Taller 2: Abstracción de datos y sintáxis abstracta

Fundamentos de Lenguajes de Programación / 750095M / Grupo 01 / Prof. Robinson Duque / Monitor Juan Marcos Caicedo / 2019-2

1. **(7 pts)** Implementar la interfaz *Bignum* para $N = 32$ utilizando listas, similar a como se muestra en la diapositiva número 12 de las transparencias del tema de Abstracción de datos, con *Bignum* de $N = 16$. Recuerde que la representación *Bignum* está definida así (para un N grande):

$$\lceil n \rceil = \left\{ \begin{array}{ll} () & n = 0 \\ (\text{cons } r \lceil q \rceil) & n = qN + r, 0 \leq r < N \end{array} \right\}$$

Donde, por ejemplo, si $N = 32$, algunas representaciones son:

- $\lceil 33 \rceil$ = '(1 1) = $((1 \times 32^0) + (1 \times 32^1))$
- $\lceil 37031 \rceil$ = '(7 5 4 1) = $((7 \times 32^0) + (5 \times 32^1) + (4 \times 32^2) + (1 \times 32^3))$
- $\lceil 41099362 \rceil$ = '(2 3 8 6 1 7) = $((2 \times 32^0) + (3 \times 32^1) + (8 \times 32^2) + (6 \times 32^3) + (7 \times 32^4) + (1 \times 32^5))$

Importante: En su implementación deben estar las entidades de la interfaz *zero, is-zero?, successor, predecessor* definidas de manera correcta para que la implementación funcione.

La implementación de la interfaz debe funcionar correctamente para el *código cliente* ya definido en clase:

- Suma
- Resta
- Multiplicación
- Potencia
- Factorial

Realice al menos 2 pruebas por cada operación para verificar su correcto funcionamiento.

2. **(11 pts)** Implementar la interfaz *Diff-tree*, que es, en sí, una representación de todos los **enteros** (negativos y no-negativos). **Diff-tree** es una lista que posee una gramática BNF recursiva, descrita así:

Diff-tree ::= (one) | (diff *Diff-tree Diff-tree*)

Por ejemplo, en esta interfaz, la lista (one) representa el 1. Si t_1 representa a n_1 y t_2 representa a n_2, entonces (diff t_1 t_2) es una representación de $n_1 - n_2$.

Así, tanto (one) como (diff (one) (diff (one) (one))) son ambas representaciones del número 1; (diff (diff (one) (one)) (one)) es una representación del -1. **Debe:**

(a) Extender esta interfaz para poder representar a todos los **enteros**, para ello debe implementar: **zero**, **is-zero?**, **successor**, **predecessor**. Debe tener en cuenta los números enteros negativos. Por ejemplo, la función **successor** recibe una de las infinitas representaciones legales del 1, debe producir una de las representaciones legales del 2. Es permisible también que para diferentes representaciones legales del 1 se obtengan diferentes representaciones legales del 2.

(b) Escribir un procedimiento **diff-tree-plus** que realiza la suma (adición) en esta representación.

3. **(11 pts)** Implementar el tipo de dato *stack* (pila), que consiste de las entidades:

- *empty-stack*: Definición de pila vacía.
- *push*: Insertar elemento en una pila.
- *pop*: Retira el elemento superior de la pila.
- *top*: Devuelve el elemento superior de la pila. (sin retirarlo)
- *empty-stack?*: Predicado que se encarga de preguntar si la pila está vacía.

El tipo de dato pila debe ser implementado y representado usando:

1. Listas
2. Procedimientos
3. Datatypes

4. **(11 pts)** En la sección 2.2 del Capítulo 2 del Libro del curso **Essentials of Programming Languages** se muestra la noción de ambiente, de la representación abstracta de estructuras de datos, la gramática recursiva de la expresión de ambiente (env-exp) y el primer interprete; sus componentes: **empty-env**, **extend-env** y **apply-env**.

