Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Práctica 6

Karla Ramírez Pulido
karla@ciencias.unam.mx

J. Ricardo Rodríguez Abreu
ricardo_rodab@ciencias.unam.mx

Manuel Soto Romero
manu@ciencias.unam.mx

Luis A. Patlani Aguilar
ayudantefc@gmail.com

Fecha de inicio: 6 de noviembre de 2017
Fecha de término: 20 de noviembre de 2017
Semestre 2018-1

Objetivos

- Agregar listas y nuevos operadores al lenguaje de la Práctica 5.
- Modificar el intérprete implementado en la Práctica 5 para que procese asignaciones locales recursivas.
- Modificar una función recursiva usando las técnicas de optimización de recursividad de cola y memoización, para analizar posteriormente su complejidad.

Descripción general

La práctica consiste en extender el intérprete de la práctica anterior. La gramática en EBNF para las expresiones del lenguaje `RCFWBAEL/L` (*Recursive Conditionals functions With Boolean Arithmetic Expressions Lists and Lazy*), que se debe implementar en esta práctica es la siguiente:

```
<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 | ...
          (Cualquier número admitido por Racket)

<bool> ::= false | true

<list> :: empty
        | {list <expr>+}

<op> ::= + | - | * | / | % | min | max | pow | zero?
       | not | and | or | < | > | <= | >= | = | /=
       | head | tail | empty?
```

Técnicas de optimización

Como se vio en clase, los lenguajes de programación que permiten usar recursividad al programador tienen el problema de que consumen demasiados registros de activación, al ejecutar funciones de esta naturaleza y se discutieron dos técnicas para optimizar este tipo de funciones: *recursividad de cola* y *memoización.*

El último ejercicio de la práctica consiste en modificar y analizar las complejidades en tiempo y espacio de la función de Tribonacci y compararla con otras versiones optimizadas.

Archivos requeridos

El material de esta práctica consta de los siguientes archivos[1]:

- `grammars.rkt` archivo con la definición de los tipos de datos abstractos que definen cada uno de los ASA para el lenguaje `RCFWBAEL/L`.
- `parser.rkt` archivo en el cual se debe implementar el analizador sintáctico para el lenguaje `RCFWBAEL/L`.
- `interp.rkt` archivo donde se debe implementar el analizador semántico para el lenguaje `RCFWBAEL/L`.
- `practica6.rkt` archivo con la lógica necesaria para ejecutar el intérprete final de `RCFWBAEL/L`.
- `test-practica6.rkt` archivo en el cual se deben agregar las pruebas unitarias de la práctica.
- `tribonacci.rkt` archivo en el cual se debe modificar la función de Tribonacci.

[1]Los archivos pueden descargarse desde la página del curso `http://lenguajesfc.com`

Desarrollo de la práctica

Ejercicio 6.1 (1 pt.) En el archivo `grammars.rkt` se encuentra el TDA que define los constructores para realizar el análisis sintáctico del lenguaje `RCFWBAEL/L`.

```
;; TDA para representar los árboles de sintáxis abtracta del
;; lenguaje RCFWBAEL/L. Este TDA es una versión con azúcar sintáctica.
(define-type RCFWBAEL/L
   [idS (i symbol?)]
   [numS (n number?)]
   [boolS (b boolean?)]
   [listS (elems (listof RCFWBAEL/L?))]
   [opS (f procedure?) (args (listof RCFWBAEL/L?))]
   [ifS (expr RCFWBAEL/L?) (then-expr RCFWBAEL/L?) (else-expr RCFWBAEL/L?)]
   [condS (cases (listof Condition?))]
   [withS (bindings (listof bindingS?)) (body RCFWBAEL/L?)]
   [withS* (bindings (listof bindingS?)) (body RCFWBAEL/L?)]
   [recS (bindings (listof bindingS?)) (body RCFWBAEL/L?)]
   [funS (params (listof symbol?)) (body RCFWBAEL/L?)]
   [appS (fun-expr RCFWBAEL/L?) (args (listof RCFWBAEL/L?))])

;; TDA para asociar identificadores con valores con azúcar sintáctica.
(define-type BindingS
   [bindingS (name symbol?) (value RCFWBAEL/L?)])

;; TDA para representar condiciones.
(define-type Condition
   [condition (expr RCFWBAEL/L?) (then-expr RCFWBAEL/L?)]
   [else-cond (else-expr RCFWBAEL/L?)])
```

El primer ejercicio a implementar en el intérprete de esta práctica, consiste en completar el cuerpo de la función `parse` (incluida en el archivo `parser.rkt`) para que realice el análisis sintáctico correspondiente. Además, se deben agregar al menos cinco pruebas unitarias dentro del archivo `test-practica6.rkt`[2].

```
;; parse: s-expression -> RCFWBAEL/L
(define (parse sexp)
   (error 'parse "Función no implementada."))
```

```
> (parse '{list 1 2 3})
(listS (list (numS 1) (numS 2) (numS 3))
```

[2]Se deben agregar pruebas significativas.

