Anexo C. Manual de uso de CrisolScript y CSCompiler.

Índice.

C.1. Introducción.

Este documento recogerá las especificaciones del lenguaje *CrisolScript* así como el uso del compilador asociado al mismo y al que denominaremos *CSCompiler* (de CrisolScriptCompiler).

El lenguaje *CrisolScript* permitirá que el diseñador pueda establecer la lógica del juego que esté creando pues mediante su uso, se permitirá especificar (programar) lo que ocurra cuando *Crisol* desencadene diferentes eventos. Estos eventos se podrán personalizar para cada entidad que exista en el universo de juego, es decir, para un mismo evento dos entidades diferentes podrán tener asociadas, a su vez, dos scripts también distintos. El objetivo de *CrisolScript* será el de ser un lenguaje, ante todo, sencillo de utilizar.

Mediante el uso de *CSCompiler* se logrará realizar una validación de los archivos con el código fuente escrito. De este modo se podrá detectar posibles errores y pasar de formato texto (el formato en el que el diseñador programará en *CrisolScript*) a formato binario. Esto último será vital para *Crisol*, pues permitirá una ejecución rápida del código.

C.1.1. Un lenguaje interpretado.

Antes de pasar a explicar el lenguaje, se deberá de tener en cuenta que *CrisolScript* será un lenguaje interpretado a nivel de código intermedio, es decir, la ejecución de los scripts se realizará sólo cuando el código fuente haya sido compilado por *CSCompiler*. El proceso para llegar del código fuente originalmente escrito por el diseñador a código intermedio será el siguiente:

1) Se escribirán los ficheros script en modo texto. Estos ficheros formarán el código fuente.

2) Los ficheros con el código fuente se compilarán para producir un archivo que poseerá las instrucciones que había en los ficheros fuentes pero preparadas para que un intérprete embebido en *Crisol* las procese. Este archivo de salida será lo que se denomine código intermedio.

3) El código intermedio producido será procesado y ejecutado por el intérprete embebido en el motor (que también se le conocerá como máquina virtual) siempre y cuando se produzca un evento para el que se hubiera asociado un fichero con código script.

Una opción alternativa al trabajo con código intermedio sería el utilizar un intérprete puro, es decir, un intérprete que trabajara directamente con los archivos de código fuente. El problema es que este intérprete se convierte en algo lento para una aplicación interactiva como es un videojuego y tales lapsus siempre serían perceptibles por el jugador. Principalmente cuando se hallen activos scripts que se ejecuten cada “x” tiempo. El uso de código intermedio no soluciona completamente este problema de pérdida de velocidad, pero sí que lo mejora de forma considerable al evitar, entre otras operaciones, los análisis léxicos y sintácticos.

C.1.2 Notación simple orientada a objetos.

El lenguaje *CrisolScript* trabaja con una sintaxis similar a lo que pudiera encontrarse en un lenguaje orientado a objetos. Si bien, se tratará sólo de un maquillaje para implementar de una forma mucho más clara y sencilla el trabajo con ciertos elementos.

Existirá un “objeto” llamado `Game` que se encargará de simular a *Crisol* como aplicación, otro llamado `World` que representará al universo de juego y por último, unas variables de tipo `entity` que permitirán gestionar a todas las entidades que existan en el universo de juego como si de objetos se trataran.

C.1.3. Sin distinción entre mayúsculas y minúsculas.

CrisolScript no distinguirá entre mayúsculas y minúsculas, por lo tanto para *CSCompiler* será lo mismo leer la palabra reservada `function` que `Function` que `FunCtION`. Esto mismo será aplicable a constantes, variables y nombres de función definidos por el programador.

Esta libertad para poner los identificadores como se quiera será un arma de doble filo. Por un lado se permitirá solventar uno de los errores más típicos en los programadores sin experiencia sin embargo, estos mismos programadores sin experiencia correrá el riesgo de crear código poco legible. Para evitar este problema, se recomendará que cada diseñador use unas normas de escritura o de lo contrario, los códigos que realice serán difíciles de comprender para otra persona. En los ejemplos de este documento, todas las palabras reservadas irán en minúsculas y los nombres de funciones y variables con la primera letra del identificador (incluido los compuestos) en mayúsculas. En el caso de las constantes se utilizarán las mayúsculas.

C.1.4. Modo general de trabajo con scripts.

El trabajo con los scripts será muy sencillo; existirán una serie de eventos en el motor que podrán tener asociados scripts definidos por el diseñador. En tales casos, éste podrá indicar el nombre del archivo script para que cuando se produzca tal evento se ejecute el código localizado en dicho script. Los archivos scripts estarán formados por una función especial (consultar el apéndice destinado a los eventos script para obtener más información) con un formato específico en cuanto a los parámetros que recibirá. Dicho formato se conocerá porque todas estas funciones asociadas a eventos poseerán un tipo (que no será otro que el evento al que pertenecen) que permitirán al compilador averiguar inmediatamente el formato de los parámetros que deberá de poder localizar. Acompañando a dicha función especial se podrán definir otras de caracter local cuya funcionalidad será la de facilitar la organización del código y cuya utilización quedará restringida al cuerpo de la función especial que representa el evento.

El diseñador no tendrá la obligación de asociar archivos con código *CrisolScript* a todos los eventos; cuando uno se desencadene y éste no tenga asociado ningún archivo script, se ejecutará únicamente el comportamiento por defecto asociado a dicho evento. Como se verá más adelante, habrá eventos que no tendrán comportamiento por defecto. Cuando sí exita comportamiento por defecto será muy importante el modo en que se retorne de los scripts pues, dependiendo de este factor, se podrá indicar al motor que realice, o no, la operación por defecto.

