Geometric

WALL·E

Tercer Proyecto de Programación

Carrera de Ciencia de la Computación
Universidad de La Habana
2023

INTRODUCCIÓN

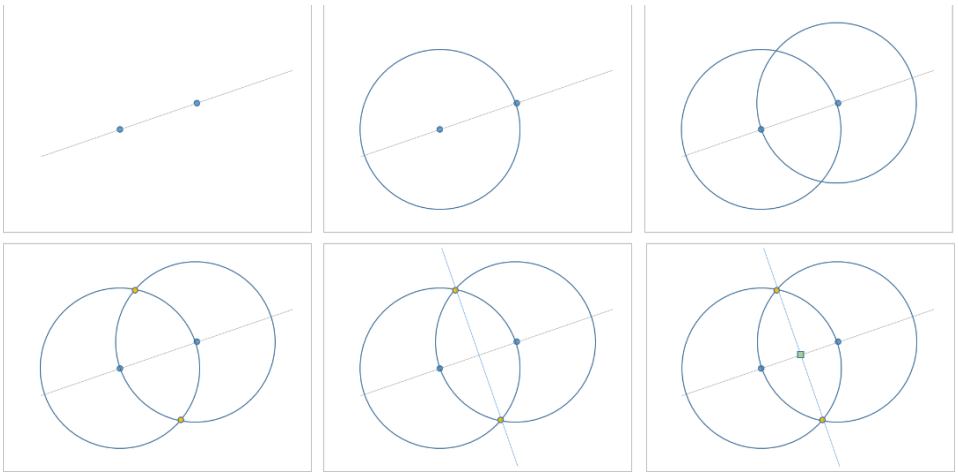
A un año de la programación de WallE se ha decidido incorporarle una componente para el análisis geométrico.

Mediante un programa, WallE será capaz de representar conceptos geométricos (como puntos, líneas o circunferencias), graficarlos y comprobar relaciones que se cumplen (por ejemplo, que las mediatrices en un triángulo se cortan en el mismo punto y que es centro de la circunferencia que lo circunscribe). Para ello WallE cuenta con un lienzo (plano), una regla y un compás para trazar rectas y circunferencias. Estas herramientas no tienen grabadas las medidas por lo que WallE nunca sabe la distancia real entre dos puntos.

ENTIDADES

El valor más simple que puede ser representado es un punto en el plano. WallE no tiene noción del tamaño del plano, ni de las coordenadas reales de un punto. No obstante, gracias a su regla y su compás puede determinar otros puntos de interés realizando algunas construcciones auxiliares.

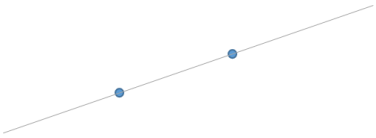
Por ejemplo, dados dos puntos p_1 y p_2, WallE podría obtener el punto medio de la siguiente forma:

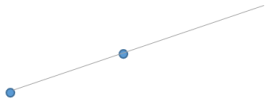


1- Se traza una recta entre los puntos
2- Configura el compás con los dos puntos (no es necesario conocer la distancia real, simplemente se ponen las puntas del compás en cada punto) y se traza una circunferencia con centro en p_1.
3- Se traza una segunda circunferencia con centro en p_2, sin cambiar la medida del compás.
4- Se obtienen los puntos de intersección entre las dos circunferencias.
5- Se traza una recta entre dichos puntos.
6- Se obtiene el punto de intersección entre las dos rectas.

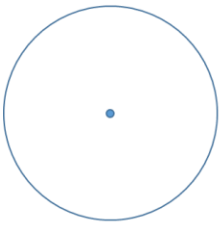
Con las herramientas básicas de regla y compás se pueden obtener (a partir de un conjunto de puntos iniciales) objetos más complejos como rectas, rayos, segmentos, circunferencias y arcos.

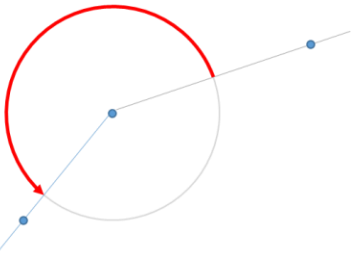
Una **recta** queda determinada por dos puntos distintos en el plano. Si se considera solamente los puntos que están en una dirección determinada de un punto se obtiene un **rayo**. Si sólo se consideran los puntos entre los puntos determinantes de la recta, se obtiene un **segmento**.