Para el siguiente punto, debe considerar la noción de Ambiente con los procesos ya mencionados (y que su código es encontrado en el Libro) y además, la funcionalidad llamada $extend-env*$. Lo que realiza este constructor, es tomar una lista de variables, una lista de valores (ambas listas de misma longitud), y un ambiente, y extender cada variable asociada con cada valor en el ambiente dado. Más formalmente:

Donde:

$$g(var) = \begin{cases} val_i & \text{si } var = var_i \text{ para cualquier } i \text{ tal que } 1 \leq i \leq k \\ f(var) & \text{en cualquier otro caso.} \end{cases}$$

$$(extend - env* \ \ (var_1 ... var_k) \ \ (val_1 ... val_k) \ \ \lceil f \rceil) = \ \ \lceil g \rceil$$

El cuerpo de la función $extend - env*$ se ve así:

```
(define extend-env*
  (lambda (vars vals env)
    (list 'extend-env* vars vals env)))
```

$\Rightarrow$ Su tarea es implementar una función llamada $check-env$, que, recibe como prámetros un ambiente e y un número n, y retorna una lista con 1 o más parejas de valores asociados que pertenecen al n-ésimo nivel del ambiente. Para efectos prácticos, cuando $n = 0$ se refiere al ambiente vacío $empty-env$. En caso de que el número n sea mayor que la cantidad de ambientes en e, debe lanzar un error mediante $eopl : error$.

Por ejemplo, sobre el ambiente e:

```
> (define e
          (extend-env 'y 8
            (extend-env* '(x z w) '(1 4 5)
              (extend-env 'a 7
                (empty-env)))))
```

Algunas salidas son:

```
> (check-env e 0)
()

> (check-env e 1)
((a 7))

> (check-env e 2)
((x 1) (z 4) (w 5))
```

```
> (check-env e 3)
((y 8))

> (check-env e 4)
check-env: Not possible to search depth on environment
```

5. **(26 pts)** Un árbol binario de búsqueda es un árbol binario que cumple con la siguiente propiedad de orden, para cada uno de sus nodos: el subárbol izquierdo de cualquier nodo (si no está vacío) sólo contiene valores menores que el del nodo, y el subárbol derecho (si no está vacío) sólo contiene valores mayores que el del nodo.

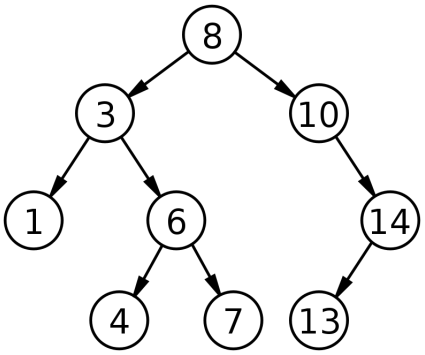

Figura 1: Ejemplo de Árbol Binario de Búsqueda

Un árbol binario con hojas vacías y con nodos interiores que pueden tomar valores enteros, se puede representar bajo la siguiente gramática:

$$\textit{Bintree} ::= () \mid (\text{Int } \textit{Bintree } \textit{Bintree})$$

Debe representar e implementar un **Árbol Binario de Búsqueda** en Racket usando:

1. Listas
2. Datatypes

Para cada una de las representaciones, debe realizar:

- Representación de un árbol vacío, *empty-bintree*

- Extractor del valor de la raíz: *current-element*, extractor del hijo izquierdo: *move-to-left-son*, extractor del hijo derecho: *move-to-right-son*.
- Proceso *number→bintree* que recibe un número y lo transforma a un árbol binario en la gramática utilizada, sin hijos inicialmente.
- Predicados: *empty-bintree?* (pregunta si es un árbol vacío), *at-leaf?* (pregunta si la posición actual es un árbol hoja, es decir, sin hijos), *bintree-with-at-least-one-child?* (si en donde se encuentra actualmente es un árbol con al menos un hijo).
- Función llamada *insert-to-left* que recibe un número y un bintree (árbol binario), e inserta dicho número, convertido en árbol binario, a la izquierda del árbol que es recibido como argumento. Su función hermana *insert-to-right* hace lo propio para la inserción del lado derecho.
- Una función llamada *bintree-order-validation* que reciba un árbol binario de búsqueda y valide su propiedad de orden, esto es, si en realidad se cumple la propiedad de orden de un árbol binario de búsqueda en el árbol binario dado. De ser así, la función debe retornar #t, de lo contrario, #f.
- Una función llamada *insert-element-into-bintree* que recibe un árbol binario de búsqueda y un número n, si el elemento n ya existe en el árbol dado, retorna el árbol, sino, debe insertar el número n en el lugar correcto del árbol donde debe estar (teniendo en cuenta la propiedad de orden).

Algunos ejemplos de las funcionalidades a implementar:

- *empty-bintree*:

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

- *current-element*:

```
> (current-element '(7 () ()))
7
```

- *move-to-left-son*:

```
> (move-to-left-son '(12 (1 () ()) (31 () ())))
(1 () ())
```

- *move-to-right-son*:

```
> (move-to-right-son '(12 (1 () ()) (31 () ())))
(31 () ())
```

- *number→bintree*:

```
> (number->bintree 93)
(93 () ())
```

- *empty-bintree?*:

```
> (empty-bintree? (move-to-left-son '(13 () ()))
#t
```

- *at-leaf?*:

```
> (at-leaf? '(7 () ()))
#t
```

- *bintree-with-at-least-one-child?*:

```
> (bintree-with-at-least-one-child? '(18 () (38 () ())))
#t
```

- *insert-to-left*:

```
> (insert-to-left 9 '(18 () ()))
(18 (9 () ()) ())
```

- *insert-to-right*:

```
> (insert-to-right 27 (insert-to-left 9 '(18 () ())))
(18 (9 () ()) (27 () ()))
```

- Arbol_Ejemplo =

```
'(8 (3 (1 () ()) (6 (4 () ()) (7 () ()))) (10 () (14 (13 () ()) ())))
```

- *bintree-order-validation*:

```
> (bintree-order-validation Arbol_Ejemplo)
#t
```

- *insert-element-into-bintree*:

```
> (insert-element-into-bintree Arbol_Ejemplo 2)
(8 (3 (1 () (2 () ())) (6 (4 () ()) (7 () ()))) (10 () (14 (13 () ()) ())))
```

Adicionalmente para la representación de **Datatypes** debe agregar:

- Su respectiva función *unparse*.
- Su respectiva función *parse*.

6. **(11 pts)** En algunas ocasiones, es útil especificar una sintaxis concreta como una secuencia de símbolos y enteros, rodeados de paréntesis. Por ejemplo, se podría definir un conjunto de *listas de prefijos*, o en inglés, *prefix lists* con las siguientes gramáticas:

$$
\begin{aligned}
\textit{Prefix-list} ::= & \ (\textit{Prefix-exp}) \\
& \ \text{pref-exp (pref)} \\
\textit{Prefix-exp} ::= & \ \text{Int} \\
& \ \text{const-exp (num)} \\
::= & \ \text{- } \textit{Prefix-exp Prefix-exp} \\
& \ \text{diff-exp (operand1 operand2)}
\end{aligned}
$$

De manera que:

```
(- - 3 2 - 4 - 12 7)
```

Es una expresión legal, de *prefix list*. A veces a esto se le llama la **Notación de prefijo polaca**, o en inglés **Polish prefix notation**. Se debe:

- Escribir la función *parse* que convierta una *prefix-list* a la sintaxis abstracta:

```
(define-datatype prefix-exp prefix-exp?
 (const-exp
  (num integer?))
 (diff-exp
  (operand1 prefix-exp?)
  (operand2 prefix-exp?)))
```