Por ejemplo, asociado a la acción de coger un ítem existe un script. Dicho script tiene como comportamiento por defecto el de pasar el ítem al inventario de la criatura que quiere cogerlo. Si en el script se programa un código que compruebe el espacio libre en el inventario que desee coger el ítem, se podrá indicar que no se desea realizar la operación por defecto (coger el ítem) cuando en el inventario ya no quede espacio libre.

C.2. Especificaciones de CrisolScript.

En este apartado se analizarán todas y cada una de las partes que hacen a *CrisolScript* un lenguaje de programación. Será, por tanto, el apartado más importante de esta documentación, pues se comentarán los aspectos concretos del mismo de cara a su utilización.

C.2.1. Palabras clave base.

Las palabras clave base, y por lo tanto, reservadas de *CrisolScript*, serán las siguientes:

`begin`	`end`	`global`	`const`
`var`	`compile`	`func`	`number`
`string`	`entity`	`script`	`function`
`if`	`then`	`else`	`while`
`do`	`return`	`ref`	`import`
`void`			

Palabras clave de CrisolScript.

Estas palabras no podrán usarse por el programador para crear constantes, variables o nombres de funciones pues, como ya se indica, serán reservadas por *CrisolScript*. Sin embargo, sí que podrán aparecer incluidas en otras palabras. Por ejemplo: `ReturnValue`, `MagicNumber`, etc serán construcciones válidas.

Cuando se estudien los diferentes eventos script se especificarán nuevas palabras claves que servirán para indicar el tipo de un script (el evento al que está asociado). También se comentarán las constantes globales internas de *CrisolScript* cuya finalidad será la de permitir al usuario una programación más cómoda.

C.2.2. Operadores lógicos y matemáticos

A continuación se muestran los operadores lógicos y matemáticos en *CrisolScript*. También se incluye el operador de asignación, que permitirá asociar un valor a una variable o constante (en este caso, sólo en su declaración).

Símbolo	Operación
`/`	División
`-`	Resta
`+`	Suma
`%`	Resto
`*`	Multiplicación
`>`	Mayor que (de izq. a dcha.)
`>=`	Mayor o igual que (de izq. a dcha.)
`<`	Menor que (de izq. a dcha.)
`<=`	Menor o igual que (de izq. a dcha.)
`:=`	Asignación (se asigna de dcha. a izq.)
`==`	Comparación “igual que”
`!=`	Comparación “distinto que”
`&&`	Y lógico
`||`	O lógico
`!`	No lógico

Operadores lógicos y matemáticos.

Los operadores lógicos permitirán realizar construcciones en las que se evalúe si una expresión es verdadera o falsa. Por ejemplo, la expresión `(3 > 4)` será falsa ya que 3 no es mayor que 4. *CrisolScript* evaluará las expresiones falsas con un valor menor igual a 1 (`false`) y las expresiones verdaderas con un valor mayor o igual a 1 (`true`).

Los operadores matemáticos permitirán realizar operaciones aritméticas utilizando constantes o variables. Además, el valor que produzcan podrá ser utilizado para construir una operación lógica o para asignárselo a una variable o constante (en su declaración).

C.2.3. Operadores sintácticos

Los operadores sintácticos en *CrisolScript* serán:

Símbolo	Significado
`,`	La coma sirve para separar los parámetros y argumentos de función y las variables cuando éstas se declaran.
`/**/`	Inicio / fin de un comentario multilínea.
`//`	Inicio de una línea de comentario de una sola línea.
`;`	Fin de una expresión.
`(`	Inicio de una subexpresión o de una lista de parámetros.
`)`	Fin de una subexpresión o de una lista de parámetros
`.`	Acceso a un método de un objeto.

Operadores sintáticos.

Los operadores sintácticos tendrán una enorme importancia a la hora de elaborar los scripts. Mención a parte merecerán los comentarios cuya utilidad será más que evidente para que el programador pueda asociar texto de ayuda en los mismos archivos de código fuente. Se recomienda encarecidamente que se haga uso de los comentarios para facilitar la compresión de lo que se esté creando.

Tal y como se indica en la tabla, los comentarios en *CrisolScript* podrán ser de dos tipos. O bien en una sola línea:

```
// Comentario
```

O bien en varias líneas:

```
/*
Comentario
Comentario
....
Comentario
*/
```

En este último caso, será comentario todo aquello que se encuentre entre `/*` y `*/`.

Otros de los aspectos vitales a comentar de la tabla, será el apartado referido al carácter '`;`' para indicar fin de expresión. En *CrisolScript*, todas las expresiones deberán de acabar con ese carácter.

C.2.4. Extensión de los archivos.

Los archivos donde se programen los scripts, funciones, etc, serán siempre de texto. Al ser de texto, podrán editarse con cualquier editor normal y corriente (wordpad, por ejemplo). La extensión de los archivos será "`.cst`" (que vendrá de *C*risol*S*cript *T*ext). El archivo de salida que producirá *CSCompiler* se denominará "`CrisolGameScripts.csb`" (cuya extensión vendrá de *C*risol*S*cript *B*inary). Este archivo contendrá las instrucciones de código intermedio que representen a los scripts.

C.2.5. Formato de un script.

El formato general de un script asociable a un evento será el siguiente:

```
script TipoScript(Parámetros)
// Funciones importadas al script
```

```
import “Archivo_1”;
// ...
import “Archivo_n”;

const
// Constantes locales al script
var
// Variables locales al script

func
// Funciones locales al script
tipo function NombreFuncionLocalAlScript(Parámetros)
const
// Constates locales a la función
var
// Variables locales a la función
begin
// Cuerpo de la función local al script
end

begin
 // Cuerpo del script
 return Valor Numérico;
end
```

Formato general de un script asociable a un evento.

A continuación se analizará cada uno de las partes del archivo script. Baste decir, a modo de introducción, que el cuerpo del script será el punto donde se comenzará a ejecutar el código y que siempre se esperará que se retorne un valor numérico. Si dicho valor numérico es mayor o igual a 1 se entenderá que se querrá ejecutar la acción por defecto asociada al evento y si el valor es menor que 1, lo que se entenderá es que no se desea ejecutar dicha acción por defecto. En caso de no retornar ningún valor, se evaluará como si se hubiera retornado un valor de 1.

