Paradigmas de Programación

Práctica 12

Ejercicios:

1. Ejercicio opcional. Descargue y descomprima el fichero `simplec.tar.gz` proporcionado. Contiene la implementación de un intérprete para un sencillo lenguaje de programación al que llamaremos Simple C.

   Eche un vistazo a los ficheros con nombre `p*.c` para familiarizarse con la sintaxis de este lenguaje. No le resultará complicado. Efectivamente, se parece al lenguaje C, pero es mucho más sencillo.

   Eche un vistazo también al fichero `simpleC.ml`. Este es el fichero que tendrá que modificar para resolver los apartados de este ejercicio, y es, por tanto, el único que tendrá que entregar después. Contiene la implementación de un tipo de dato `context` para gestionar las variables que van creando en memoria nuestros programas escritos en Simple C. Contiene también los tipos `arith` y `boolean`, para representar expresiones aritméticas y booleanas, respectivamente. Todos estos tipos sirven de apoyo para la definición del tipo `statement`, como soporte de las instrucciones del lenguaje Simple C. Un programa en Simple C se podrá entonces representar mediante un valor de tipo `statement list`. Por ejemplo, el programa

   ```
   n = 5; if (n % 2 == 0) print 0 else print 1;
   ```

   se corresponde con el valor

   ```
   [Bind ("n", C 5.);
    If (Comp (Equal, Arith_bi_op (Mod, Var "n", C 2.), C 0.),
        [Print (C 0.)], [Print (C 1.)])]
   ```

   Puede echar también un vistazo a los ficheros `myLexer.mll` y `myParser.mly`, aunque no es estrictamente necesario. Sólo debe saber que contienen la implementación de las funciones de análisis léxico y sintático capaces de transformar la sintaxis de Simple C en la sintaxis abstracta que maneja internamente el intérprete. Es decir, transforman un programa en un valor de tipo `statement list`.

   Estas funciones se llaman desde el fichero `main.ml`, al que también puede echar un vistazo, aunque, una vez más, tampoco es estrictamente necesario. Sólo tiene que saber que contiene la implementación de la función que abre un fichero de texto, transforma su contenido en una `statement list`, y llama a la función `exec` del módulo `SimpleC` para ejecutar dicha lista de instrucciones, al tiempo que intercepta todas las excepciones asociadas a los posibles errores que pueden ocurrir, e informa al usuario mediante un mensaje si alguna de esas circunstancias se produce.

   Compile el intérprete con la orden `make all` y obtendrá el ejecutable `run`. Pruebe ahora a ejecutar el primer programa. Debería obtener lo siguiente:

   ```
   $ ./run p1.c
   1.
   ```

   Volvamos al fichero `simpleC.ml`. Localice y observe la función `exec`. Esta función recibe un contexto (una lista de variables en memoria, con sus respectivos valores) y una lista de instrucciones, que va ejecutando una a una hasta llegar a la lista vacía. La ejecución

de cada instrucción devuelve un nuevo contexto actualizado (en algunos casos, podría ser el mismo), que es el que debe considerarse para la ejecución de la siguiente instrucción, y así sucesivamente hasta ejecutarlas todas. Cuando la lista se hace vacía, se devuelve el último contexto obtenido.

En esta función, el caso de la instrucción `While` está mal programado. Corríjalo. Pista: para ejecutar un `While`, primero se testea la condición; si ésta no se cumple, se devuelve el mismo contexto recibido y se continúa ejecutando el resto de instrucciones; pero si la condición se cumple, se ejecuta una vez el cuerpo del `While`, y seguidamente, en el nuevo contexto obtenido, se vuelve a ejecutar el mismo `While`. Recompile el intérprete y pruebe a ejecutar el segundo programa. Debería obtener lo siguiente:

```
$ ./run p2.c
120.
```