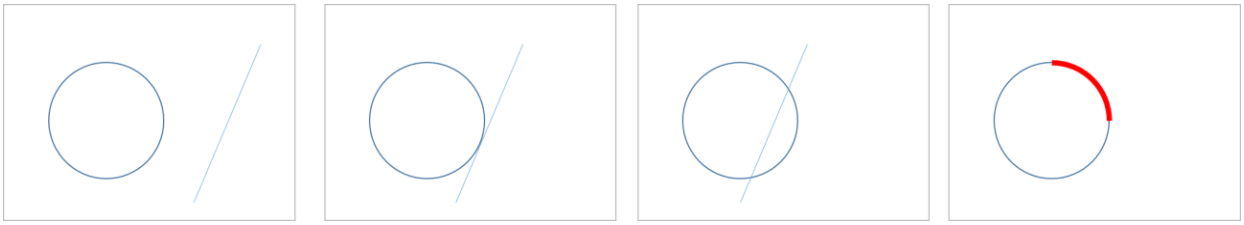


Una **circunferencia** puede ser creada a partir de un punto centro y una medida configurada en el compás. Un arco nos permite referirnos a la porción delimitada por una semi-recta inicial y una semi-recta final (seguiremos como convenio que el trazo se realiza en contra de las manecillas del reloj).





Entre todos estos elementos se puede chequear la intersección. Esta puede ser vacía (los objetos no se interceptan), un solo punto, dos puntos o infinitos puntos.



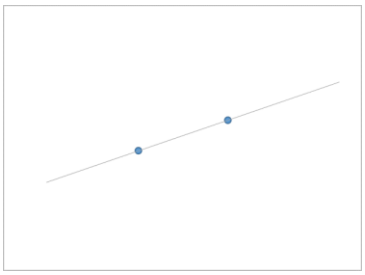
LENGUAJE G#

Para especificar las acciones que deberá realizar WallE para su análisis geométrico se utilizará un lenguaje de programación denominado G#. El lenguaje G# es un lenguaje funcional que no permite VARIABLES[1]. Un programa en G# se compone de una lista de instrucciones separadas por ';'. Las instrucciones permiten recibir argumentos de entrada, importar otros códigos, definir funciones o constantes, configurar características del visor y dibujar objetos geométricos.

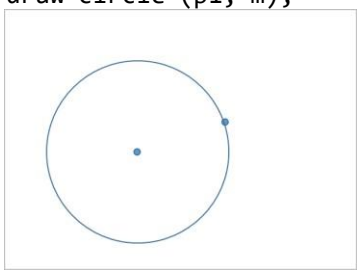
La siguiente tabla muestra algunos programas y sus correspondientes "salidas".

```
point p1;
point  p2;
draw {p1, p2};
```

```
point p1;
point  p2;
draw {p1, p2};
draw line(p1, p2);
```



```
point p1;
point  p2;
draw {p1, p2};
m = measure(p1,p2);
draw circle (p1, m);
```



Durante la evaluación de un programa es posible graficar otros objetos. Algunas instrucciones de "depuración" pueden ser intercaladas en las definiciones de un programa para que grafiquen algunos conceptos a la par que se evalúan las expresiones.

EL PROGRAMA

Un programa en G# es un conjunto de instrucciones. El conjunto de instrucciones vacío es un programa válido y no hace nada. Las instrucciones de entrada permiten declarar argumentos que serán provistos por la aplicación. Por ejemplo, la siguiente línea de código indica que la aplicación deberá proveer un punto que será tratado internamente como `p1`.

```
point p1;
```

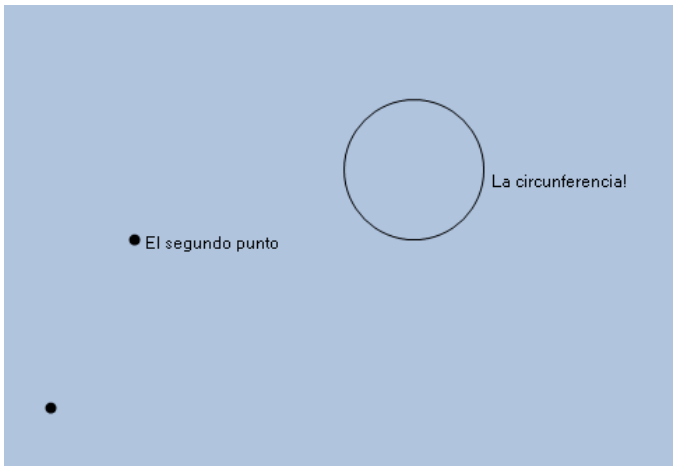
Otros objetos que pueden ser requeridos de la aplicación son `line`, `segment`, `ray`, `circle` y `arc`. Por ejemplo, el código más adelante indica que se reciben dos puntos y una circunferencia.