C.2.5.1. La función asociada al evento script.

Todo archivo script estará dominado por una función especial que no será más que la que represente al evento al que estará asociado el archivo script. Dicha función estará indicada con la palabra clave `script` seguida del tipo de script del que se trata y, encerrados entre paréntesis, los posibles parámetros que podrá recibir. Siempre que se hable a partir de ahora de script, haremos referencia, a no ser que por el contexto se hable de un archivo general, a la función especial que define un evento script. En nuestro ejemplo anterior sería:

```
script TipoScript(Parámetros)
```

Los tipos de script sirven para que el compilador sepa cuáles son los parámetros que debe de exigir, además de permitir que cuando desde *Crisol* se invoque un script, se pueda comprobar si el script asociado al evento que ha ocurrido es válido o no (es decir, que se pueda asegurar que al evento de observar no se le asocie, por ejemplo, una funcion script para el evento de coger ítem). Los tipos de scripts, así como su significado y parámetros, están explicados en el apéndice correspondiente.

La función que representa un evento siempre deberá de retornar un valor numérico (que, tal y como se explico anteriormente, indicará si se desea ejecutar la acción por defecto asociada al evento o no). Para retornar un valor numérico se utilizará la palabra reservada `return` seguida del valor numérico, esto es:

```
return ValorNumerico;
```

Si la función termina sin retornar un valor numérico *CrisolScript* usará:

```
return 1;
```

como valor por defecto.

La función script será el único elemento necesario de un archivo script, pues ésta no tendrá porque tener constantes, variables, funciones locales o código para ser válida (no haría nada, pero sería válida). Por ejemplo, un archivo script asociado a un determinado evento (sin especificar) sería correcto si se escribiera de la forma siguiente:

```
script TipoScript(Parámetros)
begin
 // Cuerpo del script, aquí es donde comenzará a ejecutarse el script
 return 1;
end
```

Sin embargo, un archivo script que tuviera sólo este contenido ya sería incorrecto:

```
void function NombreFuncion(Parámetros)
begin
// Cuerpo de una función sin constantes y sin variables
end
```

Y sería incorrecto porque faltaría la función de evento script a la que la anterior debería de pertenecer (ser local).

Podrán existir cero o más funciones asociadas a un mismo evento, la diferencia se hallará en el nombre del archivo deberá de ser distinto. Por ejemplo, si tenemos una espada y una lanza y queremos que cuando se produzca el evento de observar espada y lanza se ofrezca una descripción detallada de cada una de las armas, tendremos dos opciones. La mejor de todas será definir un script asociado al evento de observar y asociárselo a la espada y definir otro diferente y asociárselo a la lanza. La segunda opción consistiría en definir un único archivo y asociárselo tanto a la espada como a la lanza pero teniendo en cuenta que en el interior del mismo, el código debería de comprobar el tipo de arma y realizar la descripción según la misma. Este método será de ejecución más lenta (al tenerse que realizar comprobaciones) y, de cara al futuro, más complicado de mantener (pues a medida que se incrementara el número de armas, más comprobaciones se deberían de realizar, empeorando con ello la calidad del código fuente).

C.2.5.1.1. Tipos de scripts

Gracias a los tipos de scripts, sabremos a qué eventos podremos asociar código. En el *anexo D*, se mencionarán todos los tipos existentes, los parámetros que poseerán y la descripción general. Los tipos asociados a los scripts serán fácilmente distinguibles porque todos comenzarán por “On”.

Recordemos que todo script deberá de indicar si desea que se ejecute la acción por defecto asociada al mismo o si por el contrario, desea cancelarla. Aunque esta sea la norma, habrá excepciones cuando el script no tenga asociada ninguna acción por defecto (sean scripts meramente informativos como el que se lanza cuando se ha pulsado en el panel horario) dando igual la forma en la que se retorne. Normalmente, lo ideal será no retornar nada cuando se desee que se ejecute la acción por defecto ya que cuando *CSCompiler* detecte que no hay retorno en un script, se encargará de asociar por si sólo la acción por defecto (internamente introducirá una línea de código `return 1;`)

Todos los eventos serán relativos siempre a los tres “objetos” principales que existirán en *CrisolScript* y que fueron comentados al comienzo de este documento, a saber: existirán eventos asociados al objeto `World`, eventos asociados al objeto `Game` y, por último, eventos asociados a las entidades, objetos `entity`. La única excepción serán los dos eventos asociados a celdas de juego. Comprender esto facilitará en gran medida el desenvolverse con los métodos.

C.2.5.2. Las funciones.

Las funciones servirán para agrupar código que pueda ser utilizado varias veces, esto es, evitan el tener que escribir un código que se ha de ejecutar en varias ocasiones una y otra vez. El uso de funciones será imprescindibles en el caso de que hagamos scripts medianamente complejos y queramos programar de la forma más optimizada y legible posible.

Las funciones podrán existir de dos formas. Por un lado se podrán tener funciones locales a un script y por otro funciones globales (estas últimas se verán en profundidad más adelante). En ambos casos, su existencia será opcional, esto es, se podrán escribir scripts sin funciones locales y/o globales. Las funciones locales a un script siempre se escribirán de forma secuencial y antes del cuerpo de éste, es decir, antes del `begin ... end` del script. Las dos únicas diferencias entre las funciones locales y globales radicará en que estas últimas se escribirán en archivos a parte y podrán ser accesibles desde cualquier script (las funciones locales, como su propio nombre indica, serán locales al script en donde se definan).

