

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

En este caso se crea una lista anidada donde la primera expresión es una lista que genera (1 2), la cual es el primer elemento de la lista externa, añadiendo como segundo elemento el 3, generando al final la siguiente lista ((1 2) 3).

Condicionales

La gramática para las sentencias “cond” se definió de la siguiente manera:

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

Esta modificación nos permite incluir una estructura de ramificación múltiple con cláusulas condicionales y un caso else obligatorio para manejar las situaciones donde no se cumplan todas las condiciones previas. Además, se debe tener en cuenta que se toma como verdadera cualquier expresión que sea diferente de 0, aunque asumimos que en caso de que la expresión sea "**#f**" no es verdadero por simple lógica.

La estructura sintáctica del condicional se divide en:

1. La expresión comienza con la palabra clave "`cond`".
2. Contiene un número de pares (0 o más) de una expresión condicional seguido de "**==>**" y lo que retorna en caso de que la condición sea verdadera, es decir, {`expresion "==>" expresion`}*.
3. Finaliza con un caso else obligatorio definido como: "`else" "==>" expresion`.
4. Cierra con la palabra clave "`end`".

Ejemplos de la sentencia "cond"

```
cond
>(x, 0) ==> 1
<(x, 0) ==> -1
else ==> 0
end
```

En este caso se evalúa la primera condición, si x es mayor a 0 devuelve 1, si x es menor a 0 devuelve -1, y en caso de que ninguna condición se cumpla, devuelve el caso else, es decir, 0.

```
cond
-(1,1) ==> 2
-(2,2) ==> 1
+(3,5) ==> 9
else ==> 0
end
```

Aquí se evalúa la primera condición, la cual da 0, por lo que continúa a la siguiente condición, donde nuevamente es falsa, siguiendo a la última condición, la cual es verdadera y retorna el valor de 9.

Modificaciones a la función de evaluación

Listas

Debido a que se añadieron dos casos nuevos en la gramática, etiquetado como "list-exp" y "list-empty-exp" para manejar las listas, se realizó la siguiente implementación para crearlas.

```
(list-exp
  (expr1 expr2)
  (let
    (
      (expr1 (evaluar-expresion expr1 amb))
      (expr2 (evaluar-expresion expr2 amb))
    )
    (if (not (list? expr2))
        (eopl:error "Error:" expr2 "no es una lista")
        (cons expr1 expr2)
    )
  )
)

(list-empty-exp
  ()
  '()
)
```

Dentro del caso “list-exp” se reciben las dos expresiones de la gramática, donde primero guardamos en dos variables llamadas “expr1” y “expr2” cada evaluación de las dos expresiones recursivamente hasta que exista una lista vacía, donde se verifica que la segunda expresión sea una lista y así de esta manera asegurar que al crear la lista con las dos expresiones, estas se agrupen de la manera correcta de manera recursiva.

Por otro lado, en el caso “list-empty-exp” simplemente se retorna una lista vacía de acuerdo a la representación gramatical.

Condicionales

Cómo se añadió un nuevo caso en la gramática, etiquetado como “cond-exp” para manejar la sentencia de “cond”, se realizó la siguiente implementación para permitir condicionales con múltiples condiciones, devolviendo el resultado de la primera condición verdadera o un valor por defecto si ninguna condición se cumple.

```
(cond-exp
  (conds exp-trues exp-else)
  (letrec
    (
      (evaluar-cond
        (lambda (conds exp-trues)
          (cond
            [(null? conds)
             (evaluar-expresion exp-else amb)]
            [(and (number? (evaluar-expresion (car conds) amb))
                  (not (= (evaluar-expresion (car conds) amb) 0)))
             (evaluar-expresion (car exp-trues) amb)]
```

```
            [(equal? (evaluar-expresion (car conds) amb) #t)
             (evaluar-expresion (car exp-trues) amb)]
            [else
             (evaluar-cond (cdr conds) (cdr exp-trues))]
           )
         )
       )
     )
     (evaluar-cond conds exp-trues)
  )
)
```

