Herramientas en GNU/Linux para estudiantes universitarios

La herramienta de cálculo científico YACAS

José Angel de Bustos Pérez

Herramientas en GNU/Linux para estudiantes universitarios: La herramienta de cálculo científico YACAS

por José Angel de Bustos Pérez

Tabla de contenidos

Lista de tablas

Tabla de ejemplos

Capítulo 1. Introducción a YACAS

1.1. Organización de este documento

Este documento trata el uso de YACAS sobre el sistema GNU/Linux, pudiendo cambiar algunas de las cosas aqui explicadas en otros sistemas.

Lo aquí expuesto ha sido desarrollado sobre la versión 1.0.51 de YACAS.

1.2. Historia de YACAS

YACAS es todavía un programa muy reciente pero que ofrece unas características y una madurez no presentes en la mayoría del software científico a su misma edad. YACAS ofrece muy buenas perspectivas en un futuro no muy lejano.

YACAS ha sido desarrollado en C++ pensando en la portabilidad, es decir que ha sido desarrollado pensando en ser ejecutado en arquitecturas diferentes.

1.3. ¿Qué es YACAS?

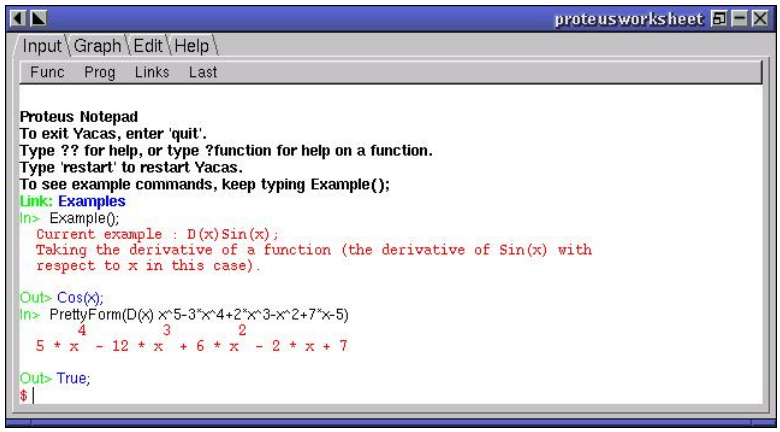
YACAS es un acrónimo de **Y**et **A**nother **C**omputer **A**lgebra **S**ystem.

YACAS no es en sí un software científico, en realidad es un lenguaje, y aunque se distribuye con un conjunto de funciones matemáticas su verdadera potencia reside en su lenguaje, mediante el cual se pueden escribir nuestras propias funciones para la realización tanto de cálculos numéricos como simbólicos.

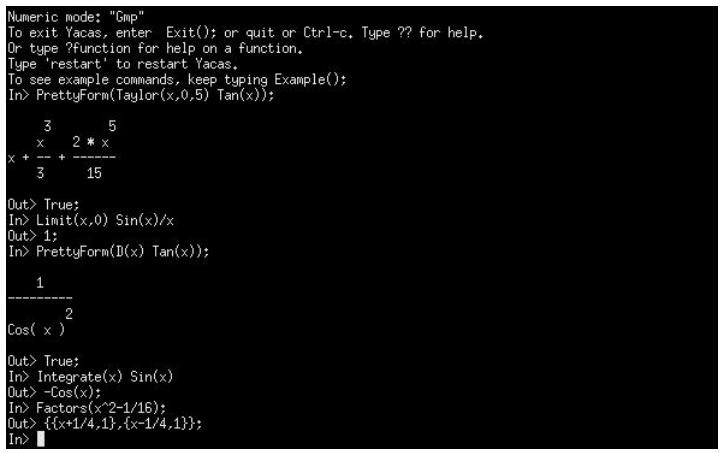
YACAS posee un mecanismo de "**plugins**" que permite la carga dinámica de librerias externas a YACAS. Mediante este mecanismo podemos ampliar las funcionalidades de YACAS.

Podemos utilizar YACAS de varias formas:

- De forma gráfica:



- En modo consola:



- Utilizando scripting.

1.4. ¿Qué podemos hacer con YACAS?

YACAS nos permite resolver problemas matemáticos de una forma, fácil y sencilla. Es bastante intuitivo y su sintaxis es similar a la de otros programas de este tipo.

1. Podemos realizar cálculos numéricos.
2. Podemos realizar cálculos simbólicos.
3. Podemos crear nuestras propias funciones, mediante el lenguaje que incorpora.
4. Podemos utilizar librerias externas en YACAS mediante el uso de su mecanismo de plugins.

5. Podemos interactuar con otros programas, gracias a esto podemos utilizar varios programas para resolver una tarea, utilizando de esta forma los puntos fuertes de cada programa.
6. Podemos utilizarlo en una arquitectura de cliente/servidor y utilizar un ordenador más potente para realizar los cálculos viendolos en el nuestro.

1.5. ¿Como se distribuye YACAS?

YACAS se puede distribuir de para uso comercial y no comercial bajo unas condiciones. El copyright que podemos encontrar en el paquete de YACAS es:

If you wish to use this package and are unsure what the conditions above mean, send me email and we can sort it out.

Ayal Pinkus

apinkus@xs4all.nl

1.6. ¿De donde podemos descargarnoslo?

Si YACAS no viene en nuestra distribución de GNU/Linux podemos descargarlo de http://www.xs4all.nl/~apinkus/yacas.html .

En la página anterior podemos encontrar toda la información necesaria sobre YACAS, tanto sus fuentes, como sus binarios para GNU/Linux o Windows (R) y documentación sobre YACAS.

También podemos descargar YACAS desde sourceforge (http://yacas.sourceforge.net).

1.7. ¿En qué plataformas funciona?

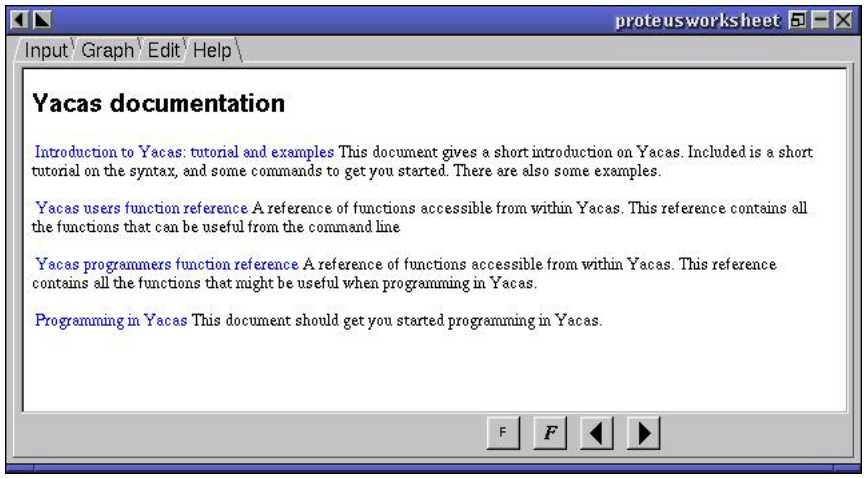
YACAS funciona en todas las plataformas en las que actualmente funciona GNU/Linux (hay soportadas más de 10 plataformas hardware distintas).

También existe versión de YACAS para el sistema operativo Windows (R). Algo curioso es que dicha versión sólo ocupa 400 kb, en un disquete se podría grabar tres veces. Nos podemos llevar YACAS en un disco a cualquier sitio y podremos utilizarlo en cualquier ordenador con Windows (R) unque no tenga instalado YACAS.

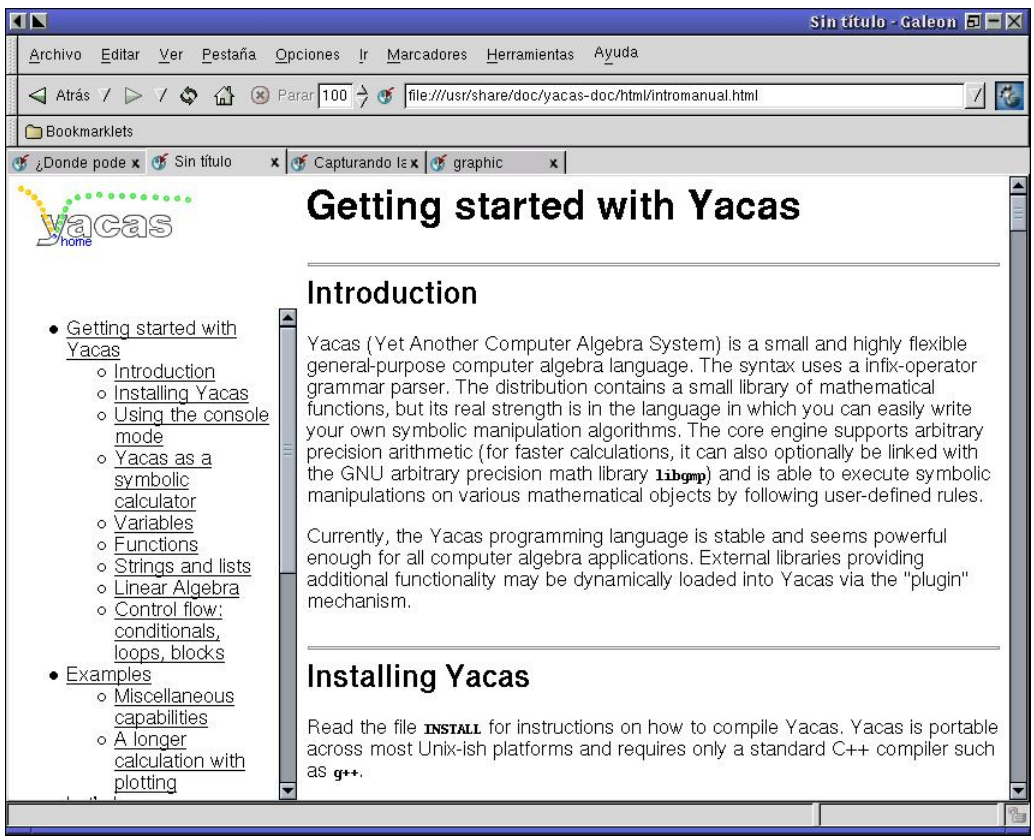
1.8. ¿Donde podemos encontrar ayuda sobre YACAS?

Podemos encontrar ayuda sobre YACAS en http://www.xs4all.nl/~apinkus/yacas.html. En esta dirección se encuentra el manual de YACAS, en Inglés. Lamentablemente todavía no hay versión en Español.

Si estamos utilizando GNU/Linux y tenemos instalada la interface gráfica de YACAS, **yacas-proteus**, podemos recurrir a la ayuda on-line que trae:



Si tenemos instalada la ayuda de YACAS en el sistema o nos la hemos bajado de la URL anterior la podemos consultar utilizando un navegador web:



También existe una lista de correo electrónico a la que nos podemos suscribir, para suscribirnos no tenemos más que mandar un correo a: <yacas-subscribe@yahoogroups.com> y a partir de ese

momento podremos participar en la lista preguntando y respondiendo, en Inglés :-(.

1.8.1. Pidiendo ayuda a YACAS

YACAS posee un mecanismo para suministrarnos ayuda sobre su funcionamiento, pare ello ver Sección 2.5.

1.9. Filosofía en el desarrollo de aplicaciones

A la gente que no está familiarizada con el mundo UNIX muchas veces les extraña que no se desarrolle una aplicación para que se use unicamente en modo gráfico y en cierta forma les molesta el que se pueda utilizar dicha aplicación en modo consola.

Esto se da principalmente en usuarios de Windows (R), llegando a veces a menospreciar una aplicación por el mero hecho de que se pueda ejecutar en modo consola y en modo gráfico.

¿Cual es la razón de que muchas aplicaciones UNIX se puedan ejecutar de ambas formas? Pues es muy simple, el que una aplicación se pueda ejecutar en modo consola tiene las siguientes ventajas:

- Se pueden programar scripts (parecidos a los procesos por lotes en MS-DOS (R)) para la realización de tareas.
- Permite la interacción de esa aplicación con otras aplicaciones mediante el uso de scripts. Por ejemplo se puede tratar una expresión simbólica con YACAS, luego tratarla numericamente con OCTAVE, hacer un estudio estadístico con PSPP y posteriormente crear un gráfico con GNUPLOT.