No pueden existir dos argumentos con igual nombre y de distintos tipos. Los argumentos pueden requerirse en cualquier lugar del código pero serán provistos por la aplicación antes de la ejecución, sin importar si la ejecución termina requiriéndolos finalmente o no (por ejemplo, están dentro de una función que no se evalúa nunca). Un argumento se puede pedir varias veces en un mismo ámbito, pero no se puede referir si no está declarado en el ámbito actual o en algún ámbito padre. Por ejemplo, una función puede indicar que hace uso de un punto recibido como argumento de nombre p1, y luego en otro ámbito fuera de la función si se requiere trabajar con dicho punto se deberá indicar nuevamente.

[1] al menos no las variables que acostumbramos a utilizar en un lenguaje de programación como C#

Para dibujar un objeto en el lienzo se utiliza el comando **draw**. Este comando recibe un objeto a ser dibujado y opcionalmente una etiqueta a ser dibujada en alguna posición aleatoria dentro del objeto. El siguiente código muestra un programa dibujando dos puntos y una circunferencia.

```
// argumentos de entrada
point p1;
point p2;
circle c;
// tiempo de dibujar
draw p1;
draw p2 “El segundo punto”;
draw c “La circunferencia!”;
```

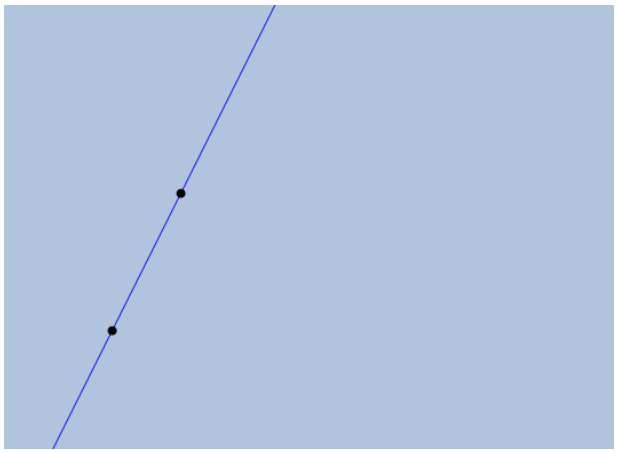


Los comandos `color` y `restore` permite escoger un color temporalmente a ser utilizado para los dibujos y luego restaurar el estado al color anterior. Los colores posibles son: blue, red, yellow, green, cyan, magenta,white, gray, black. En un inicio el color con que se dibuja en el lienzo es negro. Si se han restaurado todos los colores “apilados” con el comando `color`, entonces `restore` no tiene efecto.

```
point p1;
point p2;

color blue;
draw line(p1, p2);
restore;

draw p1;
draw p2;
```



En algunos casos conviene declarar variables temporales para almacenar resultados intermedios que pueden ser utilizados una o más veces posteriormente. Para ello se pueden definir en G# constantes. El nombre de constante se lo damos porque en realidad toman valor en un único momento y no pueden ser asignadas nuevamente. Las constantes no pueden tener el mismo nombre de un argumento ni de otras constantes en ámbitos “padres”.

```
l = line(p1, p2);
```

Si luego de esta asignación se intenta asignar nuevamente la constante debe producirse un error de compilación.

SECUENCIAS

El concepto de secuencia está presente en el lenguaje G#. Se pueden declarar de forma literal utilizando llaves. Por ejemplo, la siguiente línea manda a dibujar una secuencia de puntos con un único llamado.

```
draw {p1, p2}
```

Algunas funciones intrínsecas del lenguaje como `intersect` devuelven una secuencia de valores (en este caso de puntos). Por ejemplo, la intersección entre dos circunferencias puede resultar en dos, uno o ningún punto de intersección.

```
sec = intersect (circle1, circle2);
```

Para consultar los valores que están contenidos en una secuencia se utiliza la declaración de *match*:

```
valor0, valor1, …, resto = secuencia;
```

Esto asigna a las primeras constantes los primeros valores de la secuencia y en la última constante la secuencia de los valores restantes. Por ejemplo, quedarse con el primer elemento de una secuencia "miSecuencia" en una constante "a" puede hacerse de la forma:

```
a, rest = miSecuencia;
```

La constante `rest` se refiere al resto de la secuencia y a la constante "`a`" se le ha asignado el primer valor de la secuencia. Si no se desea trabajar con el resto se puede utilizar el comodín *underscore* (_).

```
a, _ = miSecuencia;
```

Note que hacer:

```
a = miSecuencia;
```

es válido pero únicamente se está obteniendo una nueva constante refiriéndose a la misma secuencia.