De manera que el ejemplo anterior produzca el mismo árbol de sintaxis abstracta que la secuencia de constructores:

```
(diff-exp
 (diff-exp
  (const-exp 3)
  (const-exp 2))
 (diff-exp
  (const-exp 4)
  (diff-exp
   (const-exp 12)
   (const-exp 7))))
```

Adicionalmente, debe escribir la función *unparse* correspondiente, y anexar un dibujo del árbol de sintaxis abstracta para el ejemplo:

```
(- - 3 2 - 4 - 12 7)
```

7. **(23 pts)** Dada la siguiente gramática:

```
;;  <programa>       ::= <expresion>
;;                   un-program (exp)
;;  <expresion>      ::= <numero>
;;                   num-lit (n)
;;                   ::= (<expresion>  <operacion> <expresion>)
;;                   exp-lit (exp1 op exp2)
;;                   ::= <identificador>
;;                   variable (id)
;;                   ::= var (identificador = <expresion>)*  in <expresion>
;;                   declaracion (ids exps cuerpo)
;;  <operacion>      := + - * /
;;                   primitiva
```

En un archivo en Racket, siguiendo la notación de la librería **SLLGEN**, usted debe:

- Definir los datatypes.
- Escribir la especificación léxica.

- Escribir la especificación gramática.
- Escribir la función $parse$.
- Escribir la función $unparse$.

Aclaraciones

1. El taller es en grupos de máximo tres (3) alumnos.

2. La solución del taller debe ser subida al campus virtual a más tardar el día **6 de Diciembre, 2019 (23:59)**. Se debe subir al campus virtual en el enlace correspondiente a este taller un archivo comprimido **.zip** que siga la convención *CódigodeEstudiante1-CódigodeEstudiante2-CódigodeEstudiante3-Taller2FLP20192.zip*. Este archivo debe contener los archivos: **bignum32.rkt, difftree.rkt, pilas-listas.rkt, pilas-procedimientos.rkt, pilas-datatypes.rkt, check-env.rkt, abb-listas.rkt, abb-datatypes.rkt, prefix.rkt y prefix.pdf o prefix.jpg / prefix.png (donde se muestre el arbol de sintaxis abstracta), y first-language.rkt (punto 7)** que contengan el desarrollo de los ejercicios.

3. Para todos los tipos de datos se debe **incluir la gramática que utilizó**.

4. En las primeras líneas de cada archivo deben estar comentados los nombres y los códigos de los estudiantes participantes.

5. Se debe incluir para cada procedimiento un comentario que explique lo que realiza cada función y para qué es empleada.

6. En caso de tener dudas, puede consultar con el profesor **Robinson Duque** (Martes 10-11am, Jueves 4-5pm, o con cita previa) o consultar con el monitor **Juan Marcos Caicedo** en el horario de atención de los Lunes de 2 pm a 4 pm (en el Laboratorio del Grupo de Investigación Avispa, 3er piso).

Entregas Tardías o por Otros Medios

1. Este taller **sólo se recibirá a través del campus virtual**. Adicionalmente, sólo se evaluarán los documentos solicitados en el punto 2 de la sección anterior. Cualquier otro tipo de correo o nota aclaratoria será descartado. Sólo se aceptan envíos por fuera del horario establecido bajo excusas de fuerza mayor validadas a través de la dirección de la Escuela.

2. Las entregas tarde serán penalizadas así: (-1pt) por cada hora de retraso o fracción. Por ejemplo, si usted realiza su entrega y el campus registra las 12:00 am (i.e., 1seg después de la hora de entrega), usted está incurriendo en la primer hora de retraso. Asegurese con mínimo dos horas de anticipación que el link de carga funciona correctamente toda vez que es posible incurrir en una entrega tardía debido a los tiempos de respuesta.