El caso de la instrucción `For` también está mal programado. Corríjalo. Sugerencia: evite ejecutar los lazos `For` en base a instancias más sencillas de otros lazos `For`; en lugar de eso, "traduzca" el lazo `For` a un lazo `While` equivalente y ejecute este último lazo. Recompile el intérprete y pruebe a ejecutar el tercer programa. Debería obtener lo siguiente:

```
$ ./run p3.c
120.
```

Ahora vamos a ponernos un poco serios (tampoco mucho) y no permitiremos ciertas "artimañas" que el lenguaje C sí permite. Por ejemplo, escribir en la parte `init` de un `For` cualquier instrucción que no sea una asignación. Así pues, reescriba este caso para que sólo se acepten aquellos lazos `For` que contengan una instrucción `Bind` en la parte `init`. Si no es el caso, debe activarse la excepción `Runtime_error` acompañada del string `"illegal statement in init section of for loop"`. Recompile el intérprete y pruebe a ejecutar el cuarto programa. Debería obtener lo siguiente:

```
$ ./run p4.c
illegal statement in init section of for loop
```

Por último, tampoco vamos a permitir que se manipule el contador de un `For` dentro su cuerpo. Para ello, escriba una función

```
legal_body : string -> statement list -> bool
```

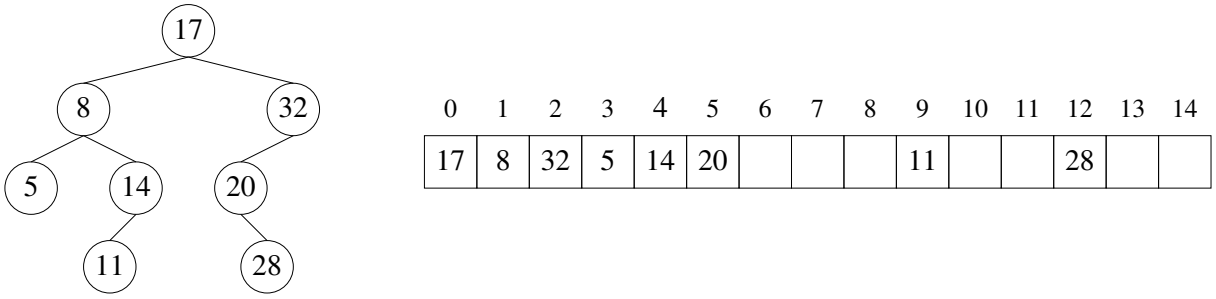
Al resolver el caso de la instrucción `For`, llame a esta función con el nombre del contador (es decir, con el string que aparece en la instrucción `Bind` de la parte `init`) y con el `body` del propio `For`. Esta función debe comprobar que en dicho `body` no existe ninguna otra instrucción `Bind` que redefina el contador. Si la función devuelve `true`, el lazo `For` puede ejecutarse. Si no es el caso, debe activarse la excepción `Runtime_error` acompañada del string `"illegal manipulation of counter in for loop"`. Recompile el intérprete y pruebe a ejecutar el quinto programa. Debería obtener lo siguiente:

```
$ ./run p5.c
illegal manipulation of counter in for loop
```

Es altamente recomendable que escriba otros programas más complejos que permitan probar a fondo el intérprete. En particular, escriba programas con lazos dentro de lazos para verificar el correcto funcionamiento de los casos `While` y `For` de su función `exec`.

Y, en el caso de la función `legal_body`, tenga en cuenta que un `Bind` ilegal no tendría porqué aparecer a primer nivel en la lista de instrucciones del `body` a estudiar, sino que podría aparecer dentro de un `If`, dentro de un `While` o incluso dentro de otro `For`. Aunque puede implementarse mediante una única función, en cierto modo, se trata de una función doblemente recursiva, en el sentido de que recorre recursivamente una lista de instrucciones, y estudia también de manera recursiva la estructura interna de cada una de esas instrucciones para comprobar su validez.