Dentro del case de “cond-exp” recibe la lista de condiciones (“conds”), la lista de expresiones a retornar de cada condición (“exp-trues”), y la expresión del caso “else” (“exp-else”). Luego se crea una función recursiva para implementar la lógica de la evaluación de la siguiente manera:

1. Se recorre la lista de condiciones para verificar si esta es verdadera o falsa, en caso de que todas sean falsas, evalúa si la lista de condiciones es nula, si es así, significa que debe retornar la expresión del “else”.
2. Se verifica si cada condición es un número diferente de cero o un “#t”, si alguna condición al recorrer recursivamente la lista de condiciones es verdadera, se evalúa la expresión que retorna esa condición en la lista.

Explicación de funciones creadas

Condicionales

Para evaluar la expresión del condicional “cond”, se tuvo que utilizar una función auxiliar recursiva para manejar la lista de condiciones y la lista de retornos de cada condición, esto se hizo debido a que era más sencillo recorrer la listas recursivamente para evaluar una por una las condiciones y verificar si eran verdaderas o falsas, en caso de que no se cumpla ninguna, se evalúa el caso de una lista nula y se evalúa la expresión “else”. Dicha función se implementó de la siguiente manera:

```
(letrec
    (
      (evaluar-cond
        (lambda (conds exp-trues)
          (cond
            [(null? conds)
             (evaluar-expresion exp-else amb)]
            [(and (number? (evaluar-expresion (car conds) amb))
                  (not (= (evaluar-expresion (car conds) amb) 0)))
             (evaluar-expresion (car exp-trues) amb)]
            [(equal? (evaluar-expresion (car conds) amb) #t)
             (evaluar-expresion (car exp-trues) amb)]
            [else
```

```
                (evaluar-cond (cdr conds) (cdr exp-trues))]
            )
          )
        )
      )
      (evaluar-cond conds exp-trues)
  )
```

Básicamente, la función evaluar-cond recibe la lista de condiciones y la lista de retornos de cada condición, luego verifica si la lista es nula, es decir, el caso donde todas las condiciones sean falsas para evaluar la expresión “else”. Si la lista de condiciones no es nula, evalúa si el primer elemento de la lista de condiciones es un número diferente de 0 o “#t”, si es así, retorna la evaluación de la expresión correspondiente en la lista de “exp-trues” que tiene lo que retorna la condición que se está evaluando en dicho momento. Por último, si la condición que se está evaluando en dicho momento no es verdadera, continua con los demás elementos de la lista de condiciones y la lista “exp-trues” recursivamente.

Ejecución y resultados

Listas

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

En esta prueba creamos una lista con dos elementos y devolvemos una lista con los dos elementos.

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

En esta prueba probamos el caso de lista vacía donde devolvemos una lista vacía.

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

En esta prueba probamos el caso de una lista de un elemento.

```
-->cons (2 cons(5 cons(6 cons(7 empty))))
(2 5 6 7)
```

En esta prueba probamos la construcción de una lista con 4 elementos.

```
-->cons(1 let l = cons(2 cons(3 empty)) in l)
(1 2 3)
```

En esta prueba probamos la creación de listas con los let.

```
-->let l = cons(1 cons(2 empty)) in let l2 = cons(3 cons(4 empty)) in cons(l l2)
((1 2) 3 4)
```

En esta prueba probamos la creación de listas anidadas con los let.

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

En devuelve el resto de la lista, en este caso tenemos dos elementos 1 y 2, el resto seria 2.

```
stre/Asignaciones/Taller-2-FLP/interpretadorClase.rkt
-->nth(cons (2 cons(5 cons(6 cons(7 empty)))), 1)
5
```

En este caso tenemos la lista con 2 5 6 7 y nos piden que devolvamos el elemento en la posición 1, en este caso es 5.

```
-->rest(let l = cons(1 cons(2 empty)) in let l2 = cons(3 cons(4 empty)) in cons(l l2))
(3 4)
```

en este caso devolvemos el resto de la lista, como tenemos 3 elementos que son una sublista 1 y 2, el 3 y el 4, al devolver el resto, nos devuelve 3,4

Condicionales

```
stre/Asignaciones/Taller-2-FLP/test.rkt
-->cond +(1,1) ==> 2 else ==> 0 end
2
-->
```

En esta prueba se evalúa una condición que sea verdadera y se espera que su resultado sea 2, lo cual es cierto, ya que 1 + 1 es 2, lo cual es diferente de 0, por lo que devuelve 2.

```
stre/Asignaciones/Taller-2-FLP/test.rkt
-->cond 0 ==> 1 else ==> 9 end
9
-->
```

En esta prueba se evalúa una condición donde se retorna el caso “else”, donde se evalúa la única condición de la sentencia, la cual es 0, por lo que es falsa y al no encontrar más condiciones, pasa al caso del “else”.

```
stre/Asignaciones/Taller-2-FLP/test.rkt
-->cond else ==> 0 end
0
-->
```

En esta prueba se evalúa las condiciones, pero como no existe ninguna (caso posible según la gramática), la lista de condiciones es vacía y simplemente retorna la expresión del “else”.

```
stre/Asignaciones/Taller-2-FLP/test.rkt
-->cond -(1,1) ==> 2 -(3,3) ==> 0 +(1,2) ==> 3  else ==> 10 end
3
```

En esta prueba se evalúan múltiples condiciones una por una, hasta que se encuentra una que es verdadera, ya que 1 + 2 es 3, lo cual es diferente de 0 y retorna el valor indicado que es 3.

```
-->cond false ==> 1 else ==> 0 end
0
```

En esta prueba verificamos el caso donde la condición sea directamente “#f”, lo cual es algo que asumimos que no debe ser válido por simple lógica, por lo que retorna la expresión del “else”.

```
stre/Asignaciones/Taller-2-FLP/test.rkt
-->cond ==(5, 4) ==> 1 let l = cons(1 cons(2 empty)) in ==(length(l), 2) ==> 3 else ==> 9 end
3
```

En esta prueba se evalúan diferentes condiciones con expresiones relacionales y la sentencia “let”. Al final la única condición válida es cuando la longitud de la lista l es 2, por lo que retorna 3.

```
stre/Asignaciones/Taller-2-FLP/test.rkt
-->cond >(5, 10) ==> true ==(1, 2) ==> 3 else ==> cond let x = 5 in -(x, 3) ==> true else ==> 0 end end
#t
```

En esta prueba utilizamos condicionales anidados, donde primero se evalúa si 5 es mayor a 10, lo cual es falso y continua con la otra condicion que tambien es falsa, por lo que retorna la expresión else, donde evalúa otra sentencia “cond”, donde se crea una sentencia “let” que retorna x-3, lo cual es 5 y como es diferente de 0, retorna “#t” según el ejemplo.

```
stre/Asignaciones/Taller-2-FLP/test.rkt
-->cond let x = cond >(5, 6) ==> 1 <(5, 6) ==> 2 else ==> 3 end l = cons(1 cons(2 cons(3 empty))) in nth(l, x) ==> cond let y = cons(1 cons(2 empty)) in ==
(first(y), 1) ==> 1 else ==> 0 end else ==> 9 end
1
```

En esta prueba, se utilizan varios casos de condicionales, primero se evalúa una sentencia “let” como primer condición, la cual define una variable x que es igual a una sentencia “cond” que retorna 2, también se crea una lista “l” igual a (1 2 3) que se utiliza en el cuerpo del “let” para devolver el elemento en la posición 2 de la lista, la cual es el último elemento. Como la primera condición que evaluamos dio 3, siendo diferente de 0, se evalúa la expresión que retorna, la cual es una sentencia “cond” donde como primer condición se encuentra una sentencia “let” que crea una lista (1 2) y evalúa si el primer elemento de la lista es 1, lo cual es cierto, por lo que la expresión del primer condicional devuelve “#t” y retorna al final la expresión 1.