Introducción

Esta tarea consiste en la implementación de un intérprete del lenguaje de programación Awk. Se implementará una versión reducida del lenguaje que llamaremos Awki (se pronuncia "oki")

Un programa Awki se compone de una lista de directivas (llamadas *reglas*) de la forma:

```
patrón { acción }
```

donde `patrón` es una condición booleana y `acción` es un fragmento de código ejecutable.

Un programa Awki lee de la entrada estándar y escribe en la salida estándar. Para cada línea de la entrada se evalúa cada patrón del programa; si el patrón es verdadero se ejecuta la acción asociada. El proceso anterior se puede especificar a través del siguiente pseudo código

```
Para cada línea de la entrada {
    Para cada regla (patrón,acción) del programa {
        if cumple patrón
           then ejecutar accion sobre línea
    }
}
```

Descripción de Awki

Explicaremos informalmente los diferentes componentes del lenguaje Awki y daremos la especificación de su sintaxis abstracta en Haskell.

Awki utiliza una sintaxis estilo C para las expresiones e instrucciones. Es además un lenguaje no tipado donde los datos básicos son las *cadenas* (*string*) y *números enteros*.

Como es habitual en los lenguajes de scripting, una cadena puede ser convertida automáticamente a número y viceversa si el contexto así lo requiere.

Expresiones

Los valores atómicos son las cadenas y los números que se representan de la manera usual:

```
"hola gente" ""
123
+4
-342
```

Los operadores de Awki por orden de precedencia decreciente son:

- `$` : Referencia de campo (se explica más abajo)
- `++ --` : Incremento y decremento (pre y pos)
- `+ - !` : más y menos unarios, negación
- `!` : negación lógica
- `* / %` : producto, división, módulo
- `+ -` : suma resta
- ‘ ‘ : concatenación de cadenas. El operador de concatenación es el espacio. Más adelante se explica más detalladamente.
- `< <= == != > >=` : operadores relacionales
- `&&` : *and* lógico
- `||`: *or* lógico
- `= += -= *= %=` : asignación.

En cuanto a la asociatividad: los operadores de asignación asocian de derecha a izquierda; los operadores relacionales no son asociativos; el resto de los operadores binarios asocian de izquierda a derecha.

Representación de valores

En base a lo anterior se define el siguiente tipo para representar las constantes en *Haskell*:

```
data Valor = Num Int    -- numero
           | Str String -- cadena
```

Como ya explicamos los valores serán tratados como números o cadenas según lo requiera el contexto. Para ello definimos operaciones de conversión.

Funciones de conversión

Para la conversión a cadena definimos una instancia de la clase `Show`:

```
instance Show Valor where
     show (Str xs) = xs
     show (Num m)  = show  m
```

Para la conversión a entero, definimos un par de funciones. La función `toIntMb` es una función parcial que solo retorna un entero si la cadena representa efectivamente un número.

```
toIntMb :: Valor -> Maybe Int
toIntMb (Num n) = Just n
toIntMb (Str xs) =
    case reads xs of
      [(n,"")] -> Just n
      _        -> Nothing
```

Sin embargo, Awki fuerza el caso `Nothing` a ser el entero 0 en contextos numéricos; esto es: cuando el valor aparece como operando de alguna operación aritmética. Se define entonces, la función `toInt` que se comporta de esa manera:

```
toInt :: Valor -> Int
toInt = fromMaybe 0 . toIntMb
```

Operaciones Aritméticas

Awki permite realizar las operaciones aritméticas habituales de los números enteros. En Haskell declaramos instancias de la clase `Num` para el tipo `Valor`:

```
instance Num Valor where
    a + b = Num (toInt a + toInt b)
    a * b = Num (toInt a * toInt b)
    a - b = Num (toInt a - toInt b)
    abs = Num . abs . toInt
    signum = Num . signum . toInt
    fromInteger = Num . fromInteger
```

Lo anterior nos permite utilizar los operadores aritméticos con expresiones del tipo Valor. Incluso es posible utilizar constantes enteras como si fueran del tipo Valor:

```
a :: Valor
a = 11

b :: Valor
b =  a + 1
```