  Gracias a los scripts, se puede crear un script que haga todo esto sin tener que teclearlo nosotros cada vez que queramos proceder de la misma manera.

- Permite la ejecución remota del programa utilizando conexiones por **telnet** o SSH. Con una aplicación que sólo funcionase en modo gráfico necesitarimos un ancho de banda enorme en nuestra conexión para trabajar de forma fluida.
- El poder funcionar en modo consola permite la creación de multiples interfaces gráficas. El ejemplo más conocido de esto es el compilador de C GNU, **gcc**. En lugar de desarrollar varios compiladores de C, cada uno con un entorno gráfico, se creó un único compilador y luego se crearon diferentes entornos para programar, **Vim**, **Emacs**, **Anjuta**, **KDevelop**, **Xwpe**, **Code Crusader**, **Xcoral**, ...

Un programa que sólo funcione en modo gráfico unicamente posee una ventaja, la comodidad de uso, pero pierde mucha potencia en el sentido de la flexibilidad con la podría interaccionar con otros programas.

Con un programa desarrollado para poder trabajar en consola siempre se podrá crear una interface gráfica que lo llame con las opciones necesarias abstrayendo al usuario de la "complejidad" del uso de la línea de comandos.

1.10. Más software científico en GNU/Linux

En GNU/Linux existe multitud de software científico. No vamos a realizar una descripción detallada de todo el software que podemos encontrar, unicamente comentaremos algunos de los programas que podemos encontrar:

1. YACAS por supuesto.
2. OCTAVE, herramienta que destaca por su potencia en cálculo numérico.
3. GNUPLOT, programa para la creación de gráficas 2D y 3D.
4. GRACE, programa para la representación de datos.
5. SNNS, progama para el tratamiento de redes neuronales.
6. LNKNET, programa para el tratamiento de redes neuronales.
7. LaTeX, entorno para la creación de documentos científicos.
8. TEXMACS, programa para la creación de documentos científicos.
9. LyX, programa para la creación de documentos científicos.
10. XFig, programa para la creación de gráficos vectoriales. Ideal para la inclusión de dichos gráficos en documentos LaTeX y/o manipulación de gráficas para su inclusión en documentos LaTeX.
11. MAXIMA, herramienta matemática.
12. Mathematica (R), el cual no es software libre.
13. MatLab (R), el cual no es software libre.

Capítulo 2. Personalización de YACAS

YACAS nos permite personalizar una serie de parámetros para una mayor comodidad.

2.1. El fichero `.yacasrc`

Cuando arranca YACAS busca el fichero `.yacasrc` en el directorio personal del usuario, si lo encuentra lo ejecuta y en caso de no encontrarlo continua con la ejecución.

Este fichero se escribe según la sintaxis de YACAS como era de suponer, y se pueden configurar varios parámetros:

- DefaultDirectory
- Help
- HistorySize
- PrettyPrinter

2.2. Los ficheros `.def`

Cuando YACAS arranca ejecuta el script `yacasinit.ys` que está en el directorio de scripts, normalmente en `/usr/share/yacas/`.

YACAS no carga en memoria todas las funciones de biblioteca que posee, ya que se podría saturar mucho la memoria del sistema.

Mediante los ficheros `.def` podemos decirle a YACAS en qué ficheros se encuentran el resto de las funciones.

Supongamos que creamos nuestras propias funciones para el tratamiento de estadística y las almacenamos en el fichero `estadistica.ys`. Este fichero lo almacenamos en nuestro directorio de scripts, el cual tiene que conocer YACAS, DefaulDirectory. Entonces para hacerle saber a YACAS en qué fichero están las funciones estadísticas que hemos creado escribiremos un fichero llamado `estadistica.ys.def` en el cual estarán los nombres de las funciones estadísticas que hemos creado, una en cada línea, y terminará con el carácter }:

```
Media
Varianza
Covarianza
...
}
```

Además también tendremos que añadir una línea como la siguiente a nuestro fichero `.yacasrc`:

```
CntDefLoad("yacascripts/estadistica.ys");
```

2.3. El fichero `.yacas_history`

Todas las ordenes que le demos a YACAS en modo interactivo serán almacenadas en el fichero `.yacas_history` en el directorio personal del usuario.

Esto nos permite el utilizar las teclas de los cursores (arriba y abajo) para recuperar y modificar ordenes previamente ejecutadas no siendo necesario reescribirlas de nuevo.

El tamaño de este archivo es por defecto de 50, es decir que YACAS unicamente almacenará las 50 últimas ordenes. Podemos cambiar esto mediante la directiva **HistorySize** (Sección 2.6).

2.4. La directiva DefaultDirectory

Cuando YACAS empieza su ejecución carga una serie de scripts de un directorio especial, en GNU/Linux normalmente es `/usr/share/yacas/`.

Podemos crear nuestros propios scripts definiendo nuestras funciones, constantes, ... y podemos especificar los directorios desde los que serán cargados nada mas solicitarlo el usuario. Para ello debemos especificar esos directorios con la directiva **DefaultDirectory** en el fichero `.yacasrc` que residirá en nuestro directorio personal o bien directamente en YACAS:

Ejemplo 2-1. Añadiendo nuevos directorios al PATH

```
In> DefaultDirectory("/home/jose/yacasscripts/");
Out> True;
In>
```

Ejemplo 2-2. Cargando nuestros propios scripts

```
In> Load("miscript.ys");
Out> True;
In>
```

YACAS buscaría `miscript.ys` en los directorios en los que estan los scripts de inicio y en los que le hayamos dicho mediante la directiva **DefaultDirectory**. Es muy importante el que todos los directorios que especifiquemos con esta directiva acaben con el carácter "/" en sistemas UNIX, ya que YACAS

antepondrá los directorios especificados al nombre del script para cargarlo. En sistemas Windows (R) habrá que utilizar el carácter "\".

2.5. La directiva Help

Podemos solicitar ayuda en YACAS de dos formas diferentes:

1. Podemos solicitar ayuda sobre un comando en particular:

   ```
   In> ?Solve
   ```

2. Podemos solicitar la ayuda general:

   ```
   In> ??
   ```

Al solicitar la ayuda YACAS la mostrará y dependerá la forma en la que lo estemos utilizando para mostrarla:

- Si estamos utilizando la interface gráfica proteusworksheet la ayuda se mostrará en la pestaña Help y se podrá navegar por ella.
- Si estamos utilizando YACAS en modo consola la ayuda se mostrará utilizando lynx que es un navegador web que funciona en modo texto sin mostrar gráficos y podremos navegar por ella.

La ayuda por defecto se instalará en el directorio `documentation` dentro del directorio en el que se instalan los scripts que utiliza YACAS, es decir que la ayuda normalmente estará en `/usr/share/yacas/documentation/`.

Por defecto se utiliza lynx como navegador para visualizar la ayuda, pero es posible configurar YACAS para utilizar otro navegador para mostrar la ayuda:

Ejemplo 2-3. Cambiando la visualización de la ayuda

```
In> Help(_f) <-- SystemCall("galeon " :FindFile("documentation/ref.html"):"#":f);
Out> True;
In> Help() := SystemCall("mozilla ":FindFile("documentation/books.html"));
Out> True;
In>
```

Con esto estamos especificando que vamos a utilizar el navegador galeon para visualizar la ayuda de comandos, **`?Comando`**, mientras que para visualizar la ayuda en general utilizaremos mozilla, **`??`**.

2.6. La directiva HistorySize

Esta directiva sirve para especificar la cantidad de ordenes que YACAS almacenará en el fichero `.yacas_history` (Sección 2.3).

Ejemplo 2-4. Cambiando el tamaño del historial de comandos

```
In> HistorySize(100);
Out> True;
In>
```

De esta forma hemos cambiado el tamaño del historial de comandos a las últimas 100 ordenes recibidas por YACAS, es decir almacenará las 100 últimas ordenes y de esta forma podremos reutilizarlas y/o modificarlas.

2.7. La directiva PrettyPrinter

Esta directiva nos permite definir como se van a mostrar por defecto los resultados de nuestras operaciones.

La mejor forma de verlo es con un ejemplo:

Ejemplo 2-5. Cambiando la presentación de resultados

```
In> Taylor(x,0,3)Sin(x);
Out> x-x^3/6;
In> PrettyPrinter("PrettyForm");

True;

In> Taylor(x,0,3)Sin(x);

     3
    x
x - --
    6

In>
```

En este ejemplo hemos cambiado la presentación normal de resultados por la presentación que ofrece el comando **`PrettyForm`**.

2.8. Personalizando YACAS

Ya hemos visto como modificar el comportamiento de YACAS. Los cambios hechos como hemos visto hasta ahora se pierden al salir de YACAS y sería muy molesto el tener que andar tecleando todas las ordenes de personalización cada vez que entremos en YACAS.

Por ese motivo podemo utilizar el fichero `.yacasrc` (yacasrc) para personalizar el entorno y de esta manera cada vez que entremos en YACAS tendrán efecto las ordenes dadas en dicho fichero.

Ejemplo 2-6. Ejemplo del fichero `.yacasrc`

```
/* AÑADIMOS MI DIRECTORIO PERSONAL DE SCRIPTS AL PATH */
DefaultDirectory("/home/jose/yacascripts/");
/* CARGAMOS VARIOS SCRIPTS CON FUNCIONES PROPIAS */
Load("complejo.ys");
Load("combinatoria.ys");
// MODIFICAMOS EL TAMAÑO DEL HISTORIAL DE COMANDOS
HistorySize(100);
```

2.9. Estableciendo la precisión de los cálculos

Podemos establecer la precisión con la que se mostrarán los cálculos:

```
In> GetPrecision();
Out> 10;
In> N(Sqrt(2));
Out> 1.4142135623;
In> Precision(20);
Out> True;
In> N(Sqrt(2));
Out> 1.4142135623730950488;
In>
```

Podemos utilizar **Precision** en el fichero `.yacasrc` para establecer la precisión por defecto con la que se mostrarán nuestros cálculos.

> **Nota:** La precisión con la que se muestran los datos no afectará a la precisión con la que se realizan las operaciones.

2.10. Ejercicios

1. Escribir un fichero `.yacasrc` para que al arrancar:
   - La precisión con la que se muestren los cálculos sea 15.
   - YACAS busque scripts en el directorio `/home/curso/scripts`.
   - YACAS almacene en el historial de comandos 200 instrucciones.
   - YACAS utilice siempre **PrettyForm** para mostrar los resultados.
   - Utiliza comentarios.

Capítulo 3. Primeros pasos

3.1. Arrancando YACAS

Podemos arrancar YACAS en modo gráfico o en la consola.

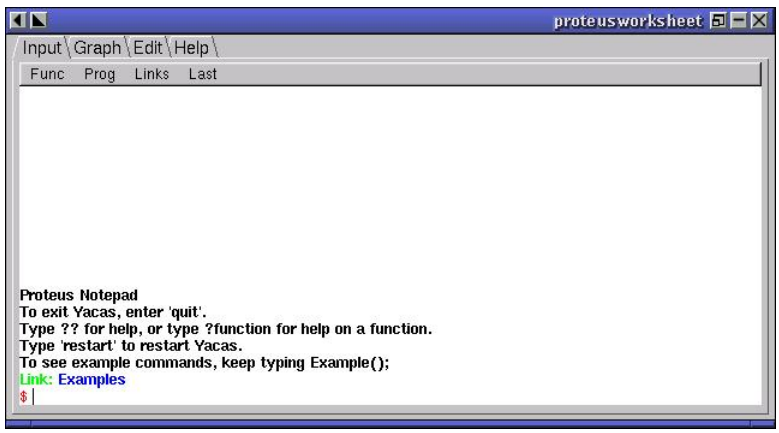
3.1.1. Arrancando YACAS en modo gráfico

Si existe una entrada en los menús de nuestro entorno gráfico bastará con arrancarlo como cualquier otra aplicación.

