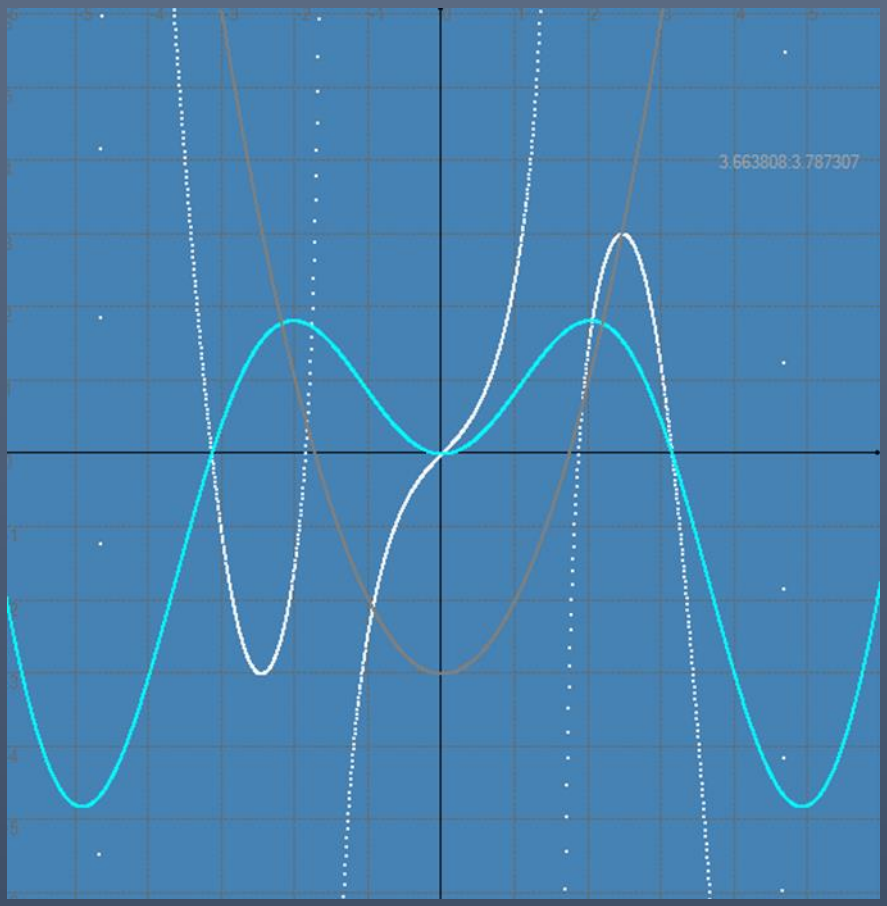
2015



Plotter

SEGUNDO PROYECTO DE PROGRAMACIÓN

COLECTIVO DE PROGRAMACIÓN

PLOTTER

Los matemáticos se han cansado de graficar funciones a mano y han pedido la colaboración de los programadores para que diseñen e implementen un ambiente de desarrollo en el que puedan escribir sus fórmulas y visualizar sus funciones. La aplicación deberá tener áreas para la edición de código, la consola de entrada y salida, y un área en que se visualicen los gráficos de las funciones.

El diseño de clases debe ser suficientemente flexible para incorporar nuevas funciones, instrucciones, operadores y tipos de expresiones en un futuro.

EL LENGUAJE M# (M DE MATEMÁTICA)

El lenguaje M# tiene una sintaxis simple con la cual un programa se representa como un conjunto de instrucciones para declarar, leer, imprimir o terminar la ejecución. Las variables son de un único tipo "numérico" aunque existen expresiones condicionales y texto.

LITERALES

Como en C# los literales permiten expresar valores que son prefijados en el código. Por ejemplo: 5, -4, 3.14159, "Hola". Para el caso numérico el tipo del literal siempre es **number**, a diferencia de en C# en el que 5 es un literal de tipo **int** y 5.0 un literal de tipo **double**. Los textos encerrados entre comillas dobles (i.e. "Entre un número") son literales de tipo **string**.

VARIABLES

Los nombres de las variables pueden estar formados por letras y números siempre empezando en letra. Por ejemplo: `a`, `a1`, `miVariable`, `mi2daVariable`.

