Laboratorio Programación Funcional

INCO

mayo 2013

Contents

---

1 Introducción

En esta tarea se debe implementar un editor de textos orientado a líneas.

El editor `edi` es una versión simplificada de el editor *ed* que se distribuye con todos los sistemas *Unix*. Incluímos un vínculo a la página de manual completa del editor *ed* que servirá como referencia para la implementación de *edi*.

---

2 El editor `edi`

edi es un editor de texto orientado a líneas. Es usado para crear, desplegar y modificar archivos de texto. *edi* es una versión reducida del editor de textos *ed* que se distribuye corrientemente con los sistemas Unix. (ver página de manual =ed(1)=)

Se invoca con un argumento de archivo. Al inicio, una copia del archivo se carga en el buffer del editor. Los cambios se realizan sobre esta copia y no

directamente en el propio archivo. En el momento de salir, cualquier cambio que no haya sido salvado (con `w`) se perderá.

La edición se realiza en dos modos distintos: *comando* y *entrada*. Cuando se incicia, *edi* está en modo commando. En este modo, *edi* lee comandos desde la entrada estándar y los ejecuta. Un comando típico puede ser:

```
10,20m30
```

que mueve el rango de líneas de la 10 a la 20 a la dirección siguiente a la línea 30 (ver comando `m`).

Cuando comandos tales como: `a` (append), `i` (insert) o `c` (change) son dados, el *edi* pasa a modo entrada. Este es el método primario para agregar texto al archivo. En este modo, ningún comando está disponible; en su lugar la entrada estándar es copiada directamente sobre el buffer de edición. La entrada está formada por líneas que están finalizadas como es usual por el carácter =". `El modo /entrada/ finaliza cuando se ingresa una línea que contiene como único carácter un punto (.=)`.

Todos los comandos *edi* operan sobre líneas completas o rangos de líneas; por ejemplo el comando `d` borra líneas; el comando `m` mueve líneas, etc.

2.1 Estructura de los comandos

En general, un comando consiste de cero o más direcciones de línea, seguidos por un carácter (que identifica el comando) y posibles argumentos adicionales. Esto es, los comandos tienen la siguiente estructura:

```
[direccion [,direccion]]comando[parametros]
```

Las direcciones indican el rango de líneas que será afectado por el comando. Si se especifican menos direcciones que las requeridas, se toman ciertas direcciones por omisión que dependen de cada comando.

2.2 Comparación de `edi` con `ed`

El editor `edi` es una restricción del `ed`. La página de manual del editor `ed` servirá de referencia para la implementación del `edi`

Enumeramos a continuación las funciones que el `edi` debe proveer. La explicación completa de las mismas está en la página de manual mencionada:

Direcciones *edi* maneja todos los tipos de direcciones que maneja *ed* excepto las de las formas: `/re/` y `?re?`. Ver más abajo la gramática de las direcciones aceptadas.

Expresiones Regulares *edi* no soporta expresiones regulares. Por lo tanto tampoco soporta ninguno de los comandos relacionados con expresiones regulares.

Argumentos *edi* siempre se invoca desde la línea de comandos con un único argumento que es el nombre del archivo a ser procesado. Cualquier otra forma de invocación es rechazada con un mensaje de error.

Comandos *edi* dispone de los siguientes comandos del *ed*:

- a, c, d, i, j, m, n, p, q, Q, t, x, y, =

La descripción de todos los comandos se puede obtener en la página de manual de ed

El comando w sólo se admite en su variante sin argumentos.

El comando q sólo es válido cuando no hay cambios sin salvar en el buffer; en otro caso es ignorado. El comando Q produce la salida incondicional.

Las direcciones por omisión (default) que asume *edi* son las mismas que tiene *ed*.

2.3 Gramática de las direcciones

La siguiente gramática representa de manera precisa las direcciones aceptadas por *edi* (no necesariamente coincide con las de *ed*)

```
direc     =   base
          |   direc op numero

base      =   numero
          |   $
          |   .
          |   'letra
          |   op numero

op        =  + | -
```

Observaciones:

- `numero` es un entero sin signo.

- `letra` es una letra minúscula. Las direcciones de la forma ='x= hacen referencia a una línea que fue *marcada* con el comando k.
- Las direcciones de la forma `op numero` son llamadas *relativas* y son equivalentes a `. op numero`
- Los operadores no se pueden aplicar sucesivas veces a un número como en *ed*. Por ejemplo, no pueden aparecer direcciones de la forma $-n$

---

3 Guía para el diseño

Se debe implementar un módulo principal cuyo nombre debe ser `Main`

```
module Main(main) where
```

Dentro de este módulo la función principal que implementa el editor se debe llamar `main`

```
main :: IO ()
```

A continuación se dan algunos criterios para diseñar la tarea. Nada de lo que se presenta aquí tiene carácter obligatorio. El estudiante puede sentirse libre de modificar, eliminar o agregar lo que estime conveniente.

La estructura general de la función `main` puede ser como la siguiente:

```
main = do editor  <-  iniciarEdi
          editor' <- procesarComandos editor
          terminarEdi editor'

procesarComandos editor =
       do comando <- leerComando
          if esValidoComando comando
          then do editor' <- ejecutarComando comando editor
                  procesarComandos editor'
          else do procesarError
                  procesarComandos editor
```

`iniciarEdi` se encarga de cargar el estado inicial del editor. Se deberá definir un tipo (algebraico) que represente el estado del editor (buffer, linea corriente, etc). Para obtener el nombre de archivo pasado como argumento

en la línea de comandos debe usarse la función `getArgs` de la biblioteca `System`

`leerComando` es una función que realiza el reconocimiento de un comando y lo retorna en una estructura adecuada. Se deberá definir un tipo `Comando` que podría ser algo así:

```
data Comando = Comando [Direc] Accion [Arg]
```

A su vez se deberá dar definiciones adecuadas para los tipos:

`Direc` Las direcciones que deberán definirse de acuerdo con la gramática de más arriba.

`Accion` Representa el comando propiamente dicho. Usar un tipo enumerado para representar las acciones (no utilizar un `Char`)

`Arg` El tipo de los argumentos. Alcanza con utilizar un sinónimo de `String`.

Para implementar el *parser* se deben utilizar los combinadores de *parsing* presentados en el capítulo 17 del libro. En el foro de la asignatura se presenta una guía para construir el parser de las direcciónes:

- https://eva.fing.edu.uy/mod/forum/discuss.php?d=29451

`ejecutarComando` es la función que interpreta el comando y realiza la acción correspondiente.

Podría ser conveniente realizar una estructura de módulos para descomponer adecuadamente el problema. Por ejemplo, podría haber un módulo para las direcciones, otro para los comandos, etc. Se recomienda leer el capítulo 15 del libro para entender mejor el mecanismo de módulos de haskell.