En caso de no existir podemos abrir una consola y teclear en ella:

```
jose@debian:~/$ proteusworksheet
```

Después de pulsar return arrancaría la aplicación:

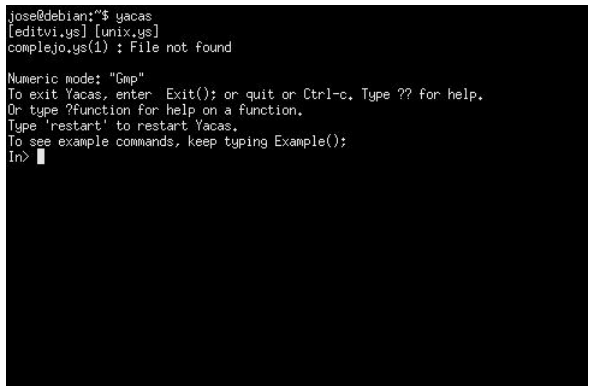


3.1.2. Arrancando YACAS en la consola

Abrimos una consola y tecleamos en ella:

```
jose@debian:~/$ yacas
```

Después de pulsar return arrancaría la aplicación:



3.2. Moviendonos por el historial de comandos

YACAS recuerda las ordenes que le vamos dando, pero sólo las últimas. Por defecto sólo recuerda las 50 últimas, pero esto puede cambiarse con HistorySize (historysize).

Dar una orden a YACAS equivale a escribir la orden y pulsar return. Podemos probar a darle varias ordenes cualquiera, es decir escribir varias frases y pulsar return después de cada una. Si no es una orden válida dará un error, no importa.

Después de haber tecleado varias si pulsamos los cursores, teclas de arriba y abajo, nos moveremos por el historial de comandos, pudiendo repetir o modificar una orden previamente dada.

3.2.1. Autocompletación (sólo en consola)

La shell de GNU/Linux posee una característica especial llamada **autocompletación**. Gracias a esta característica no es necesario teclear todos los comandos, basta con teclear unas pocas letras del principio y pulsar la tecla **TAB**.

Si en YACAS empezamos a escribir un comando que previamente ya hemos utilizado, y esta almacenado en el historial de comandos, y pulsamos **TAB** YACAS nos lo completará automaticamente.

3.2.2. Moviendonos en la línea de comandos (sólo en consola)

Existen una serie de combinaciones especiales de teclas que nos permiten movernos en la línea de comandos con mayor facilidad:

- **Ctrl-a**, vamos al inicio de la línea de comando.

- **Ctrl-e**, vamos al final de la línea de comando.

3.3. Terminando la sesión

Para terminar la sesión en YACAS basta con pulsar Ctrl-c o bien darle a YACAS la orden **quit**:

Ejemplo 3-1. Termiando la sesión

```
In> quit
Quitting...
jose@debian:~$
```

Capítulo 4. Tipos de datos en YACAS

YACAS trata todo como objetos y tiene dos tipos básicos de objetos:

- **Atoms** (átomos). Son datos indivisibles como por ejemplo números y cadenas de texto.
- **Compounds** (compuestos). Son uniones de átomos, es decir datos que se pueden separar en átomos como por ejemplo funciones, expresiones y listas.

4.1. Tratamiento de los datos en YACAS

Internamente YACAS almacena los átomos como cadenas de texto y los compounds como listas.

Podemos utilizar la función **Type** para ver que tipo de objeto estamos tratando:

Ejemplo 4-1. Uso de Type

```
In> Type("Hola");
Out> "";
In> Type(2+(3*5));
Out> "";
In> Type(a*(b+c));
Out> "*";
In> Sin();
Out> Sin;
In> Type({1,2,3});
Out> "List";
In>
```

La interpretación de los ejemplos anteriores es:

1. "**Hola**" es una cadena de caracteres, luego un átomo y en estos casos se devuelve la cadena vacía "".
2. **2+(3*5)** es una expresión y por lo tanto un compound. Sin embargo el valor devuelto es la cadena vacía "", es un átomo!!.

   La explicación es sencilla, YACAS antes de ejecutar la función **Type** evalua su argumento luego en realidad lo que ejecuta YACAS es:

   ```
   In> Type(30);
   Out> "";
   In>
   ```

   y como **30** es un átomo se obtiene "".

3. **a*(b+c)** es una expresión y por lo tanto un compound. YACAS devuelve *****. En los compounds se devuelve el operador de más alto nivel.

   No confundir con el operador que primero aparece:

   ```
   In> Type((a*b)+(c/d));
   Out> "+";
   In>
   ```

4. **Sin()** es una función y por lo tanto un compound. YACAS devuelve el nombre de la función, operador de más alto nivel.
5. **{1,2,3}** es una lista y por lo tanto un compound. YACAS devuelve **List** indicando que se trata de una lista.

4.2. Evaluación de expresiones

YACAS es case-sensitive es decir que distingue entre mayúsculas y minúsculas:

```
In> Sin(Pi);
Out> 0;
In> sin(Pi);
Out> sin(Pi);
In>
```

YACAS utiliza la función **Eval** para evaluar las expresiones. Podemos indicar que no se evalue una expresión mediante el uso de la función **Hold**:

Ejemplo 4-2. Uso de Hold

```
In> Type(2+(3*5));
Out> "";
In> Type(Hold(2+(3*5)));
Out> "+";
In>
```

4.3. Datos escalares

YACAS puede manejar números reales y complejos y operar con ellos:

Ejemplo 4-3. Tipos de datos escalares

```
In> 2+3*I;
Out> Complex(2,3);
In> 3;
Out> 3;
In>
```

4.4. Constantes

Existen en YACAS algunas constantes que ya estan definidas previamente:

- **True**, esta constante representa el valor lógico de cierto.

  ```
  In> 2*3!=5;
  Out> True;
  In>
  ```

- **False**, esta constante representa el valor lógico de falso.

  ```
  In> 2=3;
  Out> False;
  In>
  ```

- **Infinity**, esta constante representa el infinito matemático. La mayoría de las funciones analíticas pueden manejarlo como un número cualquiera.

  ```
  In> Limit(x,0) 1/x;
  Out> Infinity;
  In> Limit(x,0) -1/x;
  Out> -Infinity;
  In>
  ```

- **I** es la unidad imaginaria, la raíz cuadrada de -1. Siempre en mayúscula.

  ```
  In> I^2;
  Out> -1;
  In>
  ```

- **Pi**, representa al número **PI**.

  ```
  In> N(Pi);
  Out> 3.1415926536;
  In>
  ```

- **Undefined**, representa un resultado no definido:

```
In> 0*Infinity;
Out> Undefined;
In>
```

4.5. Cadenas de caracteres (Strings)

YACAS puede manejar cadenas de texto. Una cadena de texto es un secuencia de caracteres entrecomillados:

Ejemplo 4-4. Definiendo una cadena de caracteres

```
In> "Esto es una cadena de caracteres";
Out> "Esto es una cadena de caracteres";
In>
```

4.6. Listas (Lists)

YACAS también puede manejar listas:

Una lista no es más que un grupo ordenado de datos.

Para declarar una lista se colocan los elementos de la lista entre llaves y separados por comas:

Ejemplo 4-5. Definiendo una lista

```
In> {a,b,c,d,e,f};
Out> {a,b,c,d,e,f};
In>
```

4.7. Vectores

Un vector no es más que una lista:

Ejemplo 4-6. Vectores

```
In> {1,3,6};
Out> {1,3,6};
In>
```

4.8. Matrices

Las matrices son listas de listas, es decir una matriz es una lista en la cual cada elemento es otra lista:

Ejemplo 4-7. Matrices

```
In> {{1,4,6},{2,-8,0},{5,7,-2}};
Out> {{1,4,6},{2,-8,0},{5,7,-2}};
In>
```

4.9. Pilas

YACAS puede trabajar con pilas. Internamente interpreta las pilas como listas y tiene una serie de funciones para manejar pilas (Sección 5.7).

4.10. Funciones

Podemos definir funciones de la siguiente forma:

Ejemplo 4-8. Definición de funciones

```
In> f(x):=x^2-2*x+1;
Out> True;
In> f(0);
Out> 1;
In>
```

También es posible definir funciones más complejas:

```
In>  f(x):=[a:=x; a:=a+2;]
Out> True;
In> f(0);
Out> 2;
In>
```

4.11. Polinomios

YACAS entenderá como polinomio cualquier expresión que dependa de una o más variables "libres", es decir que no tengan valores numéricos asociados.

Esto es abusar un poco del lenguaje, ya que no tenemos un tipo de dato especial para los polinomios. Por eso utilizaremos expresiones genéricas para tratar polinomios.

Capítulo 5. Operaciones sobre los tipos de datos

YACAS dispone de una serie de operadores para actuar sobre cada tipo de dato. Pero se pueden realizar operaciones específicas sobre los datos mediante el uso de funciones. Algunas de ellas ya estan implementadas, mientras que nosotros podemos implementar las nuestras propias.

5.1. La función N

La función **N** nos da una aproximación numérica para ciertas expresiones:

```
In> Sin(Pi/3);
Out> Sqrt(3/4);
In> N(Sin(Pi/3));
Out> 0.8660254037;
In>
```

También es posible indicar la precisión con la que queremos que se muestre la aproximación:

```
In> N(Sin(Pi/3),20);
Out> 0.86602540378443864676;
In>
```

5.2. El operador %

Este operador evalua la última orden recibida por YACAS:

```
In> ArcSin(1);
Out> Pi/2;
In> %;
Out> Pi/2;
In>
```

5.3. Operaciones sobre variables

Cuando queramos reutilizar una dato necesitaremos guardarlo en una variable:

```
In> a:=3.141516;
```

```
Out> 3.141516;
In> a;
Out> 3.141516;
In>
```

También podemos declarar variables con la función **Set**:

Ejemplo 5-1. Uso de Set

```
In> Set(a,3*2);
Out> True;
In> a;
Out> 6;
In>
```

Las variables por defecto son declaradas de tipo global, es decir que serán visibles desde cualquier lugar. Ya veremos como declarar variables de tipo local.

En el caso anterior significará que cuando hagamos referencia a la variables "**a**" se sustituirá por su valor:

```
In> a:=3.141516;
Out> 3.141516;
In> a;
Out> 3.141516;
In> Vector:={a,b,c};
Out> {3.141516,b,c};
In>
```

Por lo tanto necesitaremos borrar el contenido de la variable "**a**" antes de utilizarla. Eso lo haremos con la función **Clear**:

Ejemplo 5-2. Uso de Clear

```
In> a:=2;
Out> 2;
In> b:=3;
Out> 3;
In> Clear(a,b);
Out> True;
In> a;
Out> a;
In> b;
Out> b;
In>
```

Disponemos de los operadores ++ y --. Estos operadores provienen del lenguaje C:

- El operador ++ recibe el nombre de operador de autoincremento y sirve para incrementar en una unidad una variable:

```
In> a:=2;
Out> 2;
In> a++;
Out> True;
In> a;
Out> 3;
In>
```

Este operador es equivalente a:

```
In> a:=2;
Out> 2;
In> a:=a+1;
Out> 3;
In>
```

Pero existe una diferencia fundamental y es que el operador autoincremento devuelve **True** en lugar de devolver el resultado.

> **Aviso**
>
> El operador autoincremento siempre va después de la variable, no se puede colocar antes como en C.

- El operador -- recibe el nombre de operador de autodecremento y sirve para incrementar en una unidad una variable:

```
In> a:=2;
Out> 2;
In> a--;
Out> True;
In> a;
Out> 1;
In>
```

Este operador es equivalente a:

```
In> a:=2;
Out> 2;
In> a:=a-1;
Out> 1;
In>
```

Pero existe una diferencia fundamental y es que el operador autodecremento devuelve **True** en lugar de devolver el resultado.

Aviso

El operador autodecremento siempre va después de la variable, no se puede colocar antes como en C.