Para declarar una variable se utiliza la instrucción **let**. Para asignar un valor a una variable se escribe:

```
let <var> = <expresión>
```

Ejemplos:

```
let a = 4

let x1 = 3+a

let y=f(4)*2+1.
```

Para declarar una variable no se necesita especificar de qué tipo es porque siempre será de tipo **number**. No se puede declarar una variable sin asignarle un valor y no se puede utilizar una variable sin haberla declarado con anterioridad. Si la variable ya está definida la nueva declaración produce un error. Ejemplo:

```
let a = 4

…

let a = 1000   (<< Error!)
```

No obstante con la instrucción de asignación (**set**) se puede cambia el valor:

```
set a = 1000

print a       (aquí se imprime 1000)
```

OPERADORES

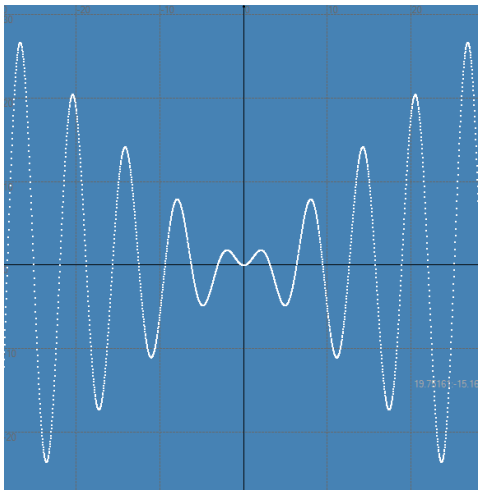
Se deben permitir los operadores comúnmente utilizados en el álgebra, como son: suma (+), resta (-), multiplicación (*), división (/), modulo (%), potencia (^), negación (-), positivo (+), negación lógica (!), el "y" lógico (&), y el "o" lógico (|). Además existen algunos operadores de comparación que evalúan tipo boolean.

Algunos operadores tienen cierta prioridad de evaluación sobre otros. Por ejemplo, la multiplicación deberá realizarse siempre antes que la suma. El orden de evaluación de los operadores básicos que usted deberá soportar en su lenguaje aparece en la Tabla 1. Los operadores que están dentro del mismo bloque tienen igual prioridad y se evalúan dependiendo del orden en que aparecen.

()	Agrupadores para expresiones. Ejemplo: 5*((2+x)/2)
‘	Derivación
^	Potencia. Ejemplo: 4^2 (4 al cuadrado)
* /	Multiplicación y división binaria
+ -	Operadores unarios de positivo o negativo. Ejemplo: -3+5 2-+2
+ -	Operadores binarios de suma o resta. Ejemplo: 3+4-5
< <= > >= != ==	Operadores de comparación Ejemplo: x<4 y>=x+5 x!=3
&	Operador lógico and. Ejemplo: -2<x & x<=4
|	Operador lógico or. Ejemplo: X<=-4 | x > 5
if else	Operador condicional (<exp1> **if** <condicional> **else** <exp2>) Ejemplo: def sign (x) = 1 if x > 0 else -1 if x < 0 else 0

Tabla 1. Operadores y sus prioridades

Los operadores aritméticos (+, -, *, /, %, ^) operan sobre expresiones numéricas y devuelven un valor numérico. Los de comparación operan sobre valores numéricos y devuelven un valor booleano. Los lógicos operan sobre valores booleanos y devuelven un valor booleano.



FUNCIONES

Como se ha ilustrado, el programa permite definir y evaluar funciones. Existen un conjunto de funciones predefinidas y que no pueden ser "redefinidas". Tal es el caso de las funciones: **sin**, **cos**, **tan**, **atan**, **cot**, **arcsin**, **arccos**, **exp**, **ln**, **abs**, que usted deberá implementar como parte de su aplicación. Las funciones homónimas del tipo **System.Math** permiten saber los argumentos que reciben estas funciones y lo que devuelven. Todas las funciones son de $\mathbb{R} \longrightarrow \mathbb{R}$, tanto las predefinidas en el lenguaje como las definidas por el usuario.

Para definir una función propia del usuario este debe utilizar la instrucción **def**. La sintaxis es:

```
def <función> ( <parámetro> ) = <expresión>
```

Ejemplos:

```
def f ( x ) = x * 4 + 1
```

```
def g ( a ) = a * (a + 1)
```

GRAFICANDO

Su aplicación deberá permitir visualizar funciones. Para ello se utilizará la instrucción `graph`. Esta instrucción está precedida de una expresión que tiene por defecto un parámetro con nombre `x` que es la variable libre que se evaluará (valores del eje x)[1].

Ejemplo:

```
def f(x) = x * sin (x)
graph f(x)
```

Es equivalente a:

```
graph x * sin (x)
```

Note que la instrucción:

```
graph 4
```

Es válida y visualiza la función constante f(x)=4.

[1] Usted podrá contar con un proyecto que brinda la implementación de un visor de funciones. Este visor permitirá fijar el rango de valores de x, mover el sistema de coordenadas, mostrar las expresiones que están siendo graficadas, cambiar colores, etc.

DERIVADAS

En cualquier expresión en un contexto donde exista un parámetro se puede utilizar el operador derivada (**'**). Este operador resuelve la derivada con respecto al parámetro. Este operador tiene mayor prioridad que los expuestos anteriormente. Note que hasta el momento los lugares donde existen parámetros son el cuerpo de una función (el parámetro de la función) o la expresión del comando **graph** (el parámetro especial **x**).

Ejemplos:

```
def f(x)=sin(x)' + cos(x)
```

```
def g(x)=f(x)'' // Las derivadas segundas se obtienen de aplicar derivada a una derivada.
```

```
graph (x^2+2*x+1)' // Se puede aplicar derivada en esta expresión porque existe el parámetro especial x.
```

CONTEXTOS

En M# existen sólo dos contextos, global y local. El contexto global es en el que se definen las funciones y las variables. El contexto local es el cuerpo de las funciones o de la instrucción **graph**. Si una variable tiene igual nombre que un parámetro, se tomará el valor del parámetro en lugar del de la variable global. Ejemplo:

```
let x = 5
def f (x) = x  << esta x se refiere al parámetro y no a la variable global
```

ENTRADA Y SALIDA

La aplicación tiene una consola que permite mostrar y recibir valores del usuario. Para ello se utilizan las instrucciones **print** y **read**.

La instrucción **print** recibe la expresión cuyo valor será mostrado en la salida. La instrucción **read** recibe la variable en la que almacenará el valor introducido por el usuario. La entrada siempre será un número.

Ejemplo:

```
let n = 0
print "Entre el valor de n"
read n
def f(x)=x + n
```

MÓDULOS

Cada código editado se almacenará en un fichero con extensión **.mat**. Este código representa un módulo. Desde un módulo se puede "invocar" la ejecución de otro módulo con la instrucción **include "<modulo>"**.

Suponga que existe un fichero en el mismo directorio de trabajo que el módulo actual con una lógica como se muestra:

MÓDULO "FUNCIONPARAMETRIZADA.MAT"

```
let n = 0
let m = 1
f ( x ) = m * x + n
```

En el módulo actual se puede incluir estas definiciones de la forma.

MÓDULO "MI NUEVO PROGRAMA.MAT"

```
include "FuncionParametrizada"
set n = 1
set m = 2
print f ( 4 )  // << imprime 9
```

Note que en el punto en que se asigna 1 a la variable n ya se definió con anterioridad al incluir el otro módulo. Se debe detectar dependencias cíclicas entre los módulos y producir un error en consecuencia. Si al incluir un módulo se incurre en la re-declaración de una variable o función debe producirse un error.

SOBRE EL DISEÑO DE CLASES

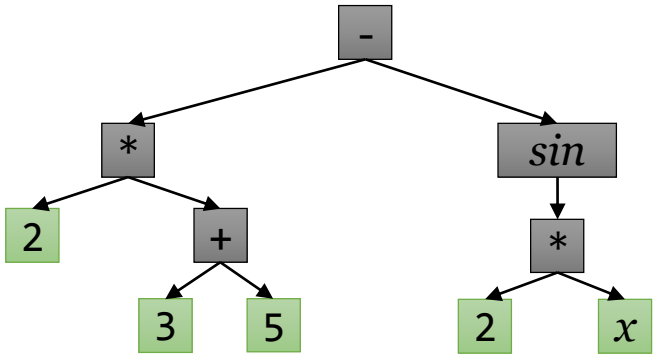
Para definir el conjunto de tipos que le permitan modelar esta aplicación tenga en cuenta lo siguiente:

1º. No vincular los tipos que permiten modelar el concepto de expresión, los mecanismos de evaluarla, las funciones, las instrucciones y demás, con la aplicación final que Usted propone. Piense que todo el esfuerzo que usted realice para resolver esas tareas debe ser fácilmente aprovechable para crear otras aplicaciones distintas a la suya. Ejemplo: una aplicación de consola, una aplicación web.

2º. Debe ser posible incorporar nuevos operadores y funciones predefinidas a su proyecto. Al punto de poder escribir: `Operadores.Add (new OperadorÑañarita ())` y una vez compilado nuevamente su proyecto el operador agregado pueda ser utilizado.

3º. Considere las facilidades que puede brindar trabajar una expresión como muestra la figura:

2 * (3 + 5) – sin (2 * x)



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: Function, Expression, Program, Console.

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**.