Práctica 1
Implementación de `Postfix`

Favio E. Miranda Perea (favio@ciencias.unam.mx)
Diego Carrillo Verduzco (dixego@ciencias.unam.mx)
Pablo G. González López (pablog@ciencias.unam.mx)

Miércoles 15 de agosto de 2018

Fecha de entrega: Domingo 26 de agosto de 2018 a las 23:59:59.

`Postfix` es un lenguaje simple basado en una pila inspirado en el lenguaje gráfico *PostScript*, el lenguaje de programación *Forth*, y las calculadoras *Hewlett Packard*.

1 Sintaxis

La unidad sintáctica básica de un programa en `Postfix` es el **comando**. Los comandos son de la siguiente forma:

- Una literal entera.
- Una de las siguientes palabras reservadas: `add`, `div`, `eq`, `exec`, `gt`, `lt`, `mul`, `nget`, `pop`, `rem`, `sel`, `sub`, `swap`.
- Una secuencia ejecutable. Una secuencia parentizada de comandos que sirve como una subrutina.

Para representar estos comandos en *Haskell* usaremos la siguiente definición:

```
data Command = I Int | ADD | DIV | Eq | EXEC | Gt | Lt
             | MUL | NGET | POP | REM | SEL | SUB | SWAP
             | ES [Command] deriving (Show, Eq)
```

Un **programa** en `Postfix` es una secuencia parentizada que consiste en (1) la palabra reservada `postfix` seguida de (2) un número natural que indica el número de argumentos que recibe el programa seguido de (3) cero o más comandos.

Los programas los representaremos como una tupla de 3 elementos. Primero debemos definir el constructor de la palabra reservada `postfix`.

```
data PF = POSTFIX deriving (Show, Eq)
```

Y finalmente definimos la representación de los programas.

```
type Program = (PF, Int, [Command])
```

2 Semántica

El significado de un programa en `Postfix` se determina ejecutando sus comandos en orden de izquierda a derecha. Cada uno de los comandos antes mencionados manipula una pila de valores que inicialmente contiene el número de argumentos del programa. Cabe resaltar que esta pila de valores inicial debe contener solo literales enteras, no puede contener secuencias ejecutables.

La pila de valores la representaremos del siguiente modo:

```
type Stack = [Command]
```

Ya que tenemos definidos todos los elementos que construyen programas en `Postfix` comenzaremos a implementar las funciones que los ejecutarán. Hay que tener en mente que durante la ejecución de los programas, si se esta trabajando con una pila inapropiada, se pueden generar errores. Estos errores deben ser manejados por la función que los genera y deben informar explícitamente al usuario cuál fue la causa de ese error.

1. (1.5 puntos) `arithOperation`. Función que realiza las operaciones de los comandos aritméticos. (`add`, `div`, `eq`, `gt`, `lt`, `mul`, `rem`, `sub`)

   ```
   arithOperation :: Command -> Command -> Command -> Command
   ```

   Ejemplo:

   ```
   *Main> arithOperation (I 1) (I 2) ADD
   I 3
   ```

2. (1.5 puntos) `stackOperation`. Función que realiza las operaciones de los comandos que alteran la pila de valores. (literal entera, `nget`, `pop`, `sel`, `swap`, secuencia ejecutable)

   ```
   stackOperation :: Stack -> Command -> Stack
   ```

   Ejemplo:

   ```
   *Main> stackOperation [I 1, I 5] SWAP
   [I 5, I 1]
   *Main> stackOperation [I 1, I 5] (ES [I 3, ADD, SWAP, I 2])
   [ES [I 3, ADD, SWAP, I 2], I 1, I 5]
   ```

3. (1 punto) `execOperation`. Función que devuelve la lista de comandos y la pila resultante de realizar la llamada a la operación `exec`.

   ```
   execOperation :: [Command] -> Stack -> ([Command], Stack)
   ```

   Ejemplo:

   ```
   *Main> execOperation [ADD] [ES [I 1, ADD], I 2, I 3]
   ([I 1, ADD, ADD], [I 2, I 3])
   ```

4. (1 punto) `validProgram`. Función que determina si la pila de valores que se desea ejecutar con un programa es valida.

```
validProgram :: Program -> Stack -> Bool
```

Ejemplo:

```
*Main> validProgram (POSTFIX, 0, [I 1, I 2, ADD]) []
True

*Main> validProgram (POSTFIX, 2, [ADD]) [I 3, ES [I 1, I 2, SUB]]
False
```

5. (4 puntos) `executeCommands`. Función que dada una lista de comandos y una pila de valores obtiene la pila de valores resultante después ejecutar todos los comandos.

```
executeCommands :: [Command] -> Stack -> Stack
```

Ejemplo:

```
*Main> seq = [I (-1), I 2, ADD, I 3, MUL]
*Main> executeCommands seq []
[I 3]
*Main> seq = [ES [I 2, MUL], EXEC]
*Main> executeCommands seq [I 7]
[I 14]
```

6. (1 punto) `executeProgram`. Función que ejecuta cualquier programa en `Postfix`.

```
executeProgram :: Program -> Stack -> Int
```

Ejemplo:

```
*Main> prg = [ES [I 0, SWAP, SUB], I 7, SWAP, EXEC]
*Main> executeProgram (POSTFIX, 0, prg) []
[I (-7)]
*Main> prg = [POP]
*Main> executeProgram (POSTFIX, 1, prg) [I 4, I 5]
*** Exception: Wrong number of arguments.
*Main> prg = [I 4, MUL, ADD]
*Main> executeProgram (POSTFIX, 1, prg) [I 3]
*** Exception: Not enough numbers to add.
*Main> prg = [I 4, SUB, DIV]
*Main> executeProgram (POSTFIX, 2, prg) [I 4, I 5]
*** Exception: Divide by zero.
*Main> prg = [I 2, MUL]
*Main> executeProgram (POSTFIX, 0, prg) []
*** Exception: Final top of stack is not an integer.
```

3 Anexo: Semántica de los comandos

- N: Agrega la literal entera N a la pila.
- `sub`: Si el primer (tope) y segundo elementos de la pila son v_1, v_2 respectivamente, eliminarlos de la pila y agregar $v_2 - v_1$ en su lugar. Si hay menos de dos valores en la pila o los dos primeros valores no son números enteros, lanzar un error. Los demás operadores aritméticos (`add` (suma), `mul` (multiplicación), `div` (división entera), y `rem` (residuo de la división entera)) se comportan de manera similar. `div` y `rem` lanzan un error si v_1 es cero.
- `lt`: Si el primer (tope) y segundo elementos de la pila son v_1, v_2 respectivamente, eliminarlos de la pila. Si $v_2 < v_1$ agregar un 1 a la pila, en otro caso agregar un 0 a la pila. Los demás operadores de comparación (`eq` (igualdad), `gt` (mayor que)) se comportan de manera similar. Si hay menos de dos valores en la pila o los dos primeros valores no son números enteros, lanzar un error.
- `pop`: Elimina el primer elemento de la pila. Lanza un error si esta es vacía.
- `swap`: Intercambia los primeros dos valores de la pila. Lanza un error si la pila tiene menos de dos valores.
- `sel`: Si el primer (tope), segundo y tercer elemento de la pila son v_1, v_2, v_3 respectivamente, eliminarlos de la pila. Si v_3 es el número cero, agregar v_1 a la pila; si v_3 es un número distinto de cero, agregar v_2 a la pila. Lanza error si la pila no contiene al menos tres valores, o si v_3 no es un número.
- `nget`: Llamemos v_{index} al tope de la pila y v_1,...,v_n a los elementos restantes en orden siendo v_n el que está en el fondo. Eliminar v_{index} de la pila. Si v_{index} es un número i tal que $1 \leq i \leq n$ y v_i es un número, agregar v_i a la pila. Lanza un error si la pila no contiene al menos un valor, si v_{index} no es un número, si i no esta en el rango $[1..n]$, o si v_i no es un número.
- (C_1 ... C_n): Agrega la secuencia ejecutable como un solo valor en la pila. Las secuencias ejecutables se usan en conjunción con `exec`.
- `exec`: Eliminar la secuencia ejecutable del tope de la pila y agregar en orden sus comandos al inicio de la secuencia de comandos actualmente en ejecución. Lanza un error si la pila es vacía o si el primer valor no es una secuencia de comandos.

¡Suerte!