Por defecto todas las variables se declaran globales, es decir que pueden ser accedidaas desde cualquier parte de YACAS. Para evitar esto podemos declarar variables de forma local con **Local**:

```
In> Local(a,b,c);
Out> True;
In>
```

Esto será útil cuando creemos nuestras propias funciones y evitar de esta manera que una función modifique variables de otra función.

5.4. Operaciones sobre escalares

Veremos algunas de las operaciones que podemos realizar sobre los escalares.

5.4.1. Operaciones usuales sobre escalares

YACAS posee los siguientes operadores para las operaciones usuales sobre escalares:

- + realiza la suma de escalares.
- - realiza la suma de escalares.
- * realiza el producto de escalares.
- / realiza la división de escalares. Por defecto no nos da el resultado de la operación, para ello tendremos que pedirselo mediante el uso de **N**:

  ```
  In> 2/3;
  Out> 2/3;
  In> N(%);
  Out> 0.6666666666;
  In>
  ```

- ^ realiza la exponenciación.

5.4.2. División entera

También disponemos de operadores para realizar la división entera, es decir obtener el cociente y el resto de una división.

Mod sirve para calcular el resto de una división entera:

Ejemplo 5-3. Uso de Mod

```
In> Mod(5,4);
Out> 1;
In>
```

Div sirve para calcular el cociente de una división entera:

Ejemplo 5-4. Uso de Div

```
In> Div(15,4);
Out> 3;
In>
```

5.4.3. Operadores de desplazamiento de bits

Todos aquellos que programan regularmente en leguajes derivados del **C** conocen los operadores de desplazamiento de bits << y >>.

Estos operadores se utilizar para desplazar bits hacía la izquierda, <<, o hacía la derecha, >>.

El uso de estos operadores es importante a la hora de realizar cálculos ya que nos permiten optimizar algunas operaciones.

Los datos en un ordenador se almacenan en base 2, y el desplazar los bits n posiciones hacía la izquierda equivale a multiplicar por 2^n:

Ejemplo 5-5. Uso del operador <<

```
In> 12<<5;
Out> 384;
In> 12*(2^5);
Out> 384;
In>
```

De igual forma el desplazar n bits hacía la derecha equivale a dividir, de forma entera, por 2^n:

Ejemplo 5-6. Uso del operador >>

```
In> 2346 >> 6;
Out> 36;
In> Div(2346,2^6);
Out> 36;
In>
```

5.4.4. Cálculo del Máximo Común Divisor

Para el cálculo del Máximo Común Divisor YACAS dispone de la función **Gcd** y la podemos utilizar de dos formas diferentes:

La primera de ellas es para calcular el Máximo Común Divisor de dos números enteros:

Ejemplo 5-7. Uso de Gcd para el cálculo del mcd de dos números

```
In> Gcd(55,1350);
Out> 5;
In>
```

También la podemos utilizar para calcular el Máximo Común Divisor de una lista de números enteros:

Ejemplo 5-8. Uso de Gcd para el cálculo del mcd de una lista de números

```
In> Gcd({2,5,6,8,9,12,45,67,89});
Out> 1;
In>
```

5.4.5. Cálculo del Mínimo Común Multiplo

Para el cálculo del Mínimo Común Múltiplo YACAS dispone del la función **Lcm**:

Ejemplo 5-9. Uso de Lcm para el cálculo del mcm

```
In> Lcm(6,4);
Out> 12;
In>
```

5.4.6. Trabajando en bases distintas de la decimal

Podemos trabajar en bases distintas de la decimal y para ello tenemos las siguientes funciones:

- **FromBase**, que convierte un número en una base dada a base decimal:

Ejemplo 5-10. Uso de FromBase

```
In> FromBase(16,FF089);
Out> 1044617;
In>
```

- **ToBase**, que convierte un número en base decimal a la base especificada:

  Ejemplo 5-11. Uso de ToBase

  ```
  In> ToBase(16,1044617);
  Out> ff089;
  In>
  ```

5.4.7. Expansiones en base n

Podemos encontrar la expansión de un determinado número en base n:

Ejemplo 5-12. Uso de PAdicExpand

```
In> PAdicExpand(156,10)
Out> 5*10+10^2+6;
In>
```

5.4.8. Aproximaciones racionales de números reales

Podemos aproximar números reales por números racionales mediante el uso de **Rationalize**:

Ejemplo 5-13. Uso de Rationalize

```
In> N(Pi);
Out> 3.1415926536;
In> Rationalize(%);
Out> 3926990817/1250000000;
In> {1.2,56.098,-0.65};
Out> {1.2,56.098,-0.65};
In> Rationalize(%);
Out> {6/5,28049/500,-13/20};
In>
```

5.4.9. Redondeos

Disponemos de varias funciones para redondear:

- **Ceil** redondea al menor entero mayor.

  Ejemplo 5-14. Uso de Ceil

```
In> Ceil(2.8);
Out> 3;
In> Ceil(-2.8);
Out> -2;
In>
```

- **Floor** redondea al mayor entero menor.

  Ejemplo 5-15. Uso de Floor

```
In> Floor(2.8);
Out> 2;
In> Floor(-2.8);
Out> -3;
In>
```

- **Round** redondea al entero más cercano.

  Ejemplo 5-16. Uso de Round

```
In> Round(2.49);
Out> 3;
In> Round(2.51);
Out> 3;
In>
```

5.4.10. Determinación de números primos

Podemos comprobar cuando un número es primo o no utilizando la función **IsPrime**:

Ejemplo 5-17. Uso de IsPrime

```
In> IsPrime(25);
Out> False;
In> IsPrime(7);
Out> True;
In>
```

Esta función comprueba si los números comprendidos entre 2 y la raíz cuadrada del número a comprobar su primalidad lo dividen. No es un algoritmo optimo y consume mucho tiempo de ejecución para números grandes.

5.4.11. Factorización en números primos

Para factorizar un número en factores primos podemos utilizar **Factors**:

Ejemplo 5-18. Uso de Factors

```
In> Factors(50);
Out> {{2,1},{5,2}};
In>
```

También es posible utilizar **Factor**:

Ejemplo 5-19. Uso de Factor

```
In> Factor(50);
Out> 2*5^2;
In>
```

5.5. Operaciones sobre números complejos

YACAS supone que todos los números son reales, pero también puede trabajar con complejos:

5.5.1. Representacion de números complejos

Lo primero que tenemos que tener en cuenta es que la unidad imaginaria es **I**, siempre en mayúsculas.

```
In> 2+3*I;
Out> Complex(2,3);
In>
```

La función **Complex** se utiliza para la representación de números complejos, pero también es posible trabajar con ellos en la forma usual:

```
In> (2+3*I)+(1-5*I);
Out> Complex(3,-2);
In>
```

5.5.2. Determinación de las partes real e imaginaria de un complejo

Para determinar las partes real e imaginaria de un número complejo podemos utilizar **Re** e **Im**:

Ejemplo 5-20. Partes real y compleja de un número complejo

```
In> Re(Complex(2,3));
Out> 2;
In> Im(Complex(2,3));
Out> 3;
In>
```

5.5.3. Determinación del módulo y el argumento de un número complejo

Para la determinación del módulo de un número complejo podemos utilizar **Abs**:

Ejemplo 5-21. Módulo de un número complejo

```
In> Abs(2+3*I);
Out> Sqrt(13);
In>
```

Para la determinación del argumento de un número complejo podemos utilizar **Arg**:

Ejemplo 5-22. Argumento de un número complejo

```
In> Arg(2+3*I);
Out> ArcTan(3/2);
In>
```

5.5.4. Conjugado de un número complejo

Podemos calcular el conjugado de un número complejo con **Conjugate**:

Ejemplo 5-23. Conjugado de un número complejo

```
In> Conjugate(2+3*I);
Out> Complex(2,-3);
In>
```

5.6. Operaciones sobre listas

Veamos algunas de las operaciones que podemos realizar con listas.

5.6.1. Creando listas

Podemos crear listas de varias formas:

1. Declarandolas directamente:

   ```
   In> NuevaLista:={2,5,6};
   Out> {2,5,6};
   In>
   ```

2. Utilizando la función **List**:

   Ejemplo 5-24. Uso de List

   ```
   In> NuevaLista:=List(2,5,6);
   Out> {2,5,6};
   In> OtraLista:=List(NuevaLista,a,b,c);
   Out> {{2,5,6},a,b,c};
   In>
   ```

   Una lista puede ser un elemento de otra lista.

3. Podemos crear una lista concatenando varias con **Concat**:

   Ejemplo 5-25. Uso de Concat

   ```
   In> PrimeraLista:={a,b,c,d};
   Out> {a,b,c,d};
   In> SegundaLista:={1,3,5,7,11};
   Out> {1,3,5,7,11};
   In> ListaFinal:=Concat(PrimeraLista,SegundaLista)
   Out> {a,b,c,d,1,3,5,7,11};
   In>
   ```

Hay que tener en cuenta que de esta forma pueden aparecer elementos duplicados:

```
In> PrimeraLista:={a,b,c,d};
Out> {a,b,c,d};
In> SegundaLista:={d,1,f,a};
Out> {d,1,f,a};
In> Concat(PrimeraLista, SegundaLista);
Out> {a,b,c,d,d,1,f,a};
In>
```

4. Podemos crear una lista como la unión de otras dos con **Union**:

Ejemplo 5-26. Uso de Union

```
In> PrimeraLista:={a,b,c,d};
Out> {a,b,c,d};
In> SegundaLista:={d,1,f,a};
Out> {d,1,f,a};
In> Union(PrimeraLista,SegundaLista);
Out> {a,b,c,d,1,f};
In>
```

Observar que en este caso se eliminan los elementos repetidos, no como con **Concat**.

5. Podemos crear una lista como la intersección de otras dos con **Intersection**:

Ejemplo 5-27. Uso de Intersection

```
In> PrimeraLista:={a,b,c,d};
Out> {a,b,c,d};
In> SegundaLista:={d,1,f,a};
Out> {d,1,f,a};
In> Intersection(PrimeraLista, SegundaLista);
Out> {a,d};
In>
```

6. Podemos crear una lista que sea la diferencia de dos listas con **Difference**:

Ejemplo 5-28. Uso de Difference

```
In> PrimeraLista:={a,b,c,d};
Out> {a,b,c,d};
In> SegundaLista:={d,1,f,a};
Out> {d,1,f,a};
In> Difference(PrimeraLista, SegundaLista);
Out> {b,c};
In>
```

Se mostrarán los elementos que están en **PrimeraLista** y que no están en **SegundaLista**, preservando el orden.

Tenemos que destacar que si hay un cierto elemento que parece **n** veces en la primera lista y **m** veces en la segunda entonces aparecerá **n-m** veces en la diferencia.

7. Podemos crear una lista con todos sus componentes iguales utilizando **FillList**:

Ejemplo 5-29. Uso de FillList

```
In> FillList(Pi,5);
Out> {Pi,Pi,Pi,Pi,Pi};
In>
```

8. Podemos crear listas eliminando elementos de una lista ya creada con **Drop**:

Ejemplo 5-30. Uso de Drop

```
In> Lista:={a,b,c,d,e,f,g,h,i,j,k};
Out> {a,b,c,d,e,f,g,h,i,j,k};
In> Drop(Lista,3);
Out> {d,e,f,g,h,i,j,k}
In> Drop(Lista,-3);
Out> {a,b,c,d,e,f,g,h}
In> Drop(Lista,{4,8});
Out> {a,b,c,i,j,k};
In> Lista;
Out> {a,b,c,d,e,f,g,h,i,j,k};
In>
```

Drop elimina los elementos indicados.

9. Podemos crear listas cogiendo elementos de una lista ya creada con **Take**:

Ejemplo 5-31. Uso de Take

```
In> Lista:={a,b,c,d,e,f,g,h,i,j,k};
Out> {a,b,c,d,e,f,g,h,i,j,k};
In> Take(Lista,3);
Out> {a,b,c}
In> Take(Lista,-3);
Out> {i,j,k}
In> Take(Lista,{4,8});
Out> {d,e,f,g,h};
In> Lista;
Out> {a,b,c,d,e,f,g,h,i,j,k};
In>
```

Take coge los elementos indicados.

