Ejercicio 1

```
int main(int argc, char** argv) {
    Foo* foo = new Foo(1);         // t = 1
    foo->sil();                    // t = 2
    foo->sil();                    // t = 3
    foo->output();                 // 3

    Bar* bar = new Bar(2);         // t = 4
    bar->sil();                    // t = 8
    bar->sil();                    // t = 16
    bar->output();                 // | 16 |

    Foo* foobar = new Bar(3);      // t = 6
    foobar->sil();                 // t = 12
    foobar->sil();                 // t = 24
    foobar->output();              // 24
}

template <typename T>
void transformprint(const T& t, const Foo<T>& foo) {
    std::cout<<foo.transform(t) <<std::endl;
}

int main(int argc, char** argv) {
    Bar bar(3);
    transformprint<int>(3,bar);               // 9
    transformprint<std::string>(std::string("la"),bar);  // lalala
}
```

Ejercicio 2

Apartado a)
Un método genérico es aquel que utiliza tipos de datos (parámetros) desconocidos a priori, de manera que podrá utilizar cualquier tipo de dato compatible con las operaciones realizadas en dicho método.
Se utiliza especificando valores concretos para elementos parámetro.
Sí se puede, se deducen automáticamente.

Apartado b)
En este caso el único fragmento de código que compila es el escrito en el lenguaje C++

Apartado c)
En el caso de Java y Haskell es necesario cuando se está utilizando el polimorfismo paramétrico. Acotar el tipo de dato, para saber el conjunto de operaciones que podemos realizar con él. Mientras que en C++ como los errores se detectan al compilar el código que instancia el código genérico al usar la clase 7 método genérico para un tipo concreto, por lo tanto no es necesario imponer ninguna restricción.

Apartado d)
En el caso de Haskell valdría con sustituir la primera línea de código por: `min :: Ord t => t -> t -> t`

El cambio realizado impone que el parámetro t se puede ordenar de modo que la operación "<" está disponible para el dato t

C++
```
template<typename T>
T min(T t1, T t2) {
    if (t1<t2) return t1;
    else return t2;
}
```

Java
```
class Min {
 public static <T>
     T min(T t1, T t2) {
         if (t1<t2) return t1;
         else return t2;
     }
}
```

Haskell
```
min :: t -> t -> t
min a b
 | a < b = a
 | otherwise = b
```

Ejercicio 3

```
esPrimo :: Int -> Bool
esPrimo n = [1,n] == [x | x <- [1..n], mod n x == 0]

divPrimos :: Int -> [Int]
divPrimos n = [x | x <- [2..n], mod n x == 0, esPrimo x]

sonDivisores :: Int -> [Int] -> Bool
sonDivisores x [] = True
sonDivisores x [y] = mod x y == 0
sonDivisores a (x:xs)
    | mod a x == 0 = sonDivisores a xs
    | otherwise = False

poderosos :: [Int]
poderosos = [x | x <- [1..], all (\y -> mod x (y^2) == 0) (divPrimos
x)]

poderosos' :: [Int]
poderosos' = [x | x <- [1..], sonDivisores x (map (^2) (divPrimos x))]

npoderosos :: Int -> [Int]
npoderosos n = take n poderosos

poderososHasta :: Int -> [Int]
poderososHasta mx = takeWhile (<=mx) poderosos
```

Ejercicio 4

```
#include <list>
#include <iostream>
using namespace std;

class Empleado {
protected:
    string nombre, dni;
    int dinero;
public:
    Empleado(const string& nom, const string& dni) : nombre(nom),
dni(dni), dinero(0) {}
    virtual ~Empleado() {}

    string suNombre() const {
        return nombre;
    }

    int suDinero() const {
        return dinero;
    }

    void hola() {}

    void pagar(int cantidad) {
        dinero += cantidad;
    }
};

class Director : public Empleado {
public:
    Director(const string& nom, const string& dni) : Empleado(nom, dni)
{}
};

class Fijo : public Empleado {
    int salario;
public:
    Fijo(const string& nom, const string& dni, int sal) : Empleado(nom,
dni), salario(sal) {}

    int sueldo() const {
        return salario;
    }
};
```

```
class PorHoras : public Empleado {
    int horas, salarioPorHora;
public:
    PorHoras(const string& nom, const string& dni, int h, int sal) :
Empleado(nom, dni), horas(h), salarioPorHora(sal) {}

    int horasTrabajadas() const {
        return horas;
    }

    int precioHora() const {
        return salarioPorHora;
    }
};

class Becario : public Empleado {
public:
Becario(const string& nom, const string& dni) : Empleado(nom, dni) {}

    int sueldo() const {
        return 0;
    }
};

class Encargado : public Fijo {
public:
    Encargado(const string& nom, const string& dni, int sal) :
Fijo(nom, dni, sal) {}
};

class Empresa {
    list<Empleado*> empresa;
    int beneficios;
public:
Empresa(list<Empleado*> emp, int ben): empresa(emp), beneficios(ben) {}
```

```
    bool contratar(Empleado& nuevoEmpleado) {
        if (dynamic_cast<Director*>(&nuevoEmpleado) != nullptr) {
            for (auto i : empresa) {
                if (dynamic_cast<Director*>(i) != nullptr) {
                    return false;
                }
            }
            empresa.push_back(&nuevoEmpleado);
            return true;
        }
        else {
            empresa.push_back(&nuevoEmpleado);
            return true;
        }
    }
    void pagar() {
        for (auto i : empresa) {
            if (dynamic_cast<Director*>(i) != nullptr) {
                Director* d = dynamic_cast<Director*>(i);
                d->pagar(beneficios);
            }
            else if (dynamic_cast<Fijo*>(i) != nullptr) {
                Fijo* f = dynamic_cast<Fijo*>(i);
                f->pagar(f->sueldo());
            }
            else if (dynamic_cast<PorHoras*>(i) != nullptr) {
                PorHoras* ph = dynamic_cast<PorHoras*>(i);
                ph->pagar(ph->horasTrabajadas()*ph->precioHora());
            }
        }
    }

    void saludar() const {
        for (auto i : empresa) {
            i->hola();
        }
    }

    void dineroEmpleados() const {
        for (auto i : empresa) {
            cout << i->suNombre() << ": " << i->suDinero() << endl;
        }
    }
};
```