Ejercicio 6.2 (1 pt.) Una vez que se complete el cuerpo de la función `parse`, implementar el cuerpo de la función `desugar` incluída en el archivo `parser.rkt`, el cual elimina el azúcar sintáctica[3] de las expresiones de `RCFWBAEL/L`, es decir, las convierte en expresiones de `RCFBAEL/L`:

```
;; TDA para representar los árboles de sintaxis abstracta del
;; lenguaje RCFBAEL/L. Este TDA es uan versión sin azúcar sintáctica.
(define-type RCFBAEL/L
   [id (i symbol?)]
   [num (n number?)]
   [bool (b boolean?)]
   [lisT (elems (listof RCFBAEL/L?))]
   [op (f procedure?) (args (listof RCFBAEL/L?))]
   [iF (expr RCFBAEL/L?) (then-expr RCFBAEL/L?) (else-expr RCFBAEL/L?)]
   [fun (params (listof symbol?)) (body RCFBAEL/L?)]
   [rec (bindings (listof Binding?)) (body RCFBAEL/L?)]
   [app (fun-expr RCFBAEL/L?) (args (listof RCFBAEL/L?))])

;; TDA para asociar identificadores con valores sin azúcar sintáctica.
(define-type Binding
   [binding (name symbol?) (value RCFBAEL/L?)])
```

Se deben agregar al menos cinco pruebas unitarias dentro del archivo `test-practica6.rkt` que correspondan con las pruebas agregadas en el ejercicio anterior.

Las únicas expresiones que tienen azúcar sintáctica, en esta práctica, son:

- `with` estas expresiones son una versión endulzada de aplicaciones a función. Por ejemplo:

  ```
  {with {{a 3}}
     {+ a 4}}
  ```

  se transforma en

  ```
  {{fun {a} {+ a 4}} 3}
  ```

- `with*` este tipo de expresiones son una versión endulzada de expresiones `with` anidadas que a su vez tienen azúcar sintáctica. Por ejemplo:

  ```
  {with* {{a 2} {b {+ a a}}}
     b}
  ```

  se transforma en

  ```
  {with {{a 2}}
     {with {{b {+ a a}}}
        b}}
  ```

  que a su vez se convierte en

[3] Tipo de sintaxis que hace que un programa sea más "dulce" o fácil de escribir.

```
{{fun {a} {{fun {b} b} {+ a a}}} 2}
```

- cond este tipo de expresiones son una versión endulzada de expresiones `if`. Por ejemplo:

  ```
  {cond
     {{< 2 3} 1}
     {{> 10 2} 2}
     {else 3}}
  ```

  se transforma en

  ```
  {if {< 2 3}
      1
      {if {> 10 2}
          2
          3}}
  ```

```
;; desugar: RCFWBAEL/L -> RCFBAEL/L
(define (desugar expr)
   (error 'desugar "Función no implementada."))
```

```
> (desugar (parse '{cond {{< 2 3} 1} {{> 10 2} 2} {else 3}}))
(iF
    (op < (list (num 2) (num 3))
    (num 1)
    (iF (op > (list (num 10) (num 2))) (num 2) (num 3)))
```

Ejercicio 6.3 (5 pts.) El analizador semántico de esta práctica evalúa las expresiones mediante ambientes y cerraduras para implementar alcance estático. Para implementar evaluación perezosa se agregan cerraduras de expresiones. Para implementar recursividad se agregan ambientes recursivos que hacen uso de cajas. De esta forma se recibe un ASA (sin azúcar sintáctica) y un ambiente de evaluación definido por:

```
;; TDA para representar los ambientes de evaluación.
(define-type Env
   [mtSub]
   [aSub (name symbol?) (value RCFBAEL/L-Value?) (env Env?)]
   [aRecSub (name symbol?) (value boxed-RCFBAEL/L-Value?) (env Env?)])

;; Para trabajar con cajas que guarden el resultado de evaluación.
(define (boxed-RCFBAEL/L-Value? v)
   (and (box? v) (RCFBAEL/L-Value? (unbox v))))
```

Los resultados devueltos por el analizador semántico, que almacena el ambiente deben ser del tipo de dato `RCFBAEL/L-Value`:

```
;; TDA para representar los resultados devueltos por el intérprete.
(define-type RCFBAEL/L-Value
   [numV (n number?)]
   [boolV (b boolean?)]
   [closureV (params (listof symbol?)) (body RCFBAEL/L?) (env Env?)]
   [exprV (expr RCFBAEL/L?) (env Env?)]
   [listV (listof (RCFBAEL/L-Value?))])
```