10. Podemos crear listas evaluando una cierta expresión desde un punto hasta otro e indicando el paso:

    Ejemplo 5-32. Uso de Table

    ```
    In> Table(i,i,1.,2,.1);
    Out> {1,1.1,1.2,1.3,1.4,1.5,1.6,1.7,1.8,1.9,2};
    In> Table(n!,n,2,5,1);
    Out> {2,6,24,120};
    In>
    ```

11. Podemos crear listas cuyos elementos sean números consecutivos con el operador **..**:

    Ejemplo 5-33. Uso del operador ..

    ```
    In> a:=1 .. 8;
    Out> {1,2,3,4,5,6,7,8};
    In>
    ```

5.6.2. Evaluación de funciones sobre listas

Existen funciones que permiten su evaluación sobre todos los elementos de una lista.

Este tipo de funciones son normalmente funciones numéricas:

```
In> Valores:={Pi,3*Pi/2,0,2*Pi};
Out> {Pi,3*Pi/2,0,2*Pi};
```

```
In> Res:=Sin(Valores);
Out> {0,Sin((3*Pi)/2),0,Sin(2*Pi)};
In> N(Res);
Out> {0,-1.000000,0,0.000000};
In>
```

5.6.3. Operaciones aritméticas sobre listas

Podemos realizar operaciones aritméticas sobre listas:

- Con el operador +/- podemos sumar/restar listas:

  ```
  In> {1,2,3}+{3,2,1};
  Out> {4,4,4};
  In>
  ```

  Las sumas/restas se harán miembro a miembro.

- Con el operador * podemos multiplicar listas miembro a miembro:

  ```
  In> {1,2,3}*{4,5,6};
  Out> {4,10,18};
  In>
  ```

  Con este operador también podemos multiplicar todos los elementos de una lista por un escalar:

  ```
  In> a*{3,6,7};
  Out> {3*a,6*a,7*a};
  In>
  ```

- Con el operador / podemos dividir listas miembro a miembro:

  ```
  In> {2,3,4}/{5,6,7};
  Out> {2/5,1/2,4/7};
  In>
  ```

  Con este operador también podemos dividir todos los elementos de una lista por un escalar:

  ```
  In> {2,3,4}/4;
  Out> {1/2,3/4,1};
  In>
  ```

- Con el operador ^ podemos elevar los elementos de la primera lista a los de la segunda componente a componente:

```
In> {2,3,5}^{0,2,-2};
Out> {1,9,1/25};
In>
```

  Con este operador también podemos elevar todos los elementos de una lista a un escalar:

```
In> {2,3,5}^2;
Out> {4,9,25};
In>
```

5.6.4. Calculando la longitud de una lista

Para calcular la longitud de una lista podemos utilizar la función **Length**:

Ejemplo 5-34. Uso de Length

```
In >a:={a,b,3,5,Sin(x)};
Out> {a,b,3,5,Sin(x)};
In> Length(a);
Out> 5;
In>
```

5.6.5. Recuperando elementos de una lista

Tenemos varias funciones para recuperar elementos de una lista:

- Pare recuperar el primer elemento podemos utilizar **Head**:

  Ejemplo 5-35. Uso de Head

```
In> MiLista:={a,b,c,d};
Out> {a,b,c,d};
In> Primer:=Head(MiLista);
Out> a;
In> MiLista;
Out> {a,b,c,d};
In>
```

  Head no elimina el primer elemento de la lista.

Si **Head** es utilizado sobre una expresión devuelve el primer operando:

```
In> Head(a*b);
Out> a;
In> Head(Hold((2*3)+(4^5)));
Out> 2*3;
In> Head(func(z,y,z));
Out> a;
In>
```

Pero si **Head** es utilizado sobre un átomo devolverá un error:

```
In> Head((2*3)+(4^5));
In function "Head" :
bad argument number 1 (counting from 1)
The offending argument 2*3+4^5 evaluated to 1030
CommandLine(1) : Argument is not a list

In>
```

- Podemos recuperar todos los elementos salvo el primero con **Tail**:

  Ejemplo 5-36. Uso de Tail

  ```
  In> Lista:={2,4,6,8};
  Out> {2,4,6,8};
  In> Tail(Lista);
  Out> {4,6,8};
  In>
  ```

- Para recuperar cualquier otro elemento o grupo de elementos de la lista podemos utilizar **Nth**:

  Ejemplo 5-37. Uso de Nth

  ```
  In> MiLista:={a,b,c,d,e,f};
  Out> {a,b,c,d,e,f};
  In> Nth(MiLista,5);
  Out> e;
  In> SubLista:=Nth(MiLista,{2,4,6});
  Out> {b,d,f};
  In> SubLista;
  Out> {b,d,f};
  In>
  ```

- También podemos recuperar elementos de una lista utilizando **[]**:

```
In> MiLista:={a,b,c,d,e,f};
Out> {a,b,c,d,e,f};
In> MiLista[5];
Out> e;
In> SubLista:=MiLista[{2,4,6}];
Out> {b,d,f};
In> SubLista;
Out> {b,d,f};
In>
```

- Podemos seleccionar dentro de una lista aquellos elementos que cumplan una determinada condición utilizando **Select**:

  Ejemplo 5-38. Uso de Select

```
In> MiLista:={2/3,a,4,{a,6,y},2*3,Hold(a+b=c),{{1,0},{0,1}},Sin(z),"Hola",2.5,2+4*I,Pi,{2,
Out> {2/3,a,4,{a,6,y},6,a+b=c,{{1,0},{0,1}},Sin(z),"Hola",2.5,Complex(2,4),Pi,{2,3,5}};
In> Enteros:=Select("IsInteger",MiLista);
Out> {4,6};
In> Complejos:=Select("IsComplex",MiLista);
Out> {4,6,2.5,Complex(2,4)};
In> Matrices:=Select("IsMatrix",MiLista);
Out> {{{1,0},{0,1}}};
In> Racionales:=Select("IsRational",MiLista);
Out> {2/3};
In>
```

  La función **Select** evalua el primer argumento sobre la lista y devuelve aquellos elementos de la lista que finalizaron con **True**.

5.6.6. Alterando una lista

Podemos eliminar elementos de una lista:

- Utilizando **Delete** se devuelve una lista nueva sin el elemento seleccionado:

  Ejemplo 5-39. Uso de Delete

```
In> Lista:={2,b,4,d};
Out> {2,b,4,d};
In> Delete(Lista,3);
Out> {2,b,d};
In>
```

- Utilizando **DestructiveDelete** eliminamos un elemento de la lista indicada:

  Ejemplo 5-40. Uso de DestructiveDelete

```
In> Lista:={2,b,4,d};
Out> {2,b,4,d};
In> DestrutiveDelete(Lista,3);
Out> {2,b,d};
In> Lista;
Out> {2,b,d};
In>
```

Podemos reemplazar elementos en una lista utilizando **Replace** y **DestructiveReplace**. Su funcionamiento es similar a **Delete** y **DestructiveDelete**, pero reemplazando elementos en lugar de eliminarlos. Estas funciones tienen un tercer elemento que será el elemento que sustituirá al elemento indicado por el segundo argumento en la lista indicada por el primer argumento.

Ejemplo 5-41. Uso de Replace y DestructiveReplace

```
In> MiLista:={a,b,c,d,e};
Out> {a,b,c,d,e};
In> Replace(MiLista,3,C);
Out> {a,b,C,d,e};
In> MiLista;
Out> {a,b,c,d,e};
In> DestructiveReplace(MiLista,3,C);
Out> {a,b,C,d,e};
In> MiLista;
Out> {a,b,C,d,e};
In>
```

También podemos insertar elementos en una lista utilizando **Insert** y **DestructiveInsert**. Su funcionamiento es similar a **Replace** y **DestructiveReplace**, pero insertando el tercer argumento que se le pasa en la posición indicada por el segundo argumento en la lista indicada por el primer argumento.

Ejemplo 5-42. Uso de Insert y DestructiveInsert

```
In> MiLista:={a,b,d,e};
Out> {a,b,d,e};
In> Insert(MiLista,3,C);
Out> {a,b,C,d,e};
In> MiLista;
Out> {a,b,d,e};
In> DestructiveInsert(MiLista,3,C);
Out> {a,b,C,d,e};
In> MiLista;
Out> {a,b,C,d,e};
```

```
In>
```

Podemos añadir elementos a una lista con **Append** y **DestructiveAppend**. Su funcionamiento es similar a los anteriores y tiene dos argumentos, el primero es la lista a la que se le añadirá el elemento que se especifica como segundo argumento. El elemento se añadirá al final de la lista.

Ejemplo 5-43. Uso de Append y DestructiveAppend

```
In> MiLista:={a,b,c,d,e};
Out> {a,b,c,d,e};
In> Append(MiLista,Infinity);
Out> {a,b,c,d,e,Infinity};
In> MiLista;
Out> {a,b,c,d,e};
In> DestructiveAppend(MiLista,Undefined);
Out> {a,b,c,d,e,Undefined};
In> MiLista;
Out> {a,b,c,d,e,Undefined};
In>
```

Podemos eliminar elementos duplicados en una lista utilizando **RemoveDuplicates**:

Ejemplo 5-44. Uso de RemoveDuplicates

```
In> MiLista:={a,b,c,a,d,b,c,e};
Out> {a,b,c,a,d,b,c,e};
In> RemoveDuplicates(MiLista);
Out> {a,b,c};
In> MiLista;
Out> {a,b,c,a,d,b,c,e};
In>
```

También es posible intercambiar dos elementos con la función **Swap**:

Ejemplo 5-45. Uso de Swap

```
In> Lista:={a,b,c,d,e,f};
Out> {a,b,c,d,e,f};
In> Swap(Lista,1,2);
Out> True;
In> Lista;
Out> {b,a,c,d,e,f};
In>
```

5.6.7. Contando ocurrencias

Podemos contar el número de ocurrencias de un elemento en una lista con **Count**:

Ejemplo 5-46. Uso de Count

```
In> Lista:={a,b,a,c,e,a,g,a};
Out> {a,b,a,c,e,a,g,a};
In> Count(Lista,a);
Out> 4;
In>
```

5.6.8. Encontrando un elemento en una lista

Podemos encontrar la posición que ocupa un determinado elemento en una lista con **Find**:

Ejemplo 5-47. Uso de Find

```
In> Lista:={a,b,c,b,e,d};
Out> {a,b,c,b,e,d};
In> Find(Lista,b);
Out> 2;
In> Find(Lista,Pi);
Out> -1;
In>
```

Si el elemento aparece más de una vez devuelve la posición de la primera ocurrencia. En caso de no aparecer en la lista devolverá un -1.

5.6.9. Ordenando listas

Podemos ordenar listas de la siguiente manera:

1. Con la función **BubbleSort**. De esta forma se utiliza el algoritmo de la burbuja:

   Ejemplo 5-48. Uso de BubbleSort

   ```
   In> Lista:={2,6,7,84,5,2,8,-4};
   Out> {2,6,7,84,5,2,8,-4};
   In> BubbleSort(Lista,>);
   Out> {84,8,7,6,5,2,2,-4};
   In> BubbleSort(Lista,<);
   Out> {-4,2,2,5,6,7,8,84};
   In> Lista;
   Out> {2,6,7,84,5,2,8,-4};
   ```

```
In>
```

El tercer argumento es la función de ordenamiento, para valores numéricos podemos utilizar > y < como funciones de ordenamiento.

En el caso de querer ordenar una lista con valores no numéricos deberemos construir una función de ordenamiento que tendrá que verificar:

- Será una función que requiera dos argumentos.
- Unicamente devolverá los valores **True** y **False**.
- Devolverá el valor **True** si el segundo argumento tiene que ir después del primero en la lista ordenada.
- Devolverá el valor **False** en el resto de casos.

El algoritmo de la burbuja es ineficiente para listas grandes. En el caso de tener que reordenar listas grandes se debería utilizar la función del siguiente punto.