Algunos ejemplos útiles:

```
a,b,_ = intersect(c1, c2); // obtiene los puntos de intersección entre dos circunferencias en a y b
_,t = { 1, 2, 3 }; // en t se almacena la secuencia { 2, 3 }
_,_ = { 2, 3, 4 }; // es válido pero no realiza nada en este caso
_ = { 2, 3 }; // es válido también
```

Si se pide valores a una secuencia que no tiene, las constantes declaradas en la asignación obtienen valor `undefined`.

El siguiente ejemplo termina evaluando `p1` en `a` y `undefined` en `b`.

```
a,b,_ = { p1 };
```

La constante que toma el valor del resto (última constante de la asignación) nunca es `undefined`, sino que toma valor de secuencia vacía (`{}`). Si la secuencia es `undefined` entonces todos los valores toman valor `undefined`, incluido el resto de la secuencia.

```
a,b,rest = { p1 }; // a es p1, b es undefined y rest es {}

a,b,rest = undefined; // a, b y rest son undefined

a,b,rest = intersect(circle1, circle1); // idem, la intersección entre dos circunferencias
iguales es undefined
```

Dada una secuencia se puede conocer cuántos elementos tiene utilizando la función intrínseca `count`. Si la secuencia es infinita (`undefined` o realmente infinita) esta función evalúa `undefined`.

SECUENCIAS INFINITAS

Si se desea un rango de valores enteros se puede utilizar la notación: `{ a … b }` donde `a` y `b` son valores constantes enteros. Si se desea empezar en a y obtener todos los valores sucesivos se puede escribir `{ a … }`.

Por ejemplo, la secuencia:

```
seq = { 1 ... };
```

se refiere a todos los valores naturales.

Otras secuencias infinitas se pueden obtener de las funciones intrínsecas:

```
points(f) // devuelve una secuencia de puntos aleatorios en una figura
samples() // devuelve una secuencia de puntos aleatorios en el lienzo
randoms() // devuelve una secuencia de valores aleatorios enteros positivos
```

Estas secuencias son aleatorias pero si se refiere a sus valores… es decir, no generan un valor distinto cada vez que se refiere a su primer elemento.

```
a,_ = randoms();
b,_ = randoms(); // b tiene igual valor que a
```

Si se hace

```
a, b, _ = randoms(); // es muy probable que a y b sean distintos
```

Dos secuencias pueden ser operadas con + para concatenar. El resultado es una secuencia que comienza con los elementos de la primera y al finalizar (si esta finaliza) continúa con los siguientes valores. Si la primera secuencia es undefined la concatenación es undefined, pero si la primera no tiene problemas, se pueden tomar sus valores en la concatenación.

```
a, rest = { 1, 2, 3 } + undefined; // a es 1 y rest es { 2, 3 } + undefined

a, rest = undefined + { 1, 2, 3 }; // a y rest son undefined
```

VALORES NUMÉRICOS

Las expresiones numéricas representan valores básicos en G#. Los siguientes son ejemplos de valores reales expresables en el lenguaje.

```
0.1, -14, 2023
```

Entre estos se pueden utilizar operadores aritméticos (+,-,*,/,%) y de comparación (<, <=, >, >=, ==, !=) para definir expresiones más complejas.

```
2+4*(5-4)
```

HACIENDO MEDICIONES

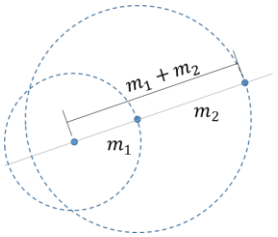
WallE no conoce las coordenadas de un punto, pero puede utilizar el compás para señalar una medida entre dos puntos. Esta medida puede utilizarse para trazar una circunferencia en cualquier posición, para comparar con otra medida (si se tiene que abrir más o menos el compás para encajar con otro par de puntos).

El objeto medida (`measure`) se puede crear a partir de dos puntos:

```
m = measure(p1, p2);
```

Dos medidas pueden compararse utilizando los operadores de comparación: <, <=, >, >=, ==, !=.