El ejercicio consiste en completar el cuerpo de la función `(interp exp env)` (ver archivo `interp.rkt`) para que realice el análisis semántico correspondiente, es decir, evaluar expresiones de `RCFBAEL/L`. Tomar los siguientes puntos a consideración:

- El valor de los identificadores debe ser buscando en el ambiente de evaluación mediante la definición de una función `lookup`, la cual encuentra el valor asociado en el ambiente y lo regresa o bien reporta un error, en caso de que no se haya encontrado. En caso de recibir un ambiente recursivo esta función abre la caja asociada para seguir con la ejecución. Ejemplo:

  ```
  > (interp (desugar (parse 'foo)) (mtSub))
  lookup: Free identifier
  ```

- Los números se evalúan a valores de tipo `numV`. Ejemplo:

  ```
  > (interp (desugar (parse '1729)) (mtSub))
  (numV 1729)
  ```

- Las expresiones booleanas se evalúan a valores de tipo `boolV`. Ejemplo:

  ```
  > (interp (desugar (parse 'true)) (mtSub))
  (boolV #t)
  ```

- Las listas se evalúan a valores de tipo `listV`. Ejemplo:

  ```
  > (interp (desugar (parse '{list 1 2 3})) (mtSub))
  (listV (list (numV 1) (numV 2) (numV 3)))
  ```

- Los operadores son n-arios, por lo que esta versión del intérprete tiene un constructor `op` que recibe una función con la cual aplica la operación definida a cada uno de sus parámetros. En este tipo de expresiones se encuentra un punto estricto, se deben evaluar cada uno de sus operandos para aplicar el operador mediante la definición de una función `strict`. Los nuevos operandos para esta práctica son: `zero?`, `head`, `tail` y `empty?`. Ejemplo:

  ```
  > (interp (desugar (parse '{/= 1 2 3 4 5})) (mtSub))
  (boolV #t)
  ```

- Las expresiones `if` deben evaluar su condición y regresar el valor correspondiente en caso de que la condición se evalúe a verdadero o falso, según sea el caso. En este tipo de expresiones se encuentra un punto estricto, se debe evaluar la condición para regresar el valor correcto mediante la definición de una función `strict`. Ejemplo:

  ```
  > (interp (desugar (parse '{if true true false})) (mtSub))
  (boolV #t)
  ```

- Las expresiones `cond` deben evaluar cada condición y regresar el valor correspondiente en caso de que alguna sea verdadera, en caso contrario, se deben seguir evaluando el resto de condiciones o ejecutar el caso `else` si ninguna condición fue verdadera. Al ser azúcar sintáctica de expresiones `if`, cada condición de este tipo de expresiones también es un punto estricto. Ejemplo:

```
> (interp (desugar (parse '{cond {true 1} {true 2} {else 3}})) (mtSub))
(numV 1)
```

- Las expresiones `with` son multiparamétricas, es decir, tienen más de un identificador. Ejemplo:

```
> (interp (desugar (parse '{with {{a 2} {b 3}} {+ a b}})) (mtSub))
(numV 5)
```

- Las expresiones `with*` presentan un comportamiento parecido al de la primitiva `with`, sin embargo, estas expresiones permiten definir identificadores en términos de otros definidos anteriormente. Por ejemplo: `{with* {{a 2} {b {+ a a}}} b}`. Se interpreta similar a `with`, la única diferencia es que también se deben procesar los identificadores. Ejemplo:

```
> (interp (desugar (parse '{with* {{a 2} {b {+ a a}}} b})) (mtSub))
(exprV
    (op + (list (id 'a) (id 'a)))
    (aSub 'a (exprV (num 2) (mtSub)) (mtSub)))
```

- Las expresiones `rec` presentan un comportamiento parecido al de la primitiva `with*`, sin embargo, estas expresiones permiten definir identificadores recursivos, es decir, que se definen en términos de sí mismos. Por ejemplo: `{rec {{fac {fun {n} {if {zero? n} 1 {* n {fac {- n 1}}}}}}} {n 5}} {fac n}}`. Se interpreta similar a `with*`, la diferencia es que se utilizan ambientes recursivos que hacen uso de cajas para almacenar el cuerpo de la función. Ejemplo:

```
> (define expr
    '{rec {
        {fac {fun {n}
            {if {zero? n}
                1
                {* n {fac {- n 1}}}}}}
        {n 5}}
        {fac n}})
> (interp (desugar (parse expr)) (mtSub))
(numV 120)
```