2. Con la función **HeapSort**:

Ejemplo 5-49. Uso de HeapSort

```
In> Lista:={2,6,7,84,5,2,8,-4};
Out> {2,6,7,84,5,2,8,-4};
In> HeapSort(Lista,>);
Out> {84,8,7,6,5,2,2,-4};
In> HeapSort(Lista,<);
Out> {-4,2,2,5,6,7,8,84};
In> Lista;
Out> {2,6,7,84,5,2,8,-4};
In>
```

El tercer argumento es la función de ordenamiento, para valores numéricos podemos utilizar > y < como funciones de ordenamiento.

En el caso de querer ordenar una lista con valores no numéricos deberemos construir una función de ordenamiento que tendrá que verificar:

- Será una función que requiera dos argumentos.
- Unicamente devolverá los valores **True** y **False**.
- Devolverá el valor **True** si el segundo argumento tiene que ir después del primero en la lista ordenada.
- Devolverá el valor **False** en el resto de casos.

Este argumento es mucho más eficiente que el de la burbuja para listas grandes.

5.6.10. Particionando una lista

Podemos partir una lista en trozos de igual longitud con **Partition**:

Ejemplo 5-50. Uso de Partition

```
In> Lista:={a,b,c,d,e,f,g};
Out> {a,b,c,d,e,f,g};
In> Partition(Lista,2);
Out> {{a,b},{c,d},{e,f}};
In>
```

Todos los miembros de la nueva lista tendrán la longitud especificada, y los que sobren no se incluirán.

5.6.11. Permutaciones de una lista

Podemos calcular las permutaciones de un conjunto de n elementos con **Permutations**:

Ejemplo 5-51. Permutaciones de un conjunto de n elementos

```
In> Permutations({a1,a2,a3});
Out> {{a1,a2,a3},{a1,a3,a2},{a3,a1,a2},{a2,a1,a3},{a2,a3,a1},{a3,a2,a1}};
In>
```

5.7. Operaciones sobre pilas

YACAS representa, internamente, las pilas como listas y existen una serie de funciones para operar sobre ellas:

- Con **Push** introducimos un elemento en una pila:

  Ejemplo 5-52. Uso de Push

  ```
  In> Pila:={};
  Out> {};
  In> Push(Pila,x1);
  Out> {x1};
  In> Push(Pila,x2);
  Out> {x2,x1};
  ```

```
In> Push(Pila,x3);
Out> {x3,x2,x1};
In> Pila;
Out> {x3,x2,x1};
In>
```

- Utilizando **Pop** podemos eliminar elementos de una pila:

  Ejemplo 5-53. Uso de Pop

  ```
  In> Pila:={};
  Out> {};
  In> Push(Pila,x1);
  Out> {x1};
  In> Push(Pila,x2);
  Out> {x2,x1};
  In> Push(Pila,x3);
  Out> {x3,x2,x1};
  In> Pop(Pila,3);
  Out> x1;
  In> Pila;
  Out> {x3,x2};
  In>
  ```

  Recordar que el último elemento añadido a la pila es accedido con el uno.

- Utilizando **PopFront** podemos eliminar el último elemento añadido a la pila:

  Ejemplo 5-54. Uso de PopFront

  ```
  In> Pila:={};
  Out> {};
  In> Push(Pila,x1);
  Out> {x1};
  In> Push(Pila,x2);
  Out> {x2,x1};
  In> Push(Pila,x3);
  Out> {x3,x2,x1};
  In> PopFront(Pila);
  Out> x3;
  In> Pila;
  Out> {x2,x1};
  In>
  ```

  La función **PopFront** devuelve el último elemento añadido.

- Utilizando **PopBack** podemos eliminar el primer elemento añadido a la lista:

  Ejemplo 5-55. Uso de PopBack

```
In> Pila:={};
Out> {};
In> Push(Pila,x1);
Out> {x1};
In> Push(Pila,x2);
Out> {x2,x1};
In> Push(Pila,x3);
Out> {x3,x2,x1};
In> PopBack(Pila);
Out> x1;
In> Pila;
Out> {x3,x2};
In>
```

5.8. Operaciones sobre vectores

Los vectores son listas y todas las funciones que actúan sobre listas se pueden utilizar sobre vectores, pero existen funciones especificas para vectores.

5.8.1. Producto escalar de dos vectores

Para calcular el producto escalar de dos vectores podemos utilizar **InProduct** y ambos vectores tienen que tener la misma longitud:

Ejemplo 5-56. Producto escalar de dos vectores

```
In> a:={1,2,3};
Out> {1,2,3};
In> b:={4,5,6};
Out> {4,5,6};
In> InProduct(a,b);
Out> 32;
In> a.b;
Out> 32;
In>
```

5.8.2. Producto vectorial de dos vectores

Para calcular el producto vectorial de dos vectores podemos utilizar **CrossProduct** y ambos vectores tienen que tener longitud tres:

Ejemplo 5-57. Producto exterior de dos vectores

```
In> a:={1,2,3};
Out> {1,2,3};
In> b:={4,5,6};
Out> {4,5,6};
In> CrossProduct(a,b);
Out> {-3,6,-3};
In> a X b;
Out> {-3,6,-3};
In>
```

5.8.3. Creación de vectores nulos

Podemos crear un vector de longitud n con todas sus componentes nulas con **ZeroVector**:

Ejemplo 5-58. Uso de ZeroVector

```
In> ZeroVector(3);
Out> {0,0,0};
In>
```

5.8.4. Vectores canónicos

Podemos crear los vectores canónicos con **BaseVector**. Esta función tiene dos argumentos:

- El primer argumento indica el vector. Componente no nula que será igual a uno.
- El segundo argumento indica la longitud del vector. Dimensión del espacio vectorial al que pertenece.

Ejemplo 5-59. Uso de BaseVector

```
In> BaseVector(1,3);
Out> {1,0,0};
In>
```

5.8.5. Normalización de vectores

Normalizar un vector es dejarlo con módulo uno y podemos normalizar vectores con **Normalize**:

Ejemplo 5-60. Normalización de vectores

```
In> Normalize({4,0,-3});
Out> {4/5,0,-3/5};
In> % . %;
Out> 1;
In>
```

5.9. Operaciones sobre Matrices

Las matrices son listas y todas las funciones que actúan sobre listas se pueden utilizar sobre matrices, pero existen funciones especificas para matrices.

5.9.1. Operaciones aritmeticas con matrices

Podemos realizar las operaciones aritmeticas con matrices usuales utilizando los operadores:

- +, para sumar matrices.
- -, para restar matrices.
- *, para multiplicar matrices.
- ^, seguido de un entero realiza la exponenciación de la matriz.

5.9.2. Creación de la matriz identidad

Podemos crear la matriz identidad con **Identity**:

Ejemplo 5-61. Creación de la matriz identidad

```
In> Identity(4);
Out> {{1,0,0,0},{0,1,0,0},{0,0,1,0},{0,0,0,1}};
In>
```

5.9.3. Creación de matrices nulas

Podemos crear matrices nulas con **ZeroMatrix**:

Ejemplo 5-62. Creación de matrices nulas

```
In> ZeroMatrix(2,3);
Out> {{0,0,0},{0,0,0}};
In>
```

5.9.4. Creación de matrices diagonales

Podemos crear matrices diagonales con **DiagonalMatrix**:

Ejemplo 5-63. Creación de matrices diagonales

```
In> DiagonalMatrix({1,2,3,4});
Out> {{1,0,0,0},{0,2,0,0},{0,0,3,0},{0,0,0,4}};
In>
```

5.9.5. Cálculo de la matriz traspuesta

Podemos calcular la matriz traspuesta con **Transpose**:

Ejemplo 5-64. Cálculo de la matriz traspuesta

```
In> Transpose({{1,2,3},{4,5,6},{7,8,9}});
Out> {{1,4,7},{2,5,8},{3,6,9}};
In>
```

5.9.6. Cálculo del determinante de una matriz

Podemos calcular el determinante de una matriz con **Determinant**:

Ejemplo 5-65. Cálculo del determinante de una matriz

```
In> Determinant({{1,2,3},{6,4,0},{0,2,-5}});
Out> 76;
In>
```

5.9.7. Cálculo de la traza de una matriz

Podemos calcular la traza de una matriz con **Trace**:

Ejemplo 5-66. Cálculo de la traza de una matriz

```
In> Trace({{1,2,3},{6,4,0},{0,2,-5}});
Out> 0;
In>
```

5.9.8. Cálculo de la matriz inversa

Podemos calcular la matriz inversa con **Inverse**:

Ejemplo 5-67. Cálculo de la matriz inversa

```
In> Inverse({{0,1},{2,0}});
Out> {{0,1/2},{1,0}};
In>
```

5.9.9. Cálculo del polinomio característico

Podemos calcular el polinomio característico con **CharacteristicEquation**:

Ejemplo 5-68. Cálculo del polinomio característico

```
In> a:={{1,2,3},{4,5,6},{6,7,8}};
Out> {{1,2,3},{4,5,6},{6,7,8}};
In> CharacteristicEquation(a,x);
Out> (1-x)*(5-x)*(8-x)-42*(1-x)+84-8*(8-x)+72-18*(5-x);
In> Expand(%,x);
Out> 14*x^2-x^3+15*x;
In>
```

5.9.10. Cálculo de los valores propios

Podemos calcular los valores propios, raíces del polinomio característico, con **EigenValues**:

Ejemplo 5-69. Cálculo de los valores propios

```
In> a:={{1,2},{2,1}};
Out> {{1,2},{2,1}};
In> EigenValues(a);
Out> {-1,3};
In>
```

5.9.11. Cálculo de los vectores propios

Podemos calcular los vectores propios del polinomio característico, con **EigenVectors**:

Ejemplo 5-70. Cálculo de los vectores propios

```
In> a:={{1,2},{2,1}};
Out> {{1,2},{2,1};
In> vp=EigenValues(a);
Out> {-1,3};
In> EigenVectors(a,vp);
Out> {{-k2,k2},{k2,k2}};
In>
```

5.10. Operaciones sobre polinomios

5.10.1. Simplificación de expresiones

YACAS considerará como polinomios todas aquellas expresiones en las que haya variables libres. Todas estas expresiones las podemos simplificar con **Simplify**:

Ejemplo 5-71. Uso de Simplify

```
In> Simplify(2*x^2-3*x+3*x^2-1);
Out> 5*x^2-3*x-1;
In>
```

5.10.2. Expandir un polinomio

Para expandir un polinomio podemos utilizar **Expand**:

Ejemplo 5-72. Uso de Expand

```
In> Expand((1+x-y)^2,x);
Out> x^2+2*(1-y)*x+(1-y)^2;
In> Expand((1+x-y)^2,y);
Out> y^2+ -2*(x+1)*y+(x+1)^2;
In> Expand((1+x-y)^2,{x,y});
Out> x^2+(-2*y+2)*x+y^2-2*y+1;
In>
```

También podemos expandir un polinomio con **ExpandBrackets**. La diferencia entre **ExpandBrackets** y **Expand** es que el primero intentará eliminar todos los parentesis, no hará falta el indicarle en una lista todas las variables sobre las que tendrá que realizar la expansión:

Ejemplo 5-73. Uso de ExpandBrackets

```
In> ExpandBrackets((a-x)*(b-x));
Out> a*b-x*b+x^2-a*x;;
In>
```

5.10.3. Grado de un polinomio

Podemos calcular el grado de un polinomio con **Degree**. Esta función además nos permite calcular el grado de una expresión como polinomio en una de sus variables:

Ejemplo 5-74. Cálculo del grado de un polinomio

```
In> expr:=x^2-3*x+2;
Out> x^2-3*x+2;
In> Degree(expr);
Out> 2;
In> expr:=a*x+a^2*x^3;
Out> a*x+a^2*x^3;
In> Degree(expr,x);
Out> 3;
In> Degree(expr,a);
Out> 2;
In>
```

5.10.4. División de polinomios

Podemos dividir polinomios con las funciones **Div** y **Mod** (Sección 5.4.2). En lugar de utilizar números enteros como argumentos tendremos que utilizar polinomios.

