1. Pistas y consejos

En este documento os sugerimos 9 funciones y 2 terminales que pueden ser usados para el conjunto de funciones y terminales de vuestra práctica final. Por supuesto, no es necesario que se usen todos: aquí tratamos de proporcionar todos aquellos elementos que hemos considerado que podrían ser útiles, pero puede (de hecho es bastante probable) que algunos no tengan sentido en el diseño que cada alumno plantee para su aplicación. También es posible (y deseable) que alguien utilice terminales o funciones que no aparezcan aquí. Lo único que debéis recordar es que una de las reglas básicas de la PG es que el conjunto de funciones y terminales debe de ser tan sencillo como sea posible sin dejar de ser lo suficientemente amplio para producir soluciones generales.

Terminales:

- index: Es una variable de iteración que toma secuencialmente varios valores de índice cuando está incluido en el interior del work de una instrucción dobl o el valor 0 en todos los demás casos.

- *len*: Es una variable que contiene la longitud del vector que se está tratando de ordenar.

Funciones:

- dobl: (dobl start end work). dobl es un operador iterativo que toma un índice de partida start y un índice de llegada end. Va incrementando la variable index en uno desde start hasta end-1 o *len*-1 (lo que sea más pequeño), ejecutando en cada iteración el trabajo contenido en work. Este trabajo, como ya sabemos, tendrá forma de árbol que deberá ser evaluado tantas veces como indiquen los parámetros. Devuelve minimo(end, *len*), es decir, el más pequeño entre end y *len*, o 0 si se produjese algún error.

- swap: (swap idx1 idx2). Toma dos argumentos de tipo entero que llamaremos, por ejemplo, idx1 e idx2. Intercambia los elementos del vector a ordenar situados en las posiciones idx1 e idx2

Intercambiar vector[idx1] por vector[idx2]

Devuelve 0 si los índices que recibe no son correctos o si sucede cualquier otro problema; en caso contrario devuelve idx1.

- wismaller: (wismaller idx1 idx2). Compara los valores en las posiciones idx1 e idx2 y devuelve el índice del menor de ellos. Si ambos son iguales, devuelve idx2. Si hay algún error, devuelve 0.

- wibigger: (wibigger idx1 idx2). Compara los valores en las posiciones idx1 e idx2 y devuelve el índice del mayor de ellos. Si ambos son iguales, devuelve idx2. Si hay algún error, devuelve 0.

- ++: (++ idx). Aumenta el valor de idx en una unidad de forma protegida, evitando que el valor de idx ascienda por encima de *len*-1. Devuelve el nuevo valor de idx.

- --: (-- idx).). Decrementa el valor de idx en una unidad de forma protegida, evitando que el valor de idx descienda por debajo de 0. Devuelve el nuevo valor de idx.

- -: (- idx1 idx2). Devuelve idx1 idx2 de forma protegida, evitando que el valor devuelto sea menor que 0. Si idx2 fuera mayor que idx1, se devuelve 0.

- ifele: (ifele idx1 idx2 work1 work2). Si el valor del elemento en la posición idx1 es menor o igual que el de la posición idx2, se ejecuta work1. De lo contrario se ejecuta work2. Devuelve el valor de retorno de work1 o work2, el que haya sido ejecutado. A menos que se produzca un error, en cuyo caso se devuelve 0.

- ifege: (ifege idx1 idx2 work1 work2). Si el valor del elemento en la posición idx1 es mayor o igual que el de la posición idx 2, se ejecuta work1. De lo contrario se ejecuta work2. Devuelve el valor de retorno de work1 o work2, el que haya sido ejecutado. A menos que se produzca un error, en cuyo caso se devuelve 0.

NOTAS:

Todos los nodos tienen que devolver un valor de tipo compatible con los demás nodos. Lo más lógico es devolver siempre índices (int), pero se valora la originalidad, así que una solución de PG tipada en la que se combinen operadores y terminales con distintos tipos de retorno obtendría mejor nota. Si devolviendo siempre índices se produjera algún error, o si los parámetros recibidos no fueran correctos (estamos tratando con índices, es fácil que se produzcan errores en forma de índices negativos, mayores que el tamaño del vector, etc.) se debe devolver simplemente 0.

Las iteraciones en programación genética introducen la posibilidad de que las evaluaciones de algunos árboles caigan en bucles infinitos. Para evitarlo, se suele restringir el número de iteraciones de cada bucle así como el número de iteraciones totales de todos los bucles del individuo. Sugerimos que en este ejercicio no se permita que cada bucle realice más de 200 iteraciones y que entre todos los bucles de ceda individuo no se lleven a cabo más de 2000 iteraciones en total. Estos límites deberían ser más que suficientes y, de nuevo, queda a discreción del alumno el modificarlos para generar mejores individuos, justificándolo siempre en la memoria del ejercicio.