- Las funciones deben evaluarse a una cerradura que incluya: los parámetros de la función, el cuerpo de la función y el ambiente dónde fue definida con el fin de evaluarlas usando alcance estático. Ejemplo:

```
> (interp (desugar (parse '{fun {x} {+ x 2}})) (mtSub))
(closureV '(x) (op + (list (id 'x) (num 2)) (mtSub))
```

- En las aplicaciones de función se encuentra un punto estricto, se debe evaluar la función para poder aplicarla con los argumentos correspondientes mediante la definición de una función `strict`. Este tipo de expresiones deben de evaluar el cuerpo de la función correspondiente considerando el ambiente donde ésta fue definida y agregando las expresiones al ambiente sin evaluar. Por ejemplo, en la siguiente llamada, el ambiente de evaluación es:

  ```
  (aSub 'a (exprV (op + (list (num 2) (num 3))) (mtSub)) (mtSub))
  > (interp (desugar (parse '{{fun {x} {+ x 2}} {+ 2 3}})) (mtSub))
  (numV 7)
  ```

Además, se deben agregar al menos cinco pruebas unitarias dentro del archivo `test-practica6.rkt` que correspondan con las pruebas agregadas en los ejercicios anteriores.

```
;; interp: RCFBAEL/L -> RCFBAEL/L-Value
(define (interp expr env)
   (error 'interp "Función no implementada."))
```

```
> (interp (desugar (parse '{head {list 1 2 3}})) (mtSub))
(numV 1)
```

Ejercicio 6.4 (1 pt.) Una vez finalizados los ejercicios anteriores, modificar el archivo `practica6.rkt` para que procese los resultados devueltos por el intérprete. Ejecutarlo y realizar pruebas significativas para verificar que la práctica se completó con éxito.

```
Bienvenido a DrRacket, versión 6.10.1 [3m].
Lenguaje: plai, with debugging; memory limit: 128 MB.
(λ) {if {> 2 3} 1 2}
2
(λ) {cond {{> 2 3} 1} {{> 2 10} 2} {else 3}}
3
```

Figura 1: *Ejecución de* `RCFWBAEL/L`

Ejercicio 6.5 (2 pts.) Los números de Tribonacci son similares a los números de Fibonacci pero en cada paso se deben sumar los tres números anteriores, empezando la sucesión con dos ceros:

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

La función que encuentra el *n*-ésimo elemento de la sucesión 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)
```

1. Definir dentro del archivo `tribonacci.rkt` una función recursiva `tribonacci` que implemente la definición anterior en Racket.

```
;; tribonacci: number -> number
(define (tribonacci n)
   (error "Función no implementada"))
```

2. Definir dentro del archivo `tribonacci.rkt` una función `tribonacci-cola` que optimice la función anterior usando la técnica de *recursión de cola* vista en clase.

```
;; tribonacci-cola: number -> number
(define (tribonacci-cola n)
   (error "Función no implementada"))
```

3. Definir dentro del archivo `triconacci.rkt` una función `tribonacci-memo` que optimice la función anterior usando la técnica de *memoización* vista en clase. Para almacenar los resultados se deben usar tablas de dispersión por medio de las funciones: `make-hash`, `hash-ref` y `hash-set!` de Racket.

```
;; tribonacci-memo: number -> number
(define (tribonacci-memo n)
   (error "Función no implementada"))
```

4. En el archivo `readme` (en formato PDF) llenar la siguiente tabla que compara los resultados devueltos por las funciones anteriores al pasarles como argumento el número 10,000.

Comparación	`tribonacci`	`tribonacci-cola`	`tribonacci-memo`
Registros usados			
Complejidad en tiempo			
Complejidad en espacio			
Tiempo en ejecución			
Resultado			

Puntaje total: *10 puntos*

Entrega

Los archivos que se deben enviar a los ayudantes de laboratorio son:

- `parser.rkt`
- `interp.rkt`
- `test-practica6.rkt`
- `tribonacci.rkt`

Recordando seguir los lineamientos de entrega de prácticas especificados en la sección correspondiente de la página del curso: `http://lenguajesfc.com/lineamientos.html`.

Referencias

Algunas referencias de consulta:

[1] Karla Ramírez, Manuel Soto, *Notas de laboratorio del curso de Lenguajes de Programación*, Semestre 2018-1, Facultad de Ciencias, UNAM. Disponibles en: [`http://lenguajesfc.com/notas.html`].

[2] Rodrigo Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.

[3] Shriram Krishnamurthi, *Programming Languages: Application and Interpretation*, Primera edición, Brown University, 2007.

[4] Dexter Kozen, *Data Structures and Functional Programming*, notas de clase, primavera 2011, Cornell University. Disponibles en: [`https://www.cs.cornell.edu/courses/cs3110/2011sp/`]