5.11. Ejercicios

1. Expresar el número 123456789 en:
   - Base dos (binario).
   - Base ocho (octal).
   - Base dieciseis (hexadecimal).

2. Elige varios números y factorizalos.
3. Crea varias listas:
   - Una con los primeros diez números pares.
   - Otra con los primeros diez números impares.
   - Otra con los primeros diez números primos.

   Utilizando las operaciones sobre listas, calcula:
   - Las contatenaciones de las tres listas.
   - Las uniones de las tres listas.
   - Las intersecciones de las listas dos a dos.

4. Crear una lista con valores numéricos y ordenarla creciente y decrecientemente.
5. Calcular el producto vectorial y escalar de dos vectores cualesquiera de dimensión tres. Normalizar el producto vectorial.
6. Calcular el valor de la función Seno en todos los valores que van desde el uno hasta el diez y que distan entre sí 0.1.

   Sugerencia: Utilizar **Table**.

7. En la lista **{a,b,c,d,E,f,g}** reemplaza la letra mayúscula por su equivalente minúscula.
8. Crear una matriz cuadrada de orden 3 en la que cada fila sea uno de los vectores de la base canónica.
9. Dada la lista **{2,a*b,{2,3},3/5,I,Pi,3.7,{{1,2},{3,6}},Pi*I}** determinar que elementos son:
   - Números enteros.
   - Números racionales.
   - Números complejos.
   - Matrices.
   - Atomos.

   Sugerencia: Utilizar **Select**, **IsInteger**, **IsRational**, **IsComplex**, **IsMatrix** y **IsAtom**.

10. Calcular el cociente y el resto al dividir el polinomio **3*x^5-3*x^2+2*x+1** por el polinomio **x+Pi**. Una vez hecho comprobar que se han hecho bien las divisiones.

    Sugerencia: Utilizar variables para almacenar dividendo, divisor, cociente y resto. Para comprobar utilizar que dividendo es igual a divisor por cociente más resto. Trabajar con expresiones simplificadas.

Capítulo 6. Cálculos matemáticos

Ya hemos visto algunas funciones matemáticas que funcionan sobre los tipos de datos que admite YACAS.

6.1. Análisis matemático

Con YACAS podemos resolver problemas de análisis matemático.

6.1.1. Funciones trigonométricas

Disponemos de las siguientes funciones trigonométicas:

Sin, **ArcSin**, **Cos**, **ArcCos**, **Tan**, **ArcTan**.

Estas funciones toman los argumentos en radianes y pueden ser aplicadas a listas.

6.1.2. Logaritmos y la función exponencial

Disponemos de:

Exp y **Ln**.

Estas funciones se pueden aplicar a listas.

Cuando el argumento es un complejo la función **Ln** trabaja con la determinación del logaritmo en el intervalo **(-Pi,Pi]**.

6.1.3. Suma de una lista de valores

Podemos sumar una lista de valores con **Sum**. Esta función podemos utilizarla de dos formas:

1. Para calcular la suma de una serie de valores sin relación alguna:

```
In> Sum({1,3,5,Pi,9});
Out> Pi+18;
In>
```

2. Para calcular la suma de una serie de valores relacionados. Por ejemplo para calcular la suma, finita, de los cuadrados de los diez primeros enteros:

```
In> Sum(n,1,10,n^2);
Out> 385;
In>
```

Los argumentos indican lo siguiente:

- El primero indica la variable.
- El segundo indica desde donde se empezará a sumar.
- El tercero indica hasta que termino se sumará.
- El cuarto indica el término general de la serie.

6.1.4. Producto de una lista de valores

La función **Factorize** funciona igual que **Sum** pero en lugar de sumar multiplica.

6.1.5. Calculando el máximo y el mínimo de una lista

Podemos utilizar **Min** o **Max** para encontrar el mínimo o el máximo de dos o más elementos. Si lo hacemos de más de dos elementos tendremos que utilizar una lista.

```
In> Min(2,-5);
Out> -5;
In> Max({4,7,2,56,8,9,-8,3});
Out> 56;
In>
```

6.1.6. Cálculo de límites

Podemos calcular límites con **Limit** para funciones de una variable. Los podemos calcular también tanto por la izquierda como por la derecha:

Ejemplo 6-1. Cálculo de límites

```
In> Limit (x,0) Sin(x)/Ln(x-1);
Out> 0;
In> Limit (x,0,Left) 1/x;
Out> -Infinity;
In> Limit (x,0,Right) 1/x;
Out> Infinity;
In>
```

6.1.7. Derivación

Podemos calcular derivadas con **D** respecto de una variable (con diferentes ordenes) y sobre varias variables:

Ejemplo 6-2. Derivando funciones

```
In> D(x) 1/x^2;
Out> (-2*x)/x^4;
In> D(x,2) 1/x^2;
Out> (-2*x^4+2*x*4*x^3)/x^8;
In> D({x,y,z}) (x/y)+z;
Out> {y/y^2,(-x)/y^2,1};
In> D({x,y,z},2) (x/y)+z;
Out> {0,(x*2*y)/y^4,0};
In>
```

Hemos hecho lo siguiente:

1. Hemos calculado la derivada respecto de **x**.
2. Hemos calculado la derivada respecto de **x** dos veces.
3. Hemos calculado las derivadas respecto de **x**, **y** y **z**.
4. Hemos calculado las derivadas respecto de **x**, **y** y **z** dos veces.

6.1.8. Desarrollos de Taylor

Podemos hacer desarrollos de Taylor utilizando **Taylor**. Para ello tendremos que indicar la variable, el punto en el que queremos hacerlo y hasta que orden.

Para hacer el desarrollo de Taylor de la funcion seno en el punto cero y hasta orden 11:

Ejemplo 6-3. Desarrollos de Taylor

```
In>Taylor(x,0,11) Sin(x);
Out> x-x^3/6+x^5/120-x^7/5040+x^9/362880-x^11/39916800;
In>
```

También podemos utilizar la función **InverseTaylor**. Esta función se utiliza de igual forma pero calcula el desarrollo de Taylor de la inversa de la función indicada.

6.1.9. Integración

Podemos utilizar**Integrate** para calcular integrales tando indefinidas (aunque todavía no resuelve todo tipo de integrales indefinidas) como definidas:

Ejemplo 6-4. Integración de funciones

```
In> Integrate(x) Sin(Ln(x));
(x*Sin(Ln(x)))/2-(x*Cos(Ln(x)))/2;
In> Integrate(x,-2,0); Sin(x);
Out> Cos(2)-1;
In>
```

Hemos hecho lo siguiente:

1. Hemos calculado la integral indefinida sobre la variable **x**.
2. Hemos calculado la integral definida entre -2 y 0.

6.1.10. Divergencia de un campo vectorial

Podemos calcular la divergencia de un campo vectorial con **Diverge**. Necesitaremos indicarle las variables:

Ejemplo 6-5. Divergencia de un campo vectorial

```
In> Diverge({i*j,j*k,k*i},{i,j,k});
Out> j+k+i;
In>
```

6.1.11. Algunas funciones útiles

- **!**, calcula el factorial del número que le precede. Para que funcione bien el analizador sintático de YACAS no se puede poner ningún operador inmediatamente después de **!**, habrá que dejar un espacio en blanco:

```
In> 3!+1;
CommandLine(1) : Error parsing expression

In> 3! +1;
Out> 7;
In>
```

- **Average**, calcula la media aritmética de una lista de valores.
- **Abs**, calcula el valor absoluto de los números reales y el módulo de los complejos. Puede ser evaluada sobre listas.
- **Bin**, calcula el número combinatorio **n sobre m**:

  Ejemplo 6-6. Uso de Bin

  ```
  In> Bin(8,5);
  Out> 56;
  In>
  ```

- **Sign**, devuelve 1 para números positivos y -1 para los negativos. Se considera el cero como positivo.
- **Sqrt**, para el cálculo de raíces cuadradas y se puede utilizar sobre listas, números enteros positivos y negativos, números reales y complejos.

6.2. Algebra

Con YACAS podemos resolver problemas de algebra.

6.2.1. Obtención de las variables de una ecuación

Podemos obtener las variables que aparecen en una ecuación con **VarList**:

Ejemplo 6-7. Uso de VarList

```
In> VarList(x+a*y-b*c);
Out> {x,a,y,b,c};
In>
```

6.2.2. Resolución de ecuaciones algebraicas

Podemos resolver ecuaciones algebraicas con **Solve**:

Ejemplo 6-8. Resolviendo ecuaciones algebraicas

```
In> Solve(a+b*x==z,x);
(z-a)/b;
In> Solve({a*x+y==c,x+z==d},{x,y});
{{d-z,a*z-a*d+c}};
In>
```

> **Aviso**
>
> Para formar las ecuaciones utilizaremos el operador **==** y no **=**.

6.2.3. Resolución de expresiones

La función **SuchThat** intenta encontrar un valor que anule una expresión para la variable dada:

Ejemplo 6-9. Uso de SuchThat

```
In> SuchThat(2*x+y-z,x);
Out> (z-y)/2;
In>
```

También es posible indicarle que intente calcular el valor que anula a una expresión para una subexpresión:

```
In> SuchThat(Ln(x)*Cos(x)+x-y*Sin(x),Ln(x)*Cos(x));
Out> y*Sin(x)-x;
In>
```

Es necesario que la subexpresión dependa unicamente de una sola variable.

6.3. Cálculo numérico

Con YACAS podemos resolver problemas de cálculo numérico.

6.3.1. Resolución de ecuaciones en una variable (Newton)

Podemos utilizar la función **Newton** para resolver ecuaciones en una variable. Se utilizará el método de Newton:

Ejemplo 6-10. Resolviendo ecuaciones por el método de Newton

```
In> Newton(x^3-x^2+x-3,x,1,0.001);
Out> 1.5747430742;
In>
```

Los argumentos indican lo siguiente:

- La ecuación de la que se quiere encontrar una solución.
- La variable, sólo puede haber una.
- Punto inicial.
- Precisión.

6.3.2. Resolución sistemas de ecuaciones

Podemos utilizar **Solve** (Sección 6.2.2) para resolver sistemas de ecuaciones:

Ejemplo 6-11. Resolviendo ecuaciones algebraicas

```
In> Solve({x+y+z==1,x-y-z==2,x-y+z==3},{x,y,z});
Out> {{3/2,-1,1/2}};
In>
```

> **Aviso**
>
> Para formar las ecuaciones utilizaremos el operador **==** y no **=**.

6.3.3. Cálculo de polinomios interpoladores

Podemos calcular el polinomio interpolador utilizando **LagrangeInterpolant**. Esta función utiliza el método de Lagrange para calcular el polinomio interpolador:

Ejemplo 6-12. Cálculo del polinomio interpolador

```
In> f(x):=LagrangeInterpolant({1,2,3},{5,8,-5},x);
Out> True;
In> f(1);
```

```
Out> 5;
In> Simplify(f(x));
Out> -8*x^2+27*x-14;
In>
```

6.4. Exportación de datos

Podemos utilizar YACAS para realizar cálculos y también es posible exportar los resultados para incorporarlos a documentos **LaTeX** o programas escritos en lenguaje**C**.

6.4.1. Exportando a LaTeX

Podemos exportar nuestros cálculos a **LaTeX** con **TeXForm**. Esta función devuelve una cadena con la expresión **LaTeX**:

Ejemplo 6-13. Exportando a LaTeX

```
In> TeXForm(D(x) Tan(x));
Out> "$\frac{1}{\left( \cos x\right)  ^{2}} $";
In>
```

> **Aviso**
>
> No todas las expresiones se pueden exportar a **LaTeX**.

