UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

ESTRUCTURAS DISCRETAS 2020-1

Profesor: M. en C. I. C. Odín Miguel Escorza Soria
Ayudante de laboratorio: Salazar González Edwin Max

Práctica 4.

EXPRESIONES ARITMÉTICAS

CLASE *Show*

1. OBJETIVOS

- Introducir la creación de tipos de datos en Haskell.
- Reforzar el conocimiento sobre expresiones aritméticas.
- Breve introducción de la clase *Show*.

El principal objetivo de la práctica 1 es reforzar algunos temas de los lenguajes formales, como las expresiones.

2. CARACTERÍSTICAS

- La especificación de tipos de datos en Haskell puede ser a través de *Data* o *Type*. En esta práctica nos enfocaremos en *Data*.
- Las instancias de la clase *Show* se usan para poder representar visualmente un dato en terminal.

Tiempo de entrega: 2 horas

Prerequisitos:

- Entorno del lenguaje de programación Haskell.
- Uso del intérprete de Haskell.
- Expresiones aritméticas.
- Operadores unarios, binarios y n-arios.
- Notacion prefija, infija y sufija.

3. DESARROLLO

Un lenguaje formal está definido por expresiones bien formadas obtenidas a través de la combinación de elementos de un conjunto de símbolos. Las expresiones bien formadas son todas las expresiones gramaticalmente correctas del lenguaje formal [2].

3.1. Expresiones

Una expresión está formada por la combinación de uno o más operadores con uno o más operandos.
Un operador se define como operaciones que se deben realizar con los operandos [4].
"Una expresión puede ser aritmética, lógica, de carácter o de cadena, en cuyo caso su evaluación regresa un número, un valor lógico (verdadero o falso), un carácter o una cadena, respectivamente" [4].

Por lo tanto una Expresión Aritmética (EA) está formada por operandos y operadores y su evaluación regresa un número.

Se tienen tres diferentes maneras para escribir una expresión aritmética, las cuales se definen en función del operador [1]:

- Notación prefija: se le conoce así al tipo de operación cuyo operador se encuentra antes del o los operandos.
  Ejemplo: + 7 10.
- Notación infija: se le conoce así al tipo de operación cuyo operador se encuentra entre los operandos. Esta notación es la más conocida dado que es con la que se trabaja normalmente.
  Ejemplo: 7 + 10.
- notación sufija: se le conoce así al tipo de operación cuyo operador se encuentra después del o los operandos.
  Ejemplo: 7 10 +.

Los operadores pueden definirse como [4]:

- Operadores monarios o unarios: los cuales se aplican únicamente sobre un operando.
  Ejemplo: − 8.
- Operadores binarios: los cuales se aplican sobre dos operandos.
  Ejemplo: 4 ∗ 8.
- Operadores n-narios: los cuales se aplican sobre n operandos.
  Ejemplo: + 2 54 100 2.

3.2. Mecanismos para descripción de expresiones

La descripción de expresiones aritméticas se pueden hacer de diferentes formas, una opción viable es tener un conjunto de reglas que garanticen la correcta construcción de expresiones aritméticas.
Por lo que definimos una EA como [1]:

1. "Un objeto elemental: un número".
2. "Si $\triangledown$ es un operador unario y E es una expresión aritmética, entonces $\triangledown$ E es una expresión aritmética".
3. "Si $\diamond$ es un operador binario infijo y E y F son dos expresiones aritméticas, entonces E$\diamond$F es una expresión aritmética".
4. "Si $\star$ es un operador n-ario y E_1, E_2, ... , E_n son expresiones aritméticas, entonces $\star(E_1, E_2, \ldots, E_n)$ es una expresión aritmética".
5. "Éstas y solo éstas son expresiones aritméticas".

Para desarrollar expresiones aritméticas bien construidas se hace uso de la gramática obtenida a través de las siguientes producciones (ver tabla 1) [1, 6]. Las producciones siguien la siguiente estructura: *símbolo ::= expresión con símbolos*[1]

[1]Notación Backus-Naur (BNf, por sus siglas en inglés).
Los meta símbolos pueden ser [5]:

- ::= significado: se define como. El elemento de la derecha desarrolla el elemento de la izquierda.
- | significado: o. Se puede elegir solamente uno de los elementos que separa.

S	::=	E	1.1	*significado*
E	::=	cons	1.2	constantes.
E	::=	var	1.3	variables.
E	::=	(E)	1.4	parentización de operadores.
E	::=	■E	1.5	operador unario.
E	::=	▸(F F)	1.6	operador n-nario.
E	::=	E⊡E	1.7	operador biario.
F	::=	E F | E	1.8	produce n operandos.
■	::=	+ | -	1.9	operador signo más y menos, respectivamente.
▸	::=	+ | - | *	1.10	operador suma, resta y multiplicación, respectivamente.
⊡	::=	+ | − | ∗ | / | *mod* | ∗ ∗	1.11	operador suma, resta, multiplicación, división, módulo y potencia, respectivamente
var	::=	a | b | c | ...	1.12	
cons	::=	0 | 1 | 2 | 7 | 3.1416 | ...	1.13	

Cuadro 1: Tabla de producciones para EA.

4. EJEMPLO

Se tiene la siguiente gramática (ver tabla 2) que produce expresiones aritméticas usando únicamente el operador suma:

W	::=	E	*significado*
E	::=	cons	constantes.
E	::=	E▸E	operador biario.
▸	::=	+	operador suma
cons	::=	0 | 1 | 2 | 7 | 3.1416 | ...	

Cuadro 2: Tabla de producciones para EA que usan únicamente el operador suma.

```
1 data EASuma = C Int | Suma EASuma EASuma
```

Código 1: Gramática que produce EA usando únicamente el operador suma

En la línea 1 del código (ver código 1) se define la gramática especificada en la tabla 2. El primer constructor (C Int) del tipo de dato EASuma representa los valores constantes mientras que el segundo constructor representa el operador binario suma.

```
2 suma :: Int -> Int  -> EASuma
3 suma n m = Suma (C n) (C m)
```

Código 2: Función *suma*

La línea 2 del código 2 especifica la firma de la función *suma* que recibe dos enteros y regresa la representación de la suma con el tipo de dato EASuma, en la siguiente línea se usan los parámetros de la función para regresar el segundo constructor (Suma EA EA) del tipo de dato EASuma.

```
> suma 2 8
Suma (N 2) (N 8)
```

Código 3: Ejemplo ejecucion funcion *suma*

5. INSTRUCCIONES

Descarga el archivo Practica4.hs y resuelve los ejercicios definidos sobre éste.

6. EJERCICIOS

- Ejercicio 1: resolver la función *pasarsumaEA* que recibe dos enteros y regresa su representación del tipo de dato EA con el constructor Suma EA EA.

```
> pasarsumaEA 3 7
Suma (N 3) (N 7)
```

Código 4: Ejemplo de llamada función *pasarsumaEA*.

- Ejercicio 2: resolver la función *pasarrestaEA* que recibe dos enteros y regresa su representación del tipo de dato EA con el constructor Resta EA EA.

```
> pasarrestaEA 4 8
Resta (N 4) (N 8)
```

Código 5: Ejemplo de llamada función *pasarrestaEA*.

- Ejercicio 3: resolver la función *pasarmultEA* que recibe dos enteros y regresa su representación del tipo de dato EA con el constructor Mult EA EA.

```
> pasarmultEA 5 9
Mult (N 5) (N 9)
```

Código 6: Ejemplo de llamada función *pasarmultEA*.

- Ejercicio 4: resolver la función *pasardivEA* que recibe dos enteros y regresa su representación del tipo de dato EA con el constructor Div EA EA.

```
> pasardivEA 2 6
Div (N 2) (N 6)
```

Código 7: Ejemplo de llamada función *pasardivEA*.

- Ejercicio 5: resolver la función *pasarmoduloEA* que recibe dos enteros y regresa su representación del tipo de dato EA con el constructor Modulo EA EA.

```
> pasarmoduloEA 3 9
Modulo (N 3) (N 9)
```

Código 8: Ejemplo de llamada función *pasarmoduloEA*.

- Ejercicio 6: resolver la función *pasarpotEA* que recibe dos enteros y regresa su representación del tipo de dato Ea con el constructor Pot EA EA.

```
> pasarpotEA 3 3
Pot (N 3) (N 3)
```

Código 9: Ejemplo de llamada función *pasarpotEA*.

- Ejercicio 7: definir la instancia *Show* para el tipo de dato EA de tal manera que las instancias creadas estén en notación infija.
  `Nota: en esta parte deberán quitar` $import(Eq, Show)$ `que viene declarado al final del tipo de dato EA`

```
> (N 8)
8
> Negativo (N 8)
(-8)
> Mult (N 2) (N 3)
2 * 3
> Pot (N 2) (N 3)
2 ^ 3
```

Código 10: Ejemplos de instancias de la clase *Show*

- Ejercicio 8 (2pts): resolver la función *mayorqueUnario* que recibe dos EA cuyo operador principal de cada uno es un operador unario y esta aplciado a un número. La función regresa *True* en caso de que el primer parámetro sea mayor que el segundo, regresa *False* en otro caso.

```
> mayorqueUnario (Negativo (N 8)) (Positivo (N 6))
False
> mayorqueUnario (Positivo (N 8)) (Negativo (N 6))
True
> mayorqueUnario (N 8) (N 6)
*** Exception: Es comparacion entre expresiones unarias
CallStack (from HasCallStack):
  error, called at Practica1.hs:74:22 in main:Practica1
```

Código 11: Ejemplos de llamada función *mayorqueUnario*.

- Ejercicio 9 (2pts, 2cp): resolver la función *eval* que recibe una EA y regresa su evaluación.

```
> eval (Div (Suma (Negativo (Modulo (N 10) (N 3))) (Pot (N 2) ↩
   (N 3))) (Resta (Positivo (N 10)) (N 8)))
3
> eval (Mult (Suma (N 6) (N 6)) (Resta (N 12) (N 2)))
120
> eval (Div (Suma (N 6) (N 6)) (Modulo (N 12) (N 2)))
*** Exception: Cuidado con las divisiones entre 0
CallStack (from HasCallStack):
  error, called at Practica1.hs:88:46 in main:Practica1
```

Código 12: Ejemplos de llamada función *eval*.

- Ejercicio para inasistentes a clase y Ejercicio Extra: no aplica para esta practica.

7. ESPECIFICACIONES

✓ Respetar las firmas de las funciones. Cambiarlas podría ser motivo de anulación del ejercicio.

× Cualquier plagio de prácticas será evaluado con 0, sin hacer indagaciones. **Creen** su propio código.

× Cualquier práctica entregada posterior a la fecha límite no será tomada en cuenta.

Se deberá contar con un difectorio cuyo nombre sea Practica4. Dentro del directorio se deben tener:

- README.txt, donde se incluya nombre y número de cuenta de los integrantes del equipo junto con comentarios que crean pertinentes sobre la práctica.
- Practica4.hs, script requerido para esta práctica.

Compirmir el directorio con el formato **ApellidoNombreP4**. Comprimir con extensión .tar.gz o .zip

Solamente un integrante del equipo deberá enviar la práctica.

Enviar la práctica al correo ciclomax9@ciencias.unam.mx con el asunto `[LC-Apellido-Nombre-P4]`.

"Documentar tu código es como limpiar tu baño; nunca quieres hacerlo pero realmente crea una experiencias más placentera para ti y tus invitados." - Ryan Campbell

Suerte ☺

Referencias

[1] Miranda, Favio y Viso, Elisa, *Matemáticas Discretas*. México DF, México: Las prensas de ciencias, 2010.

[2] Solís D., Julio E. y Torres F., Yolanda, *Lógica Matemática*. México: Unidad Autónoma Metropolitana, 1995.

[3] Sergio Balari, *Teoría de Lenguajes Formales*. España: Universidad Autónoma de Barcelona, 2014.

[4] Pes, Carlos, *Diseño de Algoritmos en Pseudocódigo y Ordinogramas*. CA, USA: Tutorial de Algoritmos de Abrirllave, 2017.

[5] Knuth, Donald E., *Backus Normal Form vs. Backus Naur Form*. Comm. ACM, 1964.

[6] Berna, Ramón *Autómatas y lenguajes*. México: Tec de Monterrey, 2003.