Dadas dos medidas también se puede obtener una medida que sea la suma (basta con trazar una recta y ubicar dos circunferencias, primero una con radio igual a la primera medida, y luego otra ubicada en una de las intersecciones con radio igual a la segunda).



La diferencia entre dos medida es también posible (siendo siempre “positiva”). Son válidas también las operaciones siguientes:

`measure * natural = measure` (Qué medida se obtiene replicando un número de veces una medida)

`measure / measure = natural` (Cuántas veces cabe una medida dentro de otra)

En la multiplicación se utilizará una versión del valor, entera y positiva. Es decir, `m * -2.3` es equivalente a: m * 2 (suponiendo que m es de tipo `measure`).

CONDICIONALES

La expresión

```
if <expr> then <expr> else <expr>
```

Permite expresar valores condicionales (similar al operador ternario `?:` en C#). La expresión de la cláusula `if` se evalúa. Si es "verdadera" la expresión completa tiene el valor de la cláusula `then`, si no, tiene el valor de la cláusula `else`.

En G# no existen los valores booleanos pero en su lugar se puede utilizar cualquier objeto o número. Las expresiones que evalúan Falso son las siguientes:

0	Valor numérico
{}	Secuencia vacía
undefined	Valor de un objeto que no está bien definido.

Cualquier otro valor evalúa verdadero.

Las expresiones pueden ser operadas con **and**, **or** y **not** devolviendo 1 en caso de evaluar verdadero y 0 en caso de evaluar falso.

ENCAPSULANDO CON FUNCIONES

En G# se pueden definir funciones de la siguiente forma:

```
<identificador>(<lista de parámetros>) = <expresión>;
```

Por ejemplo, la siguiente función le resultará familiar.

```
Fib(n) = if n <= 1 then 1 else Fib(n-1) + Fib(n-2);
```

Ahora, supongamos que queremos representar el conjunto de pasos necesarios para obtener el punto medio entre dos puntos descrito al inicio del documento.

```
point p1;
point p2;

l1 = line(p1, p2);
m = measure (p1, p2);
c1 = circle (p1, m);
c2 = circle (p2, m);
i1,i2,_ = intersect(c1, c2);
l2 = line(i1, i2);
medio,_ = intersect(l1, l2);

draw {p1, p2};

color red;
draw medio;
restore;
```

Si se desea calcular nuevamente el punto medio entre otros dos puntos se necesitaría declarar nuevamente estos pasos sustituyendo por los puntos nuevos. Estos cálculos pueden ser encapsulados en una función `puntoMedio` que reciba los puntos y devuelva el valor del punto medio computado.

Como el cuerpo de una función se forma de una única expresión se necesita un tipo de expresión que permita tener varias acciones antes de evaluar la expresión final.

EXPRESIÓN *LET IN*

La expresión `let in` tiene la siguiente sintáxis:

```
let
<lista de instrucciones terminadas con ;>
in <expresión>
```

Esta construcción permite definir un contexto propio en el que se pueden tener nuevas definiciones de constantes o funciones a ser utilizadas en la cláusula **in**. Por ejemplo:

```
let
  a = 4;
  b = 5;
in a + b
```

Como `let in` es una expresión, esta puede ser ubicada en cualquier lugar donde una expresión se necesita.

```
if let a = 4; b = 5; in a + b then 1 else 2
```

La prioridad del `if then else` es igual que la del `let` y es la más baja de todas las expresiones, por tanto:

```
if a then b else c + 4
```

significa que la cláusula `else` es `c+4` en lugar que a toda la expresión `if then else` se le suma `4`. Lo mismo pasa con la expresión:

```
let c = 3; in c + 4
```

No obstante los paréntesis pueden ser utilizado para desambiguar en caso necesario.

```
(let c = 3; in c) + 4
```

Los lets se pueden anidar si se necesita que una constante tenga un valor definido con otra expresión `let`.

```
let
  a = let
        b = 4;
      in b + 2;
  in a + 5
```

Las constantes no pueden redefinirse en ambientes anidados para que no parezca que se les está pudiendo cambiar el valor. El siguiente código debe dar un error en compilación.

```
 let
  a = 5;
  b = let
        a = 4; // Error, redefinición de la constante a.
      in a + 2;
  in a + b
```

