Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Práctica 2

Karla Ramírez Pulido karla@ciencias.unam.mx
J. Ricardo Rodríguez Abreu ricardo_rodab@ciencias.unam.mx
Manuel Soto Romero manu@ciencias.unam.mx

Fecha de inicio: 23 de agosto de 2017
Fecha de término: 6 de septiembre de 2017
Semestre 2018-1

Objetivos

- Entender el concepto de *Tipo de dato abstracto* (TDA).
- Aprender a implementar tipos de datos abstractos en Racket mediante la primitiva `define-type` de `plai`.
- Usar las funciones generadas por `plai` al definir tipos de datos abstractos para implementar las operaciones asociadas al TDA correspondiente.
- Usar las primitivas `type-case` y/o `match` para utilizar el mecanismo de apareamiento de patrones sobre tipos de datos abstractos.
- Usar tipos de datos abstractos para representar gramáticas.

Descripción general

La práctica consiste en definir distintos tipos de datos abstractos e implementar algunas funciones asociadas a los mismos. **Por cada definición de función se deben incluir al menos cinco pruebas unitarias**. No se permite modificar el contrato especificado para cada función ni usar primitivas que resuelvan directamente los ejercicios.

Archivos requeridos

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

- `practica2.rkt` archivo con la estructura para definir los tipos de datos abstractos e implementar las funciones asociadas.
- `test-practica2.rkt` archivo dónde se agregarán las pruebas unitarias de la práctica (al menos cinco por cada función definida).

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

Desarrollo de la práctica

Ejercicio 2.1 (2.5 pts.) Definir un tipo de dato abstracto `funcion` para trabajar con funciones matemáticas simples. El TDA debe contener:

- Un constructor `(x)` que representa a la variable independiente `x`. Nótese que no recibe parámetros.
- Un constructor `(cte n)` para representar constantes, `n` es un número entero.
- Un constructor `(sum f g)` que representa la suma de funciones, los parámetros `f` y `g` son de tipo `funcion`.
- Un constructor `(mul f g)` que representa el producto de funciones, los parámetros `f` y `g` son de tipo `funcion`.
- Un constructor `(div f g)` que representa la división de funciones, los parámetros `f` y `g` son de tipo `funcion`.
- Un constructor `(pot f n)` que representa funciones elevadas a una potencia `n`, el parámetro `f` es de tipo `funcion` y el parámetro `n` es un entero.

Una vez definido el TDA, definir las siguientes funciones:

1. Una función `funcion->string` que regresa una cadena de caracteres representando a la función que recibe como parámetro.

```
;; funcion->string: funcion -> string
(define (funcion->string f)
    ...)
```

```
> (funcion->string (mul (cte 2) (x)))
"(2 * x)"
```

2. Una función `evalua` que devuelve el resultado de evaluar la función pasada como parámetro con el valor `v`, es decir, regresa $f(v)$.

```
;; evalua: funcion -> funcion
(define (evalua f v)
    ...)
```

```
> (evalua (mul (cte 2) (x)) 1729)
(mul (cte 2) (cte 1729))
```

3. Una función `deriva` que regresa la derivada de una función.

```
;; deriva: funcion -> funcion
(define (deriva f)
    ...)
```

```
> (deriva (mul (cte 2) (x)))
(sum (mul (cte 2) (cte 1)) (mul (x) (cte 0)))
```

–

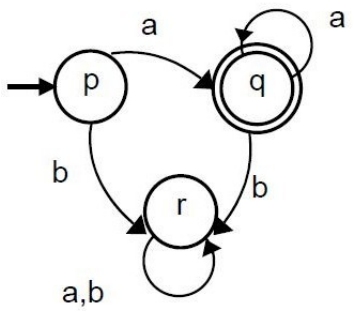
Ejercicio 2.2 (2.5 pts.) Un Autómata Finito Determinista (AFD) se define formalmente como la quintupla: $(Q, \Sigma, q_0, \delta, F)$ dónde:

- Q es un conjunto de estados.
- Σ es un alfabeto.
- $q_0 \in Q$ es un estado inicial.
- $\delta : Q \times \Sigma \to Q$ es una función de transición.
- $F \subseteq Q$ es un conjunto de estados finales.

Definir un tipo de dato abstracto `automata` para trabajar con autómatas finitos deterministas. El TDA debe contener un único constructor (`afd estados alfabeto inicial transicion finales`) dónde:

- `estados` es el conjunto de estados y se representará por una lista de símbolos.
- `alfabeto` es el alfabeto y se representará por una lista de símbolos.
- `inicial` es el símbolo inicial y se representará por un símbolo.
- `transicion` es la función de transición y se representará por una función (`procedure`).
- `finales` es el conjunto de estados finales y se representará por una lista de símbolos.

Por ejemplo, el siguiente autómata



se debe poder construir como:

```
(afd '(p q r) '(a b) 'p transicion '(q))
```

La función `transicion` se define como sigue[2]:

```
;; Función de transición.
;; transicion: symbol symbol -> symbol
(define (transicion estado simbolo)
   (match estado
      ['p (if (symbol=? 'a simbolo) 'q 'r)]
      ['q (if (symbol=? 'a simbolo) 'q 'r)]
      ['r 'r]))
```

Una vez definido el TDA, definir las siguientes funciones:

1. Una función `verifica` que dado un AFD, verifica que esté bien construido. Esto es, debe verificar que el símbolo inicial y el conjunto de símbolos finales pertenezcan o estén contenidos en el conjunto de estados.

   ```
   ;; verifica: automata -> boolean
   (define (verifica atm)
      ...)
   ```

   ```
   > (verifica (afd '(p q r) '(a b) 'p d '(q)))
   #t
   ```

2. Una función `acepta?` que dado un AFD y una lista de símbolos que representan una expresión, determina si es aceptada por el autómata.

   ```
   ;; acepta?: automata (listof symbol) -> boolean
   (define (acepta? atm lst)
      ...)
   ```

   ```
   > (acepta? (afd '(p q r) '(a b) 'p d '(q)) '(a a))
   #t
   ```

–

Ejercicio 2.3 (2.5 pts.) Definir un tipo de dato abstracto `arreglo` para trabajar con arreglos.

El TDA debe contener:

- Un constructor `(arrg tipo dim elems)` que permite definir un arreglo. El parámetro `tipo` es un predicado para identificar el tipo de los elementos, el parámetro `dim` sirve para definir el tamaño del arreglo y `elems` es una lista de tamaño `dim` y con elementos de tipo `tipo`.

[2]Se podría tener una definición más robusta. Esta definición se da a manera de ejemplo.

- Un constructor `(agrega-a e a i)` para representar la operación de agregar un elemento `e` en el `arreglo a` en la posición `i`.
- Un constructor `(obten-a a i)` para representar la operación de obtener el elemento en la posición `i` del `arreglo a`.

Una vez definido el TDA, se debe de definir una función `(calc-a arreglo)` que evalúe expresiones del tipo `arreglo`.

```
;; calc-a: arreglo -> arreglo
(define (calc-a arr)
   ...)
```

```
> (define a (arrg number? 5 '(1 2 3 4 5)))
> (calc-a a)
(arrg number? 5 '(1 2 3 4 5))
> (define b (arrg boolean? 4 '(1 2 3 4)))
> (calc-a b)
error: Los elementos no son del tipo especificado.
> (define c (arrg boolean? 3 '(#t #t #t #f)))
> (calc-a c)
error: Dimensión inválida
> (calc-a (agrega-a 2 a 4))
(arreglo number? 5 '(1 2 3 4 2))
> (calc-a (agrega-a 2 a 7))
error: Índice inválido
> (calc-a (agrega-a #t a 7))
error: Los elementos no son del tipo especificado.
> (calc-a (obten-a 2 a))
3
> (calc-a (obten-a -1 a))
error: Índice inválido
```

—

Ejercicio 2.4 (2.5 pts.) Definir un tipo de dato abstracto `conjunto` para trabajar con conjuntos. El TDA debe contener:

- Un constructor `(cjto l)` para representar un conjunto, dónde `l` es una lista.
- Un constructor `(esvacio? c)` el cual representa el predicado que indica si el conjunto `c` es vacío.
- Un constructor `(contiene? c e)` para representar el predicado que indica si el conjunto `c` contiene al elemento `e`.

- Un constructor `(agrega-c c e)` el cual representa la operación para agregar el elemento `e` en el conjunto `c`.
- Un constructor `(union c1 c2)` para representar la operación de `union` entre dos conjuntos `c1` y `c2`.
- Un constructor `(interseccion c1 c2)` el cual representa la operación de intersección entre dos conjuntos `c1` y `c2`.
- Un constructor `(diferencia c1 c2)` el cual representa la operación de diferencia entre los conjuntos `c1` y `c2`.

Una vez definido el TDA, definir una función `calc-c` que evalúe expresiones del tipo `conjunto`. La función debe mantener el conjunto siempre sin elementos repetidos.

```
;; calc-c: conjunto -> conjunto
(define (calc-c cto)
   ...)
```

```
> (define a (cjto '(1 7 2 9)))
> (define b (cjto '(1 2 3 4))
> (calc-c (cjto '(1 1 7 2 9))
(cjto '(1 7 2 9))
> (calc-c (esvacio? a))
#f
> (calc-c (contiene? a 1))
#t
> (calc-c (agrega a 9))
(cjto '(1 7 2 9))
> (calc-c (union a b))
(cjto '(1 7 2 9 3 4))
> (calc-c (interseccion a b))
(cjto '(1 2))
```

Puntaje total: *10 puntos*

Entrega

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

- `practica2.rkt`
- `test-practica2.rkt`

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

Puntos extra

Escoger alguno de los siguientes ejercicios y resolverlo **individualmente**[3]:

Punto extra 2.1 (1 pt.) Investigar quién es Bárbara Liskov, cuales son sus aportes a la abstracción de datos y su relación con la Teoría de Lenguajes de Programación. Escribir un ensayo de al menos dos cuartillas. El ensayo debe incluir: Título, Introducción, Desarrollo, Conclusiones y Bibliografía. No se tomará en cuenta ningún ensayo que no esté debidamente citado e incluya las referencias correspondientes. El ensayo deberá entregarse de forma impresa a más tardar el 6 de septiembre de 2017 durante la sesión de laboratorio.

Punto extra 2.2 (1 pt.) Dar la definición de un tipo de dato abstracto para trabajar con montículos mínimos y definir una función `agrega` para agregar elementos en el montículo y otra llamada `saca` que obtiene el elemento mínimo del montículo. El ejercicio debe enviarse a más tardar el 6 de septiembre de 2017 en un archivo `cuenta_p2.rkt` donde `cuenta` es el número de cuenta del alumno, al correo `manu+ldp@ciencias.unam.mx` con el asunto `[LDP-Extra P2]`.

Referencias

Algunas referencias de consulta:

[1] Karla Ramírez, Manuel Soto, Ricardo Rodríguez, *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] Eric Tánter, *PREPLAI: Scheme y Programación Funcional*, Primera edición, 2014. Disponible en: [`http://users.dcc.uchile.cl/~etanter/preplai/`] (Consultado el 4 de agosto de 2017).

[4] Matthias Felleisen, Robert Findler, Matthew Flatt, Shriram Krishnamurthi, *How to Design Programs*, Segunda Edición, The Mit Press, 2017. Disponible en: [`http://www.ccs.neu.edu/home/matthias/HtDP2e/`] (Consultado el 4 de agosto de 2017).

[5] Matthias Felleisen, David Van Horn, Conrad Barski, *Realm Of Racket*, Primera edición, No Starch Press, 2013.

[3] Pueden enviarse ambos ejercicios para ser revisados, sin embargo, sólo uno será tomado en cuenta.