Una función podrá invocarse desde el cuerpo de un script o desde el cuerpo de otra función. Para que la llamada a función sea correcta, las funciones deberán de recibir los parámetros adecuados (si es que poseen argumentos). Por otro lado, las funciones globales podrán llamar sólo a aquellas funciones que estén definidas en su mismo archivo (y que serán, por tanto, también funciones globales) y no podrán utilizar las constantes o variables definidas en el script que las use (a diferencia de lo que ocurre con las funciones locales). Por último, las funciones locales podrán llamar a todas aquellas funciones locales que se definan en el mismo script, así como a aquellas funciones globales que sean importadas.

Toda función tendrá un tipo mediante el cuál deberá, o no, retornar un valor. Cuando una función no retorna ningún valor decimos que su tipo es `void` y cuando retorne un valor su tipo podrá ser `entity`, `string` o `number`. Siempre que se lea `return` se abandonará el cuerpo de una función. En caso de que la función sea de tipo `void`, el `return` deberá de ir seguido de un punto y coma (“`;`”) y en el caso de que la función sea de tipo `entity`, `string` o `number`, el `return` deberá de ir seguido de un valor acorde al tipo que devuelven (ya sea una constante o una variable de ese tipo o, incluso, una llamada a una función que devuelva el mismo tipo que la función original), seguido de un punto y coma (“`;`”).

Nunca dos funciones locales podrán existir con el mismo nombre. Esto mismo se extenderá a aquellos archivos con funciones globales que se incluyan con la palabra reservada `import` (la cual se estudiará más adelante). Tampoco podrán ser iguales a función de API alguna (se verán más adelante también) sin embargo, funciones locales a scripts distintos sí que podrán tener el mismo nombre ya que en tales casos no podrán existir conflictos, al ser compilaciones completamente independientes.

C.2.5.3. Las variables.

Las variables en *CrisolScript* se declararán siempre después de la palabra clave `var`. No será obligatorio que un script o una función tenga variables. Los nombres de las variables sólo podrán crearse usando caracteres alfabéticos y/o el signo subrayado bajo `‘_’`. No será aceptado el carácter `‘ñ’` ni los acentos.

CrisolScript soportará tres tipos de variables:

- **Los números**. Los números en *CrisolScript* serán reales, con lo que su gestión será muy cómoda y fácil para el programador. Para designar una variable de tipo número se usará la palabra clave `number`.

- **Las cadenas de caracteres**. Las cadenas de caracteres representarán porciones de texto y podrán tener una longitud variable. Se declararán utilizando la palabra clave `string`. Todas las cadenas de caracteres deberán de ir siempre entre comillas dobles (`“cadena”`), estando permitida la cadena vacía o de longitud cero (“”).

- **Entidades**. Para poder trabajar con las diversas entidades que se han creado en un área o con el jugador mismamente, tendremos unas variables especiales que actuarán a modo de objetos. Estas variables se declararán utilizando la palabra clave `entity`.

A la hora de declarar variables podremos hacerlo separando dichas variables por comas, logrando declarar varias variables de un mismo tipo a la vez y/o usando el operador de asignación `':='`, con lo que ademas las inicializaremos a la vez que las declaramos. En caso de no inicializar una variable al ser declarada, *CrisolScript* la inicializará a cero (0) si es de tipo `number` o `entity` o a cadena vacia (“”) si es de tipo `string`.

Según lo dicho, son declaraciónes válidas de variables:

```
var
 number x, pi := 3.141516, y, z := -23;
 string Nombre := “Bilbo”, Apellido := “Bolson”;
 entity Arbol;
```

Las operaciones matemáticas de división, suma, resta, resto, multiplicación y comparación en términos de mayor, mayor o igual, menor o menor o igual sólo se podrán realizar con números a excepción de la operación de suma, que con cadenas de caracteres adquirirá unas posibilidades especiales, ya que al sumar dos o mas cadenas, lograremos concatenarlas. Por ejemplo:

```
var
 string Nombre := “Bilbo”, Apellido := “Bolson”;
 string NombreCompleto;

begin
 // Como vemos, concatenamos la variable nombre que contiene la
 // candena “Bilbo”, la cadena vacia “ “ y por último la variable
 // apellido que contiene la cadena “Bolson”. El resultado será la
 // cadena “Bilbo Bolson”.
 NombreCompleto := Nombre + “ “ + Apellido;
end
```

C.2.5.4. Las constantes.

Las constantes no serán más que variables que no podrán cambiar su valor. Se declararán siempre después de la palabra clave `const`. Al igual que con las variables, las constantes no serán obligatorias.

A la hora de declarar constantes no podremos hacer uso de la posibilidad que nos brinda la coma, es decir, no podremos declarar varias constantes a la vez para un mismo tipo. Además, siempre deberenos de asignar un valor en su declaración.

Así, serán constantes bien declaradas:

```
const
 number PI := 3.141516;
 number Z := -23;
 string NOMBRE := “Bilbo”;
 entity ARBOL := 0;
```

Y estarán mal declaradas:

```
const
 number X, PI := 3.141516, Y, Z := -23;
 string NOMBRE := “Bilbo”, APELLIDO := “Bolson”;
 entity ARBOL;
```

C.2.5.4.1. Constantes globales internas.

Existirán una serie de constantes globales internas que servirán para más cómoda la programación y comprensible el código para otras personas. Al ser constantes internas no podrán declararse otras constantes, variables o funciones con igual identificador al utilizado para éstas.

El uso de constantes globales abarcara el trabajo con identificadores numéricos relativos a alineaciones, teclas, resultados de expresiones lógicas, etc. *CrisolScript* podría funcionar perfectamente sin estas constantes globales pero obligaría al programador a trabajar directamente con valores numéricos, haciendo mucho menos legibles los scripts y más difíciles de comprender a un simple vistazo (si bien, otra opción que tendría el programador sería la de definirse él mismo las constantes) cosa, ésta última, que haría incómoda y no inmediata la programación. Las constantes globales internas y sus valores se pueden encontrar en el anexo correspondiente.