En un `let` anidado se puede utilizar cualquier constante declarada en un `let` "padre" excepto aquella a la que se le está siendo asignada la expresión con el `let` involucrado. Es decir, el siguiente código no es válido.

```
let
  a = let
        b = 4;
      in b + a; // Error, la constante 'a' no está definida
  in a + 5
```

Una vez que un let se evalúa, las constantes que se hayan definido en su ámbito no pueden ser leídas en el ámbito padre:

```
let
  a = let
        b = 4;
      in b + 2;
  in a + b // Error, la constante 'b' no está definida
```

Los nombres de las constantes declaradas en un `let` sí pueden ser reusadas en un `let` que no sea padre.

```
let
  a = let
        c = 4
      in c
  b = let
        c = 5 // válido
      in c
  in a + b
```

Pero en un ámbito se pueden utilizar nombres de constantes declaradas en ámbitos hijos si se usan luego que estos hayan sido evaluados. Es decir:

```
let
  a = let
        b = 4
      in b
  b = let // Es válido porque está declarada luego del hijo que la usa
        c = 5
      in c
  in a + b
```

Uno de los usos más comunes de las expresiones `let` será para permitir encapsular un conjunto de pasos en una función, por ejemplo, la siguiente función evalúa el punto medio de dos puntos.

```
mediatriz(p1, p2) =
    let
       l1 = line(p1, p2);
       m = measure (p1, p2);
       c1 = circle (p1, m);
       c2 = circle (p2, m);
       i1,i2,_ = intersect(c1, c2);
       l2 = line(i1, i2);
    in l2;

puntoMedio(p1,p2)=
    let
       medio,_ = intersect(line(p1,p2), mediatriz(p1,p2));
    in medio;
```

CHEQUEO DE TIPOS

El lenguaje G# tiene un fuerte chequeo de tipos y debe descubrir tempranamente (en compilación), cualquier posible conflicto con los tipos de sus expresiones y las operaciones que pueden hacerse sobre estas. Por ejemplo, no tiene sentido interceptar dos números, o sumar dos circunferencias (al menos no en este lenguaje).

Para ello en el lenguaje se debe realizar una inferencia de los tipos de las expresiones. A diferencia de otros lenguajes en los que las variables, parámetros y retornos de función tienen declarado los tipos, en G# solo se conoce los tipos de los argumentos de la aplicación (point, line, segment, etc.) y de las funciones intrínsecas (intersect, count, randoms, etc.).

No obstante, esto es suficiente para poder inferir la compatibilidad de tipo de las expresiones y si son válidas o no las operaciones sobre estas. Por ejemplo:

```
point p1;
a = p1; // a se infiere que es de tipo point
b = a + 2; // Error, no puede sumar un punto y un número
```

Algunos chequeos particulares. La secuencia { } es compatible con cualquier tipo de secuencia. El valor `undefined` es válido con cualquier tipo de objeto (point, segment, line, secuencias, etc.) o número.

La expresión `if then else` deberá devolver el mismo tipo de valores en la cláusula `then` y en la cláusula `else`.

INFERENCIA DE TIPOS EN FUNCIONES [OPCIONAL]

La inferencia de los parámetros en una función y su tipo de retorno es la parte más complicada. No obstante, en G# la compatibilidad de tipos está determinada por el funcionamiento. Por ejemplo, ¿qué tipo pudiera inferirse para el parámetro y el retorno de la siguiente función?

```
identity(n) = n;
```

Está claro que cualquier tipo es válido para n y que éste determinaría el tipo de retorno. Es decir:

```
identity(4) + 5 //es un número

intersect(identity(circle1), circle2) // es válido
```

pero

```
identity(circle) + 5 // Error, una circunferencia no puede operarse con un número.
```

Si ayuda para algo, vea la similitud entre la especificación del tipo de `n` en la función `identity` y lo que se pudiera tener en C# con la genericidad.

```
T identity<T> (T n) { return n; }
```

Las secuencias son siempre del mismo tipo de elementos. Es decir, `{ 1, circle }` no es una secuencia válida. La operación de match es únicamente válida para secuencias por lo que si se tiene algo como:

```
first(s) = let
              f,_ = s;
           in f;
```

Se tiene algo similar que en `identity`. La función `first` recibe una secuencia pero "genérica", es decir, el tipo de retorno de `first` dependerá de la secuencia que esté recibiendo como parámetro.

```
first({1, 2, 3}) // es un número
```

mientras que

```
first({ circle1, circle2, circle3 }) // es una circunferencia
```

pero

```
first (4) // Error, s debe ser una secuencia
```

Sin embargo, si en la función `first` hacemos algún uso de un elemento de s restringimos esa genericidad.

```
first(s) = let
              f,_ = s;
              a = f + 4;// f se infiere número y por tanto el tipo de s secuencia de números
           in f;
```