6.4.2. Exportando a C

Podemos exportar nuestros cálculos a **C** con **CForm**. Esta función devuelve una cadena con la expresión:

Ejemplo 6-14. Exportando a C

```
In> CForm(D(x) Tan(x));
Out> "1. / pow(cos(x), 2.)";
In>
```

6.5. Ejercicios

1. Calcular el límite en cero de **Sin(x)/x**.
2. Calcular la derivadas de **ArcTan(Exp(Sin(Cos(Ln(x*y)))))**:
   - Derivada respecto de **x**.
   - Derivada respecto de **x** y de **y**.
3. Calcular la media aritmética de **1,-2,3,-4,5**.
4. Resolver el sistema de ecuaciones:

   3x-y=-1

   x-y=0

5. Despejar la variable **x** en **a*x-b*c=d**.
6. Encuentra una raíz del polinomio **2*x^3+3*x^2-x+5** con una precisión de **0.001**.
7. Se ha realizado un experimento y se han recogido los siguientes datos:

   Tabla 6-1.

x	**f(x)**
1	2.3
2	-3.6
3	0
4	5.8

   Calcular el polinomio interpolador que aproxima el experimento.

Capítulo 7. Programación

Podemos programar nuestras propias funciones y scripts utilizando las funciones que trae consigo YACAS.

7.1. Interactuando con el usuario

Podemos interactuar con el usuario para la petición de datos o simplemente para mostrar resultados.

7.1.1. Mostrando información

Podemos mostrar información utilizando **Echo**. Existen dos formas de utilizar esta función:

Ejemplo 7-1. Mostrando información

```
In> Echo(N(Sqrt(2)));
1.4142135623
Out> True;
In> Echo({"El logaritmo de 2 es", N(Ln(2))});
El logaritmo de 2 es 0.6931471805
Out> True;
In>
```

1. En la primera forma si es una expresión la evalua e imprime su resultado. Si fuera una cadena de texto simplemente la imprimiría.
2. En la segunda forma si es una lista imprimiría todos y cada uno de los miembros de la lista separados por un espacio en blanco, evaluando todas y cada una de las expresiones que aparecieran en la lista.

Echo siempre devuelve **True**.

7.1.2. Solicitando información al usuario

Podemos pedir información al usuario con **Read**. Esta función lee una expresión desde la entrada estándar, el teclado, y no la evalúa. Para indicar que se termina la expresión se hace terminando la expresión con punto y coma "**;**":

Ejemplo 7-2. Solicitando información al usuario

```
In> a:=Read();
2+3;Out> 2+3;
In> a;
```

```
Out> 2+3;
In> N(a);
Out> 5;
In>
```

7.2. Interactuando con ficheros

Podemos leer y/o grabar datos en ficheros.

7.2.1. Guardando datos en ficheros

Podemos guardar datos en ficheros con **ToFile**:

Ejemplo 7-3. Guardando datos en ficheros

```
In> ToFile("Datos") [PrettyForm(D(x)D(y) Sin(x*y));];
Out> True;
In>
```

De esta forma en el fichero `Datos` se almacenará el resultado obtenido. Hay que utilizar la función **PrettyForm** para lograr que los datos se almacenen en disco. Se perderán todos los datos del fichero `Datos`.

7.2.2. Leyendo desde ficheros

Podemos leer datos desde ficheros con **FromFile**. Supongamos que tenemos el fichero `Datos` conteniendo:

```
Sin(x*y);
```

Ejemplo 7-4. Leyendo datos de un fichero

```
In> FromFile("Datos") D(x)D(y) Read();
Out> Cos(x*y)-x*y*Sin(x*y);
In>
```

Leerá el fichero `Datos` y derivará respecto de **y** y luego respecto de **x**.

7.3. Simplificaciones

7.3.1. Sustitución de expresiones

Es posible realizar sustituciones en nuestros cálculos. Hay veces que necesitamos escribir expresiones muy largas que han de ser tratadas. Para simplificar su escritura las podemos sustuituir por otras y una vez terminados los cálculos volver a ponerlas en su forma original:

Ejemplo 7-5. Sustitución de expresiones

```
In> Simplify((x+3*x^2)^2-(2+x^2-3*x^3);
Out> 9*x^4+9*x^3-2;
In> Subst(x, Cos(a)+Ln(b)) %;
Out> 9*(Cos(a)+Ln(b))^4+9*(Cos(a)+Ln(b))^3-2;
In>
```

7.3.2. Reglas de simplificación

Es posible indicar algunas reglas de simplificación que no se hacen por defecto, para ello disponemos de los operadores **/:** y **/::**.

Ejemplo 7-6. Especificando reglas de simplificación

```
In> Sin(x)*Ln(a*b);
Out> Sin(x)*Ln(a*b);
In> % /: {Ln(_x*_y) <- Ln(x)+Ln(y)};
Out> Sin(x)*(Ln(a)+Ln(b));
In>
```

/: aplica las reglas de simplificación, pero puede ser necesario aplicarlas varias veces en ese caso se utiliza **/::**:

```
In> Sin(x)*Ln(a*b);
Out> Sin(x)*Ln(a*b);
In> % /: {a <- 2, b <- 3};
Out> Sin(x)*(Ln(2*3));
In> Sin(x)*Ln(a*b);
Out> Sin(x)*Ln(a*b);
In> % /:: {a <- 2, b <- 3};
Out> Sin(x)*(Ln(6));
In>
```

Tenemos que tener cuidado a la hora de establecer las reglas ya que si establecemos reglas contradictorias podriamos entrar en un bucle infinito.

7.4. Sentencias de control de flujo

Disponemos de varias sentencias para el control de flujo:

7.4.1. Estructura condicional If

Con este tipo de bucles se rezlizará una acción u otra dependiendo de si se cumple una condición:

Ejemplo 7-7. Estructura condicional If

```
In> sign(x):=If (IsPositiveReal(x), 1, -1);
Out> True;
In> sign(3);
Out> 1;
In> sign(2+3*I);
Out> -1;
In>
```

Con esta estructura se emplean tres argumentos:

1. El primer argumento es la condición y debe devolver **True** o **False**.
2. El segundo argumento es la acción que se realizará cuando la condición dada por el primer argumento sea cierta.
3. El tercer argumento es la acción que se realizará cuando la condición dada por el primer argumento sea falsa. Este argumento es opcional.

7.4.2. El bucle For

Con este tipo de bucle se ejecuta un conjunto de instrucciones mientras una condición es cierta:

Ejemplo 7-8. El bucle For

```
In> For(i:=1,i<=5, i++) Echo({i,i!});
1 2
2 2
3 6
4 24
5 120
Out> True;
In>
```

7.4.3. El bucle ForEach

Con este tipo de bucle podemos realizar una determinada acción para los elementos de una lista:

Ejemplo 7-9. El bucle ForEach

```
In> ForEach(i,{1,2,3,4,5}) Echo(i^2);
1
4
9
16
25
In>
```

Es necesario que especifiquemos la variable que se utilizará dentro del bucle. Esta variable tomará todos y cada uno de los datos de la lista.

7.4.4. El bucle While

Con este tipo de bucle ejecutamos una serie de instrucciones mientras una condición es cierta. Tenemos que tener cuidado de no entrar en un bucle infinito, para ello dentro del conjunto de instrucciones a ejecutar tendremos que incluir "algo" que haga que se alcance el final del bucle:

Ejemplo 7-10. El bucle While

```
In> i:=1;
Out> 1;
In> While(i<5) [ Echo(N(Sqrt(i))); i++;]
1
1.4142135623
1.7320508075
2
Out>
In>
```

7.4.5. El bucle Until

Este bucle funciona igual que el bucle **While** pero se ejecuta hasta que la condición dada se hace verdadera:

Ejemplo 7-11. El bucle Until

```
In> i:=1;
Out> 1;
In> Until(i>4) [ Echo(N(Sqrt(i))); i++;]
1
1.4142135623
1.7320508075
2
Out>
In>
```

7.5. Creación de funciones propias

Veamos ahora como crear nuestras propias funciones con todos los recursos que hemos visto de YACAS:

7.5.1. Comentarios

Siempre que programemos, independientemente del lenguaje, es conveniente la utilización de comentarios. Esto nos facilitará la depuración y/o ampliación del código. Especialmente si hace mucho tiempo que no lo utilizamos.

Los comentarios en YACAS son como en lenguaje C, se considerará comentario todo aquello que esté entre **/*** y ***/**. YACAS también admite comentarios al estilo C++, es decir será comentario todo lo que se encuentre después de **//** y hasta el final de linea.

Ejemplo 7-12. Ejemplos de comentarios

```
/* ESTO ES UN COMENTARIO */
/* ESTO TAMBIEN
   LO ES */
// Y ESTO TAMBIEN, PERO AL ESTILO C++
```

7.5.2. Bloques de código

Podemos crear bloques de código de dos formas:

- Utilizando **[ ]**, cada instrucción deberá ir seguida de un punto y coma.

  ```
  In> [a:=1;b:=2;c:=3;]
  Out> 3;
  In>
  ```

- Utilizando **Prog**, cada instrucción ira seguida por una coma.

```
In> Prog(a:=1,b:=2,c:=3)
Out> 3;
In>
```

7.5.3. Localizando funciones

Hay veces que es necesario conocer en que fichero tenemos almacenada una función, eso se hace con **FindFunction**:

Ejemplo 7-13. Uso de FindFunction

```
In> FindFunction("Sum");
Out> "sums.rep/code.ys";
In>
```

7.5.4. Ejemplo de creación de funciones

Vamos a crear una función que dada una lista pida al usuario un número y calcule la suma de todas las componentes de la lista elevadas al número introducido por el usuario.

La función sería:

```
MiFuncion(list):=[
  /* DECLARAMOS LOCALES LAS FUNCIONES */
  Local(poten, tmplist, tmpRes, i);
  /* PEDIMOS EL DATO */
  Echo("Introduce la potencia: ");
  poten:=Read();
  Echo("");
  /* ELEVAMOS A LA POTENCIA ESPECIFICADA */
  tmplist:=list^poten;
  tmpRes:=0;
  /* SUMAMOS TODOS LOS MIEMBROS DE LA LISTA */
  For(i:=1,i<=Length(list),i++) tmpRes:=tmpRes+tmplist[i];
  /* MOSTRAMOS LOS RESULTADOS */
  Echo({"La suma es", tmpRes});
];
```

Esta función la almacenaremos en un fichero `mifuncion.ys` que estará en nuestro directorio de scripts. Crearemos un fichero `mifuncion.ys.def` para que YACAS sepa donde se encuentra y evitar que este cargado en memoria. El fichero `mifuncion.ys.def`:

```
MiFuncion
}
```

A continuación tendremos que modificar el fichero `.yacasrc` de la siguiente manera:

```
/* DIRECTORIO DE SCRIPTS */
DefaultDirectory("/home/jose/yacascripts/");
/* INDICA LA DEFINICION DE LA FUNCION */
CntDefLoad("yacascripts/funcion.ys");
```

En nuestro directorio de scripts tendremos que tener los ficheros `mifuncion.ys` y `mifuncion.ys.def`.

Al arrancar YACAS ya podremos utilizar la función:

```
In> MiFuncion({1,2,3,4});
Introduce la potencia:
6;
La suma es 4890
Out> True;
In>
```

7.6. Ejercicios

1. Hacer una función que pida un número por teclado y diga si es par o impar.
2. Hacer una función que tenga un único argumento y que imprima su módulo y argumento si es un número complejo y en caso contrario que imprima su valor absoluto.
3. Escribir una función que pida números por pantalla e imprima sus factoriales hasta que se introduzca un cero. Hacerlo con **While** y con **Until**.
4. Hacer una función que admita un único argumento, una lista, y que compruebe cuales de los elementos de la lista son primos y cuales no. Una vez comprobados que imprima todos los primos juntos y los no primos factorizados.
5. Hacer una función que pida por teclado un número e imprima todos los primos menores que él.
6. Hacer una función que admita una lista donde sus elementos sean los coeficientes de un polinomio y pida por teclado un punto para calcular una solución del polinomio y también la precisión de los cálculos.
7. Crear una función definida de la siguiente manera:
   - La función será la identidad para todo x menor o igual que -6.
   - La función será elevar al cuadrado para todo x entre -6 y 6.

- La función será la identidad para todo x mayor o igual que 6.

Apéndice A. GNU Free Documentation License

A.1. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject.

(Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this

License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

A.3. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document’s license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

A.6. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

A.7. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

A.9. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

A.10. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

A.12. ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with...Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.