2. En programación imperativa, un árbol binario se puede representar en un array colocando el nodo raíz en el índice 0 y, a partir de ahí, el hijo izquierdo de un nodo en el índice i se encontrará en el índice $2i+1$ y su hijo derecho se encontrará en el índice $2i+2$. Tal y como se muestra en la figura siguiente, esta representación utiliza las posiciones matemáticas para vincular a los nodos sin necesidad de punteros. Por supuesto, la estructura es más eficiente cuanto más lleno esté el árbol, ya que, si el árbol no está demasiado lleno, el array contendrá muchas posiciones vacías. No obstante, aunque no complicaremos tanto este ejercicio, siempre podrían aplicarse técnicas de compactación de tablas para no desperdiciar tanto espacio.



Para representar en un array árboles binarios de cualquier tipo utilizaremos un `'a option array`. De esta forma, las posiciones del array contendrán o bien el valor `Some x` para cada nodo `x` del árbol, o bien el valor `None` cuando sean vacías.

Utilizando este tipo y también el módulo `BinTree` de la práctica anterior, se pide implementar las siguientes funciones:

```
from_bin : 'a BinTree.t -> 'a option array
breadth : 'a option array -> 'a list
mem : ('a -> 'a -> bool) -> 'a -> 'a option array -> bool
```

La función `from_bin` permite construir árboles binarios sobre arrays a partir de los árboles del módulo `BinTree`.

La función `breadth` permite obtener el recorrido en anchura de un árbol binario almacenado en un array. Sugerencia: dada la disposición de los nodos en este tipo de estructura, un simple lazo `for` que recorra el array ignorando las posiciones vacías debería ser suficiente.

Si el árbol es un árbol binario de búsqueda como los descritos en el ejercicio 4 de la práctica anterior (el del ejemplo anterior lo es), tiene sentido implementar una función de búsqueda que utilice la misma estrategia que la función `mem` de ese mismo ejercicio. Así pues, debe implementar una nueva función `mem` para valores de tipo `'a option array` equivalente a ésa. Y, atención, se pide explícitamente que en este caso no utilice recursividad, sino lazos `while` y referencias.

Realice todas estas implementaciones en un fichero `aTree.ml` y compílelo bajo la forma de un módulo con la interfaz proporcionada.

Ejemplo de ejecución:

```
$ ocaml
OCaml version ...

# #load "bintree.cmo";;
# #load "atree.cmo";;
# open BinTree;;
# open ATree;;

# let bt =
    comb 17 (comb 8 (leaf_tree 5) (comb 14 (leaf_tree 11) empty))
            (comb 32 (comb 20 empty (leaf_tree 28)) empty);;
val bt : int BinTree.t = <abstr>

# let at = from_bin bt;;
val at : int ATree.t =
  [|Some 17; Some 8; Some 32; Some 5; Some 14; Some 20; None; None; None;
    Some 11; None; None; Some 28; None; None|]

# breadth at;;
- : int list = [17; 8; 32; 5; 14; 20; 11; 28]

# mem (<) 14 at;;
- : bool = true

# mem (<) 29 at;;
- : bool = false
```

3. Ejercicio opcional. Estudie con detalle el contenido del fichero `queue.ml` proporcionado. Contiene una clase para definir objetos capaces de gestionar colas de elementos de cualquier tipo. Se trata de una implementación muy eficiente basada en dos listas.

   En un fichero `bq.ml`, escriba una nueva versión de la función `breadth` del módulo `BinTree`, que utilice explícitamente un objeto de tipo `'a BinTree.t queue` para ir recorriendo en anchura los niveles de un árbol binario. Si realiza una buena definición, esta nueva versión de `breadth` debería recorrer sin problema árboles muy grandes, incluso de varios millones de elementos.

   Existe una estructura de datos denominada bicola o cola doblemente terminada. Se trata de una cola donde los elementos se pueden añadir y quitar por ambos extremos. En el propio fichero `queue.ml`, defina `queue'` como una subclase de `queue` que añada los siguientes métodos: `push'`, para encolar un elemento por el principio de la cola; `peek'`, para consultar cuál es el último elemento de la cola sin desencolarlo; y `pop'`, para recuperar y desencolar el último elemento de la cola.