Con este cambio, el uso:

```
first({ circle1, circle2, circle3 }) // Error, s debe ser tipo secuencia de números
```

BIBLIOTECAS

Desde un programa de puede importar las instrucciones declaradas en otros ficheros mediante la instrucción:

```
import "NombreDeFichero.geo"
```

Este tipo de instrucción deberá aparecer al inicio del programa únicamente. Si una biblioteca importada tiene definiciones repetidas, se debe producir un error de compilación. Si se importan dos veces un mismo fichero, este deberá ser incluido una única vez (la primera).

Suponga los ficheros A, B, C y D. En B se importa A, en C se importa A y en D se importa B y C. El orden en que se incluyen las definiciones será: A, B, C, D.

COMANDOS Y FUNCIONES

Comando	Descripción
`point <id>`	Declara que se recibe un argumento de tipo punto con nombre <id>
`line <id>`	Declara que se recibe un argumento de tipo recta con nombre <id>
`segment <id>`	Declara que se recibe un argumento de tipo segmento con nombre <id>
`ray <id>`	Declara que se recibe un argumento de tipo semirecta con nombre <id>
`circle <id>`	Declara que se recibe un argumento de tipo circunferencia con nombre <id>
`point sequence <id>`	Declara que se recibe un argumento de tipo secuencia de puntos con nombre <id>
`line sequence <id>`	…
…	
`color`	Establece el color a ser utilizado
`restore`	Restablece el color anterior
`import <string>`	Incluye en el programa actual las definiciones del fichero indicado.
`draw <exp> <string>`	Dibuja el o los objetos definidos en <exp>
`line(p1,p2)`	Devuelve una recta que pasa por los puntos p1 y p2.
`segment(p1,p2)`	Devuelve un segmento con extremos en los puntos p1 y p2.
`ray(p1,p2)`	Devuelve una semirecta que comienza en p1 y pasa por p2.
`arc(p1,p2,p3,m)`	Devuelve un arco que tiene centro en p1, se extiende desde una semirecta que pasa por p2 hasta una semirecta que pasa por p3 y tiene medida m.
`circle(p,m)`	Devuelve una circunferencia con centro en p y medida m.
`measure(p1,p2)`	Devuelve una medida entre los puntos p1 y p2.
`intersect(f1,f2)`	Intersecta dos figuras (puntos, rectas, etc.) y devuelve la secuencia de puntos de intersección. Si la intersección coincide en infinitos puntos devuelve `undefined`.
`count(s)`	Devuelve la cantidad de elementos de una secuencia. Si la secuencia es infinita devuelve `undefined`.
`randoms()`	Devuelve una secuencia de valores aleatorios numéricos entre 0 y 1.
`points(f)`	Devuelve una secuencia de puntos aleatorios en una figura.
`samples()`	Devuelve una secuencia de puntos aleatorios en el lienzo.

LA APLICACIÓN

La aplicación visual debe permitir diseñar programáticamente un conjunto de pasos para obtener o dibujar figuras.

Todos los programas deberán poder guardarse en ficheros para un futuro uso. Los códigos que se hayan creado como bibliotecas también deberán salvarse.

En este documento se exponen un conjunto pequeño de instrucciones, objetos y funciones. No se conforme con ello… Proporcione nuevos tipos de funciones y tipos de objetos. Tenga en cuenta que un gran peso de la evaluación lo determina la extensibilidad propuesta. Es decir, cuán fácil es incorporar un nuevo operador, comando, objeto, etc.

Propóngase como meta hacer su solución lo suficientemente extensible como para que se pueda incorporar sin mucha dificultad un posible nuevo operador para la potencia (2^3 == 8), o un concepto nuevo como filtros.

```
{ a1, …, an } where x > 0
```

¡Sea creativo!

No obstante, no puede obviar ninguno de los comandos descritos en este documento puesto que su proyecto será evaluado con códigos propuestos por los profesores.

SOBRE LEGIBILIDAD DEL CÓDIGO

IDENTIFICADORES

Todos los identificadores (nombres de variables, métodos, clases, etc) deben ser establecidos cuidadosamente, con el objetivo de que una persona distinta del programador original pueda comprender fácilmente para qué se emplea cada uno.

