Paradigmas de la Programación
Examen Final

25 de febrero de 2009

Importante para libres: Deberán aprobar la parte de regulares previo a la correción de los ejercicios de libres. El puntaje del examen (que no corresponde al del acta) será el mínimo de la media de los ejercicios sin tener en cuenta los de libre y la media teniendo en cuenta éstos. De esta manera, el resultado de los ejercicios de libres sólo restan puntaje. Los ejercicios de libres son los que estan marcados con (*).

Primera Parte

1. La siguiente función `{Elegir Xs}` toma una lista de listas `Xs` como argumento:

```
fun {Elegir Xs}
   case Xs
   of nil then nil
   [] X|Xr then
      Y={Elegir Xr}
   in
      if {Length X}>{Length Y} then X else Y end
   end
end
```

   a) ¿Cuál es el resultado de `{Elegir [[a]]}`?

   b) ¿Cuál es el resultado de `{Elegir [[b a] [a] [b c]]}`?

   c) Dé una función recursiva a la cola `{ElegirConAcc Xs Y}` equivalente a `{Elegir Xs}` que utilice a `Y` como acumulador.

2. La función `string2int` recibe una lista de carateres donde cada caracter es un número, la función devuelve el número representado por la lista. Por ejemplo, para la entrada `['2' '3' '4']`, `{ string2int ['2' '3' '4'] } = 234`. Dé dos implementaciónes de la función `string2int`, una utilizando un acumulador y otra utilizando alguna de las funciones fold. Suponga que existe una función `digitToInt d` que transforma un caracter `d` en el número correspondiente.

3. Observe el siguiente código.

```
declare
proc {DifDiv X Y ?Z}
   {MidDiv X Y Z}
end

proc {MidDiv X Y ?Z}
   {Div X Y Z}
end

proc {Div X Y Z}
   Z = X / Y
end

{Browse {DifDiv 3 0}}
```

El procedimiento `Div` contiene la operación de kernel `/` que produce una excepción cuando el segundo argumento es cero. Así como está el código, su ejecución produce un error de excepción no capturada. Para este ejercicio deberá implementar las siguientes variaciones del código anterior, **sin utilizar cláusulas `if` ni `case`**:

a) `DifDiv` devuelve `error` si el segundo valor es cero. `MidDiv` levanta excepción si el segundo argumento es cero.

b) `DifDiv` devuelve `error` si el segundo valor es cero. `MidDiv` devuelve `error` si el segundo argumento es cero.

c) `DifDiv` devuelve `error` si el segundo valor es cero. `Div` devuelve `error` si el segundo argumento es cero.

Recuerde: No debe utilizar cláusulas `if` ni `case` para estas variaciones.

4. (*) Escriba una función `{SelectMap Xs F G H}` que tome una lista `Xs` y tres funciones unarias `F`, `G` y `H` como parámetros. La función devuelve una lista cuyos elementos son obtenidos aplicando `G` ó `H` a los elementos de `Xs` dependiendo de si el resultado de aplicar `F` al elemento da `true` (en cuyo caso aplica `G`) o `false` (en este caso aplica `H`).

   Por ejemplo, con las definiciones

   ```
   fun {IsPos N} N>=0 end
   fun {Inc N} N+1 end
   fun {Dec N} N-1 end
   ```

   la llamada `{SelectMap [~1 1 2 2] IsPos Inc Dec}` devuelve `[~2 2 ~3 3]`.

   a) Escriba la función de manera que sea recursiva a la cola.

   b) Escriba la función en términos de `Map` de la manera más compacta posible.

Segunda Parte

1. Defina un tipo conjunto que tenga las siguientes operaciones:

- `Crear`: Crea un conjunto vacio.
- `EsVacio`: Devuelve true o false dependiendo de si el conjunto es vacio o no.
- `Agregar`: toma un entero y lo agrega al conjunto.
- `Eliminar`: toma un entero y lo elimina del conjunto.
- `Pertenece`: toma un entero y devuelve true o false dependiendo si el entero esta en el conjunto o no.

La implementación tiene que ser abierta, con estado y empaquetada. Tiene además que ser implementada con una lista ordenada de pares. El primer elemento del par es el número en si, mientras que el segundo es un booleano que indica si el número está en el conjunto o si fue borrado. Tenga cuidado con el caso en que se agrega un número que fue borrado.

Muestre un ejemplo de uso de cada una de las operaciones.

2. Implemente un TAD diccionario de una manera segura y con estado usando `Wrap` y `Unwrap`. Suponga que la operación `NewWrapper` está dada. El TAD tiene las siguientes funciones:

   - `D = {NewDictionary V}`: Devuelve un diccionario vacío. `V` es el valor por omisión a usar en `GetCond` (ver más abajo).
   - `{Put D L V}`: Agrega un elemento al diccionario con clave `L` (que es un literal) y valor `V`. Si ya había un elemento con esa clave, lo sobreescribe.
   - `{Delete D L}`: Borra el elemento con clave `L` del diccionario.
   - `V = {GetCond D L}`: Devuelve el elemento con clave `L` del diccionario. Si no existe devuelve el valor por defecto indicado en `NewDictionary`.

   Las claves del diccionario deberán ser literales. Las funciones deberán levantar excepciones cuando esto no sea respetado. Los valores podrán ser de cualquier tipo.

3. Observe el siguiente código donde se implementa un esquema básico de productor consumidor. El código tiene tres partes claramente distinguibles: 1) La definición del productor, 2) la definición del consumidor y finalmente 3) el código que hace que el productor y el consumidor interactúen.

```
declare
proc {DGenerate N Xs}
   X|Xr = Xs
in
   X = N
   {DGenerate N+1 Xr}
end

fun {DSum ?Xs A Limit}
   if Limit > 0 then
      X|Xr = Xs
```

```
   in
      {DSum Xr A+X Limit-1}
   else A end
end

local Xs S in
   thread {DGenerate 0 Xs} end % Producer thread
   thread S={DSum Xs 0 150000} end % Consumer thread
   {Browse S}
end
```

a) Reimplemente este código de manera que el productor produzca solamente a medida que el consumidor necesite datos. Utilice el procedimiento `ByNeed` y **no** la instrucción `lazy`.

b) La *lazyness* puede ser implementada de la siguiente manera utilizando alto orden. El productor pasa al consumidor una función de cero argumentos al consumidor. Cuando el consumidor necesita un elemento, llama a la función. La función devuelve un par `X#F2` donde `X` es el próximo elemento a consumir y `F2` es una nueva función que tiene el mismo comportamiento que `F`. Modifique el código anterior para usar esta técnica. Es importante que las tres componentes sean de nuevo claramente indentificables en el código que proponga.

4. (*) Semántica de Disparadores. ¿Qué pasa si para disparar un *trigger* protegiendo una variable X obviamos la condición de que se intente bindear a X? Argumente y muestre ejemplos que respalden sus argumentos.