A partir de ahora, cuando haya que poner un ejemplo, se utilizarán las constantes globales para facilitar la lectura de los mismos. Así, cuando mostrábamos los retornos de las funciones script, podríamos sustituir `return 1;` por `return TRUE;`

C.2.5.5. Conversión de tipos.

Por conversión de tipos se entenderá la capacidad que tendrá *CrisolScript* para trabajar con dos datos de diferente tipo, forzando la conversión de uno de ellos al otro para así poder realizar una operación lógica o matemática. *CrisolScript* incorporará una conversión de tipos muy básica, pero muy útil, entre cadenas de caracteres y números. Supongamos que tenemos dos variables:

```
var
 number X := 2002;
 string Name := “Crisol ”;
```

Pues bien, se podrán hacer cosas como estas:

```
begin
 Name := Name + X;
end
```

Con esto hemos logrado que la variable `Name` pase a ser `“Crisol 2002”`, es decir, hemos pasado el número que contenía la variable `X` a una cadena de caracteres.

Esta misma operación a la inversa también será posible si el `string` es un valor numérico. Supongamos:

```
begin
 Name := “33”;
 X := X + Name;
end
```

El resultado será que `X` contendrá el valor 2035, pues primero se convirtió `Name` a un valor numérico y luego se sumo a `X` para finalmente añadirlo a `X` también. En caso de que el `string` `Name` tuviera una cadena de caracteres que no fuera un número (por ejemplo, el valor `“Crisol”`) y se repitiera la operación, el resultado sería que la cadena se transformaría al valor numérico 0.

Todo lo explicado también será aplicable a operaciones directas de asignación de cadena a número. Por ejemplo:

```
begin
 // Asignamos a la variable X el valor 33
```

```
 Name := “33”;
 X := Name;
end
```

Y de número a cadena:

```
begin
 X := 33;
 Name := X;
End
```

Ahora `Name` pasará a contener la cadena “33”.

C.2.5.6. Argumentos en scripts y funciones.

Tanto los scripts como las funciones podrán poseer argumentos. La única diferencia entre ambas construcciones se centrará en que los scripts poseerán unos argumentos cuyos tipos están determinados, precisamente, por el tipo del script (el tipo del evento) que se esté programando. Esto es necesario pues dependiendo del evento al que estén asociados, los parámetros a pasar serán distintos (por ejemplo, el script `OnObserveEntity` será un script que tendrá dos argumentos de tipo `entity` siempre). Debe de quedar claro que los argumentos serán las variables con los tipos de valores que un script y/o función necesite para para funcionar mientras que los parámetros serán los valores que se pasen a los scripts y/o funciones cuando sean invocados (en el caso de los scripts, la invocación siempre se realizará desde *Crisol*).

En caso de que un script o una función deba de recibir parámetros éstos tendrán de colocarse dentro de los paréntesis que van después del tipo del script o del nombre de la función. Además, cuando el número de ellos sea superior a uno deberán de ir separados por comas.

Para declarar una función que deba de recibir parámetros, sus argumentos se pondrán con el formato tipo de argumento y nombre de argumento. Por ejemplo:

```
void function FuncionDePrueba(number Argumento1, string Argumento2)
begin
 // Cuerpo de la función
end
```

En el ejemplo anterior vemos una función con dos argumentos; un número y una cadena de caracteres. Los parámetros se pasarán desde el exterior cuando se llame a la función y podrán ser variables del tipo que corresponda al parámetro o constantes. Por ejemplo, serían llamadas válidas:

```
FuncionDePrueba(23.3, “Hola”)
FuncionDePrueba(Numero, “”)
FuncionDePrueba(0, Nombre)
```

Siendo `Numero` y `Nombre` dos variables o constantes previamente declaradas (la primera deberá de ser tipo `number` y la segunda de tipo `string`).

Nótese que gracias a la conversión de tipos un parámetro de tipo `string` podría conectar con argumentos `number` y viceversa.

Cuando una función no tenga argumentos asociados, no se le podrá pasar parámetro alguno. Utilizando la palabra reservada `void` (vacio), podremos declarar funciones de este tipo (nótese, que los script que no reciben parámetros también se declaran utilizando `void`). Como ejemplo de función sin argumentos podremos tener:

```
void function FunciónDePrueba(void)
Begin
```

```
 // Cuerpo de la función
end
```

Para invocar a una función que no tiene argumentos bastará con poner el nombre de ésta seguida de los paréntesis vacíos. Por ejemplo:

```
FunciónDePrueba()
```

C.2.5.6.1. Funciones con argumentos por referencia.

La forma normal de pasar parámeros a una función es por valor. Esto quiere decir que a una función le podemos pasar como parámetro una variable o una constante. Sin embargo, a diferencia de los scripts, las funciones también admiten el paso de parámetros por referencia.

Pasar un parámetro por referencia supone que la función esperará recibir una variable, Nunca una constante, y que el argumento que recoja la variable pasada como parámetro apuntará a la misma dirección de memoria que la variable original. En otras palabras, que cualquier cambio que se efectúe sobre el argumento que recoja la variable pasada a la función, como parámetro, repercutirá también en la variable original.

Para indicar que queremos que un argumento sea por referencia, se deberá de colocar la palabra reservada `ref` antes del tipo del mismo en la función. Nótese, de nuevo, que sólo está permitido para funciones.