Concatenación

La operación básica que se realiza con cadenas es la concatenación. Awki no tiene un operador para esta operación si no que se representa escribiendo los valores uno a continuación del otro, separados por un espacio si se requiere:

```
c = "hola" " que tal" ;
c = c c "chau";
```

La implementación en Haskell es:

```
concatv :: Valor -> Valor -> Valor
concatv v1 v2 = Str $ show v1 ++ show v2
```

Comparación de valores

Los operadores relacionales permiten comparar dos valores. Las comparaciones entre cadenas o entre números se realizan de la manera habitual. Cuando queremos comparar una cadena contra un número se hará de la siguiente manera: Si la cadena es convertible a entero (por `toIntMb`), se comparan los números; en otro caso se comparan ambos valores como cadenas.

Lo anterior lo implementamos como instancias de las clases `Eq` y `Ord`:

```
instance Eq Valor where
    a == b =
        case (toIntMb a, toIntMb b) of
            (Just a1 , Just b1) -> a1 == b1
            _                   -> show a == show b


instance Ord Valor where
    compare a b =
        case (toIntMb a, toIntMb b) of
            (Just a1 , Just b1) -> compare a1 b1
            _                   -> compare (show a) (show b)
```

Conversión a Booleanos

Como es habitual en este tipo de lenguajes la cadena nula y el número 0 son interpretados como False en un contexto booleano Mientras que una cadena no nula y un número distinto de 0 se interpretan como True. Esto ocurre cuando se aplican los operadores lógicos: `&&`, `||`, `!`. Notar que la cadena “0” se interpreta como True, mientras que el número 0 es False.

```
toBool :: Valor -> Bool
toBool (Num m) = m /= 0
toBool (Str xs) = not (null xs)
```

Las variables

Las variables en Awki se representan por identificadores tal como en el lenguaje C. Una referencia a una variable no inicializada no es un error: se considera que su valor es 0 o "" según el contexto.

Existen variables especiales a las cuales Awki asigna automáticamente. Por tal razón son llamadas variables *automáticas*. En general son variables que se asignan cada vez que se ingresa una línea de la entrada. Awki considera que la entrada esta formada por registros los que a su vez están formados por campos. Los registros se corresponden con las líneas de la entrada, mientras que los campos son las palabras que integran una línea. Más precisamente se considera que los campos están separados por una secuencia de espacios y/o tabuladores.

- `NR` - Indica el número de la línea o de registro.
- `NF` - Indica la cantidad de *campos* que tiene la línea.
- `$0` - Es el contenido de la línea.
- `$i` - Es el valor del campo i-ésimo. El índice `i` puede ser una expresión que requiere ser evaluada. En caso de ser una expresión compleja debe ir entre paréntesis debido a la precedencia del operador `$`.

Por ejemplo, si se ingresa la tercera línea de la entrada y esta es: `"hola que tal  12 34"` Tendremos los siguientes valores para las variables automáticas:

```
NR    3
NF    5
$1    "hola"
$2    "que"
$3    "tal"
$4    "12"
$5    "34"
```

Otros ejemplos referencias a campos:

```
$NF         "34"    (el último campo)
$(NF - 1)   "12"    (el penúltimo campo)
```

Representación de las expresiones

Se define el siguiente tipo para representar las expresiones en Haskell:

```
data Expr
    = Lit Valor                   -- valor simple
    | Var String                  -- variable
    | Op2 BOp Expr Expr           -- operación binaria,   a + b
    | Op1 UOp Expr                -- operación unaria,    -a
    | Assign String Expr          -- asignación           x = e
    | Accum BOp String Expr       -- asignación ac.,      x += e
    | PP Bool       --   prefijo
         Bool       --   incremento
         String     --   variable
    | Field Expr

-- operadores binarios
data BOp = Add | Sub | Mul | Div | Mod
         | Lt  | Gt  | Le  | Ge  | Ne | Equal
         | And | Or  | Concat

-- operadores unarios
data UOp =  Plus | Minus | Not
```

Cada constructor se explica por sí mismo con la ayuda del comentario que acompaña cada línea.

Algunos ejemplos para ilustrar el constructor `PP` que representa los operadores de incremento/decremento en sus dos modalidades pre y posfija:

```
++x   ---->  PP True True "x"
x++   ---->  PP False True "x"
--x   ---->  PP True False "x"
x--   ---->  PP False False "x"
```

Los Patrones

Los *patrones* son las condiciones que indican sobre qué líneas se debe ejecutar una cierta acción.

Los patrones de Awki son:

- `BEGIN` - Es un patrón especial que sirve para indicar una acción que se ejecutará antes de comenzar a leer líneas de la entrada. Las variables NR y NF valen 0 y los campos quedan sin asignar.

- `END` - Es un patrón especial que sirve para indicar una acción que se ejecutará luego de terminar el procesamiento de todas las líneas de la entrada. Las variables automáticas quedan con los valores que resultaron del procesamiento de la última línea.
- *expresión* - Se considera verdadera si la expresión evalúa a verdadero; esto es: un número distinto de 0 o una cadena no vacía.

La representación en Haskell de los patrones es como sigue:

```
data Patron
    = BEGIN
    | END
    | Pat Expr
```

Las acciones

Las acciones se escriben en un lenguaje de scripting con instrucciones de control al estilo de C:

Las instrucciones simples son:

- *expresión* - Esta instrucción tiene como efecto la evaluación de la expresión. Tiene sentido cuando la expresión provoca efectos laterales (asignación, incremento, decremento)
- `print e1, e2, ..., en` - Esta instrucción envía a la salida los valores de las expresiones `ei` separados por un tabulador y termina con un fin de línea .

Las instrucciones de control:

- `exit` - Tiene como efecto que Awki deje de ejecutar la acción corriente y termine de procesar la entrada ignorando las líneas que restan. Sin embargo, si hubiera una acción END, esta será ejecutada.
- Instrucciones de control a la C, con su semántica habitual: `if`, `while`, `do`, `for` y secuencia `{... ; ...}` (por más detalles consultar cualquier Manual de Awk)

Representación en Haskell:

```
data Statement
    = Empty
    | Simple Expr
    | Print [Expr]
```

```
  | Exit
  | Sequence [Statement]
  | If Expr Statement Statement
  | For Expr Expr Expr Statement
  | While Expr Statement
  | DoWhile Statement Expr
```

Programa Awki

Un programa Awki esta formado por una secuencia de reglas `(patrón, acción)`. Su representación en Haskell resulta:

```
newtype AwkiProg = AwkiProg [(Patron,Statement)]
```

Se pide

Se deberá escribir un módulo Haskell que

1. importe el módulo `AwkiSA`, suministrado por el equipo docente, que contiene la definición de la sintaxis abstracta de Awki explicada en la sección anterior.
2. implemente una función:

   ```
   runAwki :: AwkiProg -> String -> String
   ```

   que realiza la ejecución del programa Awki sobre el *string* de entrada y produce el *string* de salida.

Importante:

- El módulo a implementar debe llamarse `RunAwki` y debe exportar la función solicitada `runAwki`.

  ```
  module RunAwki(runAwki) where
  import AwkiSA
  ...
  ```

- El archivo `AwkiSA` no puede ser modificado.
- Pueden escribirse otros modulos auxiliares que serán entregados junto con el módulo pedido.

- El archivo `AwkiSA` no debe ser entregado. La tarea se compilará con una versión nuestra del módulo coincidente con la original distribuida con esta letra.
- Las tareas serán compiladas y ejecutadas con el compilador *ghc* de la versión *2014.2.0.0* de la *Haskell Platform* en ambiente *Linux*.

En la semana del 11/05 será enregado un parser de Awki que permitirá ejecutar programas escritos en su sintaxis concreta.

Referencias

- *AWK*. Entrada en Wikipedia. http://es.wikipedia.org/wiki/AWK.
- *The AWK Programming Language*. Aho, Alfred V. and Kernighan, Brian W. and Weinberger, Peter J. Addison-Wesley Longman Publishing Co., Inc. 1988.
- *The GNU Awk User's Guide*. http://www.gnu.org/software/gawk/manual/html_node/

Entrega

La tarea se puede entregar desde el 28/05 al 01/06. La entrega finaliza el 01/06 a las 13 horas. Se debe entregar un archivo `TareaFP.tar.gz` que contenga todos los módulos implementados por el estudiante.