Los nombres de las variables deben indicar con la mayor exactitud posible la información que se almacena en ellas. Por ejemplo, si en una variable se almacena "la cantidad de obstáculos que se ha encontrado hasta el momento", su nombre debería ser `cantidadObstaculosEncontrados` o `cantObstaculos` si el primero le parece demasiado largo, pero nunca ~~`cOb`~~, ~~`aux`~~, ~~`temp`~~, ~~`miVariable`~~, ~~`juan`~~, ~~`contador`~~, ~~`contando`~~ o ~~`paraQueNoChoque`~~. Note que el último identificador incorrecto es perfectamente legible, pero no indica "qué información se guarda en la variable", sino quizás "para qué utilizo la información que almaceno ahí", lo cual tampoco es lo deseado.

- Entre un nombre de variable un poco largo y descriptivo y uno que no pueda ser fácilmente comprensible por cualquiera, es preferible el largo.
- Como regla general, los nombres de variables **no** deben ser palabras o frases que indiquen acciones, como ~~eliminando~~, ~~saltar~~ o ~~parar.~~

Existen algunos (muy pocos casos) en que se pueden emplear identificadores no tan descriptivos para las variables. Se trata generalmente de pequeños fragmentos de código muy comunes que "todo el mundo sabe para qué son". Por ejemplo:

```
int temp = a;
a = b;
b = temp;
```

"Todo el mundo" sabe que el código anterior constituye un intercambio o *swap* entre los valores de las variables a y b, así como que la variable `temp` se emplea para almacenar por un instante uno de los dos valores. En casi cualquier otro contexto, utilizar `temp` como nombre de variable resulta incorrecto, ya que solo indica que se empleará para almacenar "temporalmente" un valor y en definitiva todas las variables se utilizan para eso.

Como segundo ejemplo, si se quiere ejecutar algo diez veces, se puede hacer

```
for (int i = 0; i < 10; i++)
  ...
```

en lugar de

```
for (int iteracionActual = 0; iteracionActual < 10; iteracionActual++)
  ...
```

Para los nombres de las propiedades (*properties* en inglés) se aplica el mismo principio que para las variables, o sea, expresar "qué devuelven" o "qué representan", solo que los identificadores deben comenzar por mayúsculas. No deben ser frases que denoten acciones, abreviaturas incomprensibles, etc.

Los nombres de los métodos deben reflejar “qué hace el método” y generalmente es una buena idea utilizar para ello un verbo en infinitivo o imperativo: Agregar, Eliminar, ConcatenarArrays, ContarPalabras, Arranca, Para, etc.

En el caso de las clases, obviamente, también se espera que sus identificadores dejen claro qué representa la clase: Robot, Obstaculo, Ambiente, Programa.

COMENTARIOS

Los comentarios también son un elemento esencial en la comprensión del código por una persona que lo necesite adaptar o arreglar y que no necesariamente fue quien lo programó o no lo hizo recientemente. Al incluir comentarios en su código, tome en cuenta que no van dirigidos solo a Ud., sino a cualquier programador. Por ejemplo, a lo mejor a Ud. le basta con el siguiente comentario para entender qué hace determinado fragmento de código o para qué se emplea una variable:

// lo que se me ocurrió aquel día

Evidentemente, a otra persona no le resultarán muy útiles esos comentarios.

Algunas recomendaciones sobre dónde incluir comentarios

- Al declarar una variable, si incluso empleando un buen nombre para ella pueden quedar dudas sobre la información que almacena o la forma en que se utiliza
- Prácticamente en la definición de todos los métodos para indicar qué hacen, las características de los parámetros que reciben y el resultado que devuelven
- En el interior de los métodos que no sean demasiado breves, para indicar qué hace cada parte del método

Es cierto que siempre resulta difícil determinar dentro del código qué es lo obvio y qué es lo que requiere ser comentado, especialmente para Ud. que probablemente no tiene mucha experiencia programando y trabajando con código hecho por otras personas. Es preferible entonces que “por si acaso” comente su código lo más posible.

Otro aspecto a tener en cuenta es que los comentarios son fragmentos de texto en lenguaje natural, en los cuales deberá expresarse lo más claramente posible, cuidando la ortografía, gramática, coherencia, y demás elementos indispensables para escribir correctamente.

Todos estos elementos son importantes para la calidad de todo el código que produzca a lo largo de su carrera, pero además **tendrán un peso considerable en la evaluación de su proyecto de programación**.