Como ejemplo, vamos a ver la definición de una función que recibe un parámetro por valor y no por dirección y que le suma un valor.

```
void function SumaUnNumero(number Argumento)
begin
 Argumento := Argumento + 10;
end
```

Si ahora llamamos a dicha función así:

```
var
 number Parametro := 5;
begin
 SumaUnNumero(Parametro);
end
```

El valor de `Argumento` dentro de `SumaUnNumero` es de 15 pero al regresar `Parametro` sigue valiendo 5. Sin embargo, vamos a suponer ahora que `SumaUnNumero` recoja los parámetros recibidos por referencia. Tendremos:

```
void function SumaUnNumero(ref number Argumento)
begin
 Argumento := Argumento + 10;
end
```

Si ahora llamamos a dicha función así:

```
var
 number Parametro = 5;
begin
 SumaUnNumero(Parametro);
end
```

El valor de `Argumento` dentro de `SumaUnNumero` es de 15 y al regresar el valor de `Parametro` es también 15. ¿Por qué?. Muy sencillo, porque indicamos a la función `SumaUnNumero` que la variable que toma el valor de `Parametro`, esto es `Argumento`, lo

hiciera por referencia (`ref`), de tal forma que cualquier cambio que se realizará en `Argumento` se viera también reflejado en `Parametro`. Nótese que si `Parametro` fuera constante en lugar de variable habría un error a la hora de compilar.

C.2.5.7. El tipo de una función.

Todas las funciones tienen un tipo asociado. El tener un tipo indica que dicha funcion debe de devolver, o no, un valor acorde a dicho tipo. El tipo de una función se pone siempre antes de la palabra cable `function`.

Los posibles tipos de una función serán:

- El tipo `void`. Cuando la función tiene el tipo vacío o nulo y no tiene que retornar valor alguno.
- El tipo `number`. Cuando la función debe de retornar un valor numérico.
- El tipo `string`. Cuando la función debe de devolver una cadena de caracteres.
- El tipo `entity`. Cuando la función debe de devolver una entidad.

Cuando la función tiene un tipo distinto a `void` se puede utilizar ésta dentro de una expresión, es decir, como si fuera una variable o una constate. Por ejemplo, supongamos la siguiente función que, dado un número recibido, lo multiplica por 10 y lo retorna:

```
number function MultiplicaPorDiez(number Value)
begin
 // Multiplica por 10 y retorna
 return (10 * Value);
end
```

Con esta función podremos hacer cosas como:

```
var
 number Result;
 string StrResult;
begin
 Result := MultiplicaPorDiez(5) / 10;
 StrResult := “5 * 10 = ” + Result * MultiplicaPorDiez(1);
end
```

C.2.5.8. Estructuras de control.

CrisolScript incorpora una serie de estructuras de control inspiradas en diversos lenguajes de programación si bien, *CrisolScript* hará uso de una mínima parte de las posibles estructuras de control que podría utilizar. Por ejemplo, no tiene la estructura `for` ya que con la `while` se puede realizar el mismo trabajo. Del mismo modo, tampoco existirá la estructura `switch – case` ya que con los `if`’s anidados también se podrá suplir esa capacidad. El motivo de no usar estas estructuras es el de mantener un lenguaje pequeño y, sobre todo, fácil de utilizar para personas que nunca hayan programado.

Antes de proceder a enumerar las estructuras de control, conviene tener en cuenta que éstas siempre se ejecutarán en función de cómo se evalúe una expresión lógica. Dicha expresión se considerará verdadera si su valor es mayor o igual a 1 y falsa si su valor es inferior a 1.

C.2.5.8.1. Sentencia if / then / else.

Escogerá la ejecución del código detrás del `then` o detrás del `else` dependiendo de cómo se evalue la expresión entre paréntesis. Las dos formas sintáticas de la expresión serán:

```
if (expresión verdadera) then
 // Código
```

y

```
if (expresión verdadera) then
 // Código
else
 // Código
```

Nótese que en donde se indica que va el código sólo se podrá poner una línea, a no ser que se utilice un bloque `begin` ... `end` en cuyo caso, el hecho de que se entre en el mismo supondrá la ejecución de todas las líneas encerradas entre `begin` ... `end`.

C.2.5.8.2. Sentencia while / do.

Realizará una operación iterativa mientras la expresión que evalúa en cada iteración sea cierta. La única forma sintáctica será:

```
while (expresión verdadera) do
  // Código
```

Al igual que en la anterior sentencia, si no se usa un bloque `begin` ... `end` sólo se ejecutará una línea de código después del `do`.

C.2.5.9. El ámbito.

Por ámbito se entenderá la validez que tendrán las variables, constantes y funciones que sean declarados en un determinado punto de un script. A continuación se hablará de cada uno de los casos.

C.2.5.9.1. El ámbito en variables y constantes.

Las variables y constantes podrán ser declaradas como locales a un script o bien como locales a una funcion. En el primer caso, las variables y constantes sólo podrán ser usadas dentro del cuerpo del script o bien dentro del cuerpo de las funciones locales al script (en las funciones globales no se podrá usar estas variables y constantes). En el caso de hablar de funciones locales a una función, las variables y constantes sólo podrán ser usadas dentro del cuerpo de la función a la que son locales. Como no hay conflictos, se podrán usar los mismos nombres para variables o constantes en funciones distintas, así como en scripts.

En el caso de las constantes y variables de carácter global (se verán más adelante) se podrá acceder a las mismas desde cualquier punto de un script o función sin importar si ésta es local o global. Sin embargo, se deberá de tener en cuenta que si se declara una constante o variable local a un script o función con el mismo nombre que una variable o constante global, estaremos haciendo algo válido solo que a la hora de utilizar ese identificador, estaremos trabajando con el ámbito más inmediato, en este caso, mandaría más un identificador en una función o script que uno global.

Por ejemplo, supongamos que `X` es una variable global. Si tenemos una función así:

```
void function Prueba(void)
const
 number X := 10;
begin
 // Código
End
```

Todas las veces que usemos `X` entre el `begin` y el `end` de la función, se referirá al `X` local a la misma.

C.2.5.9.2. El ámbito en las llamadas a funciones locales a un script.

Aquellas funciones que sean locales a un script, las que se definen antes del cuerpo de éste, podrán ser llamadas únicamente desde el cuerpo del script o bien desde el cuerpo de otras funciones que sean locales al script. Nunca podrán ser llamadas desde funciones globales o desde el cuerpo de otro script o funciones locales a otros scripts diferentes al que son declaradas (por esto mismo, dos scripts distintos podrán tener funciones locales con el mismo nombre).

C.2.5.10. Uso de funciones globales en un script.

Resulta de una gran importancia poder definir funciones en archivos distintos a un script de cara a poder utilizarlas en distintos scripts. Por ejemplo, podemos pensar en crear un conjunto de funciones que sean necesarias para distintos tipos de script. Si no pudieramos definirlas en archivos independientes, para cada script que crearamos y que las necesitasen, deberíamos de volver a tener que escribirlas como locales a los mismos. Esto, además de una pérdida de tiempo, será caótico para trabajar a corto plazo.

Para solucionar este problema, *CrisolScript* permite que el diseñador cree archivos en donde existan funciones. Estas funciones serán iguales, en su formato, que las vistas como locales a un script. Por ejemplo, un archivo de funciones globales podría ser así:

```
// Este es el archivo FuncionesGlobales.cst

void function FuncionGlobal_1(Parámetros)
const
 // Constantes locales a la función
var
 // Variables locales a la función
begin
 // Cuerpo de la función local al script
end

void function FuncionGlobal_2(Parámetros)
const
 // Constantes locales a la función
var
 // Variables locales a la función
begin
 // Cuerpo de la función local al script
end

// ...

void function FuncionGlobal_N(Parámetros)
const
 // Constantes locales a la función
var
 // Variables locales a la función
begin
 // Cuerpo de la función local al script
end
```

Estas funciones definidas en el archivo “`FuncionesGlobales.cst`“ podrá ser usadas, en cualquier script a partir de ahora. Lo único que tendríamos que hacer sería usar la palabra reservada `import` en la parte que va justo después de definir los parámetros de un script. Por ejemplo, si queremos usar el archivo con las funciones globales definidas anteriormente, podríamos escribir cosas como éstas:

```
script NombreScript(Parametros)

// Aquí importamos las funciones globales
```

```
import “FuncionesGlobales.cst”

const
 // Constantes locales
var
 // Vbles locales

func
 // Funciones locales
void function FunciónLocal(Parametros)
const
 // Constantes locales
var
 // Vbles locales
begin
 // Código
end

begin
 // Ahora desde el script podremos llamar a funciones definidas en
 // FuncionesGlobales.cst además de a FunciónLocal
 FunciónLocal(Parámetros);
 FunciónGlobal2(Parámetros);
end
```

Recordemos que las funciones locales a un script podrán llamar a las funciones globales importadas al mismo, pero las funciones globales importadas sólo podrán llamar a otras funciones globales que se hallen definidas en el mismo archivo en donde estén definidas ellas. De igual forma, las funciones locales a un script podrán acceder a variables y constantes definidas en el espacio global, en el script y como locales a ellas mismas pero las funciones globales importadas sólo podrán actuar sobre las variables y constantes definidas en el espacio global y las definidas en su propio cuerpo (nunca podrán acceder a las del script que las importa).

C.2.5.11. Objetos y métodos.

CrisolScript incorpora una noción muy elemental de programación orientada a objetos. Por un lado existirán dos objetos globales y por otro las variables de tipo `entity`. En ambos casos, en este documento, estaremos hablando de objetos (aunque sólo sea un maquillaje para facilitar el trabajo al programador de *CrisolScript*).

Existirán dos objetos globales. Uno será `Game`, que representará el engine como aplicación, y el otro será `World`, que representará el universo de juego. A parte de estos dos objetos globales, las variables de tipo `entity` representarán las entidades del juego, esto es, las paredes, ítems, criaturas (jugador y npc’s) y objetos de escenario (tanto los que sean contenedores como los que no). Estas variables no serán más que objetos que tendrán, para cada tipo de entidad, una serie de métodos particulares si bien, a parte existirá un amplio número de métodos comunes a todas ellas.

Un objeto se deberá de entender como una variable o constante que tiene asociada funciones. Así, para poder utilizar el objeto `Game`, sólo nos hace falta conocer las funciones que posee. Una vez sabido esto, podremos usarlas. Por ejemplo, si queremos enviar un mensaje a la consola de juego, tendremos que usar el objeto `Game` de esta forma:

```
Game.WriteToConsole(“Hola”)
```

En este ejemplo, podemos ver que entre el objeto y la función que utilizamos existe un punto ‘.’. El punto servirá para que *CrisolScript* sepa distinguir entre objeto y función. El objeto irá a la izquierda y la función a la derecha del punto respectivamente. Las funciones de los objetos se denominarán métodos a partir de ahora con el fin de facilitar su distinción entre las funciones

que el diseñador pueda definir (y otras que veremos más adelante denominadas funciones del API).

Todos los métodos de los objetos `World` y `Game`, así como los de las entidades, estarán creados en el interior de *Crisol*, por lo que el programador de *CrisolScript* tan sólo deberá de utilizar la documentación de referencia para poder usarlos (consultar apéndices).

En el caso de los objetos `World` y `Game`, *CrisolScript* podrá adivinar fácilmente si se está llamando a un método correcto o incorrecto sin embargo, con las entidades no ocurrirá lo mismo ya que podemos usar una variable de tipo entidad y llamar a cualquiera de los métodos existentes para las criaturas, ítems, objetos de escenario y paredes y, en este caso, *CrisolScript* no podrá hacer nada en tiempo de compilación (mientras lee el código fuente para transformarlo a código intermedio). Sólo podrá detectarse el problema en tiempo de ejecución (cuando el script comience a ser ejecutado en el motor).

Siempre que exista algún tipo de error en tiempo de ejecución el script se interrumpirá. El error podrá ser consecuencia de un problema de contexto (lanzar el método desde un estado de juego no válido), de ejecutar un método asociado a una entidad nula, de que los parámetros sean incorrectos, etc.

C.2.5.11.1. El objeto Game.

El objeto `Game` se encargara de representar a *Crisol* en calidad de apliación. Esto quiere decir que trabajará principalmente con cuadros de diálogo e interfaces.

La relación de métodos pertenecientes a este objeto se pueden hallar en el anexo correspondiente.

C.2.5.11.2. El objeto World.

El objeto `World` se encargará de representar el universo de juego, las distintas áreas, el control global de las entidades de juego, etc.

La relación de métodos pertenecientes a este objeto se pueden hallar en el anexo correspondiente.

C.2.5.11.3. Los objetos entity.

El control de las entidades de juego se llevará a cabo a través de las variables de tipo `entity`, que podrán referenciar a una criatura, jugador o no, a un ítem, a un objeto de escenario o a una pared. Independientemente del tipo de entidad que represente una variable o constante `entity`, existirá un número de métodos que serán comunes a todas ellas. Si bien, por cada tipo concreto habrá una serie de métodos que sólo serán válidos para dichos tipos concretos (es decir, las entidades de tipo criatura podrán tener métodos que no se hallen en las entidades de tipo pared pero, a su vez, ambos tipos de entidades podrán tener métodos comunes, que se hallen disponibles para las dos).

Siempre que se utilice una variable de tipo `entity` para ejecutar un método asociado, *Crisol* deberá de comprobar que se cumplan dos condiciones, por un lado que la variable `entity` apunte a una entidad válida en el universo de juego y, por otro, que el método utilizado esté disponible para ese tipo de entidad. Cuando esto no ocurra, el script donde estuviera localizado el uso de la variable `entity` será interrumpido.

La relación de métodos pertenecientes comunes a todos los tipos de entidades, así como aquellos particulares de cada tipo, se podrán consultar en el anexo correspondiente.

C.2.5.12. Funciones del API.

Las funciones del API serán funciones que podrán usarse en cualquier momento y desde cualquier punto. Su objetivo será el de servir de punto de apoyo para facilitar la programación con el motor. En algunos casos, como en el de las funciones destinadas a trabajar con los colores RGB, su uso será necesario.

Todas las funciones del API comenzarán siempre por API y a continuación le seguirá el nombre de la función propiamente dicho.

Las funciones del API existentes podrán consultarse en el anexo correspondiente.

C.2.5.13. El archivo de definiciones globales y de scripts a compilar.

Este archivo jugará un papel fundamental pues en él se definirán las variables y constantes globales, así como los archivos script (nunca archivos de funciones globales) que se desearán compilar. Las constantes y variables globales serán aquellas que podrán ser accesibles desde cualquier script y función, tanto locales como globales. Los archivos script a compilar irán después de indicar las posibles constantes o variables globales y sólo podrán ser utilizandos en *Crisol* aquellos que sean especificados aquí para ser compilados. El formato vendrá a ser el siguiente:

```
global
 const
  // Declaración de constantes globales
 var
  // Declaración de variables globales

  // Nombres de los archivos script a compilar entre comillas
  // y con extensión
 compile “NombreArchivoScript1.cst”;
  // ...
 compile “NombreArchivoScriptN.cst”;
```

Gracias a este fichero, el compilador será capaz de generar un archivo con el código intermedio y los datos suficientes para que el ínterprete embebido en *Crisol* sepa ejecutar el código de los distintos scripts o saber si existe un archivo asociado a un determinado evento.

C.3. El compilador CSCompiler.

Para que el motor pueda utilizar el código fuente programado, se deberá de compilar éste, antes utilizando el compilador en línea *CSCompiler*. Dicho compilador tendrá dos objetivos. Por un lado el de generar un archivo de salida denominado "`CrisolGameScripts.csb`", que será el que *Crisol* irá a buscar para localizar los distintos scripts (en caso de no hallarlo se entenderá que el diseñador no ha programado ningún script) y por otro el de avisar al diseñador de los posibles errores que haya cometido al programar.

El compilador siempre intentará ofrecer la mejor y más completa ayuda cuando localice errores, indicando el archivo y la línea en donde encontrarlos. Gracias a esta información, el diseñador podrá repasar lo que ha escrito y corregir los fallos que el compilador haya localizado. Cuando existan errores no se generará el archivo "`CrisolGameScripts.csb`".

Para poder compilar la totalidad de los archivos script, se deberá de suministrar a *CSCompiler* el nombre (junto a su extensión) del archivo de definiciones globales. En él encontrará qué archivos scripts deberá de abrir para compilar. Por ejemplo, si nuestro archivo de definiciones globales se llamara "`Global.csb`" bastaría escribir:

```
CSCompiler Global.csb
```

Al ejecutar *CSCompiler* emitirá el archivo "`CrisolGameScripts.csb`" si no halló ningún error.

C.3.1. Las opciones de CSCompiler.

CSCompiler dispone de una serie de opciones para el trabajo de compilación. Todas ellas se introducirán siempre delante del nombre del archivo de definiciones globales y precedidas de un guión. Así, el formato será `–X` donde `X` hará referencia a un carácter que indique la opción a realizar. Todas las opciones, a menos que se diga lo contrario, podrán ser usadas de forma simultánea y en cualquier orden (siempre y cuando precedan al nombre del archivo con definiciones globales). Las distintas opciones serán:

- **Emitir los mensajes en un fichero**. Esta opción será la normalmente usada ya que por defecto *CSCompiler* escribirá todos los mensajes en pantalla, resultando que si existen muchos errores o avisos, estos no llegarán a verse por completo, haciendo que sea imprescindible que el compilador envíe toda la información a un fichero. Usando el carácter `f` indicaremos al compilador que emita los mensajes a un fichero. Así, si escribimos:

  ```
  CSCompiler –f Globals.cst
  ```

  se creará un archivo llamado "`CSResult.txt`" que contendrá todos los mensajes para que puedan ser examinados con total tranquilidad.

- **Mostrar el código intