



Lenguajes de Programación, 2017-2

Práctica 6: Recursividad

Karla Ramírez Pulido

José Ricardo Rodríguez Abreu

Manuel Soto Romero

Fecha de inicio: 21 de abril de 2017

Fecha de término: 5 de mayo de 2017



1. Objetivos

- Implementar una versión completa del intérprete para el lenguaje diseñado en la Práctica 5 del curso. La implementación deberá soportar condicionales, identificadores, expresiones aritméticas, expresiones booleanas, listas, funciones de primera clase y expresiones recursivas, usando evaluación glotona para interpretar las expresiones¹.
- Programar tres versiones de un procedimiento que encuentre el n -ésimo número de Tribonacci usando distintos mecanismos de recursividad para ilustrar sus diferencias y ventajas.

2. Archivos requeridos

Anexo a este archivo en formato PDF se encuentran los siguientes archivos, necesarios para desarrollar la práctica:

- Un archivo `tribonacci.rkt` que contiene el esqueleto de algunas funciones recursivas a completar.
- Un archivo `grammars.rkt` que contiene los TDA necesarios para implementar el intérprete.
- Un archivo `parser.rkt` que contiene las funciones necesarias para convertir sintaxis concreta en la sintaxis abstracta correspondiente.
- Un archivo `interp.rkt` que contiene los procedimientos necesarios para evaluar las expresiones del lenguaje.
- Un archivo `practica6.rkt` que se encarga de ejecutar el intérprete final, usando todas las funciones anteriores.

3. Desarrollo de la práctica

Parte I: Intérprete recursivo

Completar los siguientes ejercicios para lograr la correcta ejecución del archivo `practica6.rkt` que implementa un intérprete para el siguiente lenguaje descrito en notación EBNF²:

¹La práctica se entrega siguiendo los lineamientos especificados en la página del curso <http://lenguajesfc.com/lineamientos.html> y por equipos de tres integrantes.

²Del inglés Extended Backus–Naur Form

```

<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 | .. | z | A | ... | Z | aa | ab | ... | aaa | ...
      (Cualquier combinación de caracteres alfanuméricos
       con al menos uno alfabético)

<num> ::= ... | -2 | - 1 | 0 | 1 | 2 | ...

<bool> ::= true | false

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

<op> ::= + | - | * | / | % | min | max | pow
        | neg | and | or | < | > | <= | >= | = | != | zero?
        | head | tail | empty?

```

Ejercicios

1. (1 pt.) Completar el cuerpo de la función (`parse sexp`) contenida en el archivo `parser.rkt` que recibe una expresión en sintaxis concreta y posteriormente construir el árbol de sintaxis abstracta del lenguaje RCFWBAEL.
2. (1 pt.) Completar el cuerpo de la función (`desugar sexps`) contenida en el archivo `parser.rkt` que recibe una expresión dentro del lenguaje RCFWBAEL y eliminar el azúcar sintáctica, es decir, regresar el árbol de sintaxis abstracta dentro del lenguaje RCFBAEL.

RCFBAEL es una versión *desendulzada* de RCFWBAEL que no cuenta con constructores para `with`, `with*` ni `cond`. Para eliminar el azúcar sintáctica de este tipo de expresiones, considerar:

- `with` puede ser expresado como una aplicación de función endulzada.
- `with*` puede ser expresado como una cadena de expresiones `with` anidadas.
- `cond` puese ser expresado como una cadena de expresiones `if` anidadas.

3. (3 pts.) Completar el cuerpo de la función (`interp expr env`) contenida en el archivo `interp.rkt` que recibe un árbol de sintaxis abstracta `RCFBAEL` y un ambiente de sustitución y a su vez regresa la interpretación del árbol como un valor de tipo `RCFBAEL-Value`³.

El funcionamiento correcto del intérprete, dependerá (1) tanto del cómo se introducen las variables al ambiente al aplicar una función, (2) como de la implementación de una función `lookup` que recupere el valor de los identificadores en ambientes recursivos.

Parte II: Números de Tribonacci

Los números de Tribonacci son como los números de Fibonacci pero sumando los tres números anteriores cada vez, empezando la secuencia con 2 ceros:

0, 0, 1, 1, 2, 4, 7, 13, 24, 44, ...

La función que encuentra el n -ésimo número de Tribonacci se define recursivamente como:

```
tribonacci 0 = 0
tribonacci 1 = 0
tribonacci 2 = 1
tribonacci n = tribonacci (n-1) + tribonacci (n-2) + tribonacci (n-3)
```

A continuación se presenta un procedimiento recursivo que encuentra el n -ésimo número de Tribonacci en el lenguaje Racket:

```
(define (tribonacci n)
  (cond
    [(< n 2) 0]
    [(= n 2) 1]
    [else
     (+ (tribonacci (- n 1))
        (tribonacci (- n 2))
        (tribonacci (- n 3)))])])
```

Usar esta definición para completar el cuerpo de las funciones faltantes del archivo `tribonacci.rkt` como sigue:

1. (1 pt.) La función (`tribonacci-cola n`) debe encontrar el n -ésimo número de Tribonacci usando recursividad de cola.
2. (2 pts.) La función (`tribonacci-cps n`) debe encontrar el n -ésimo número de Tribonacci usando *Continuation Passing Style*.
3. (2 pts.) La función (`tribonacci-memo n`) debe encontrar el n -ésimo número de Tribonacci usando memoización⁴.

³El ambiente usando una estructura de lista con comportamiento de pila está definido en el archivo `grammars.rkt`.

⁴Se deben usar *hashes* para almacenar los resultados.

Referencias

- [1] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Brown University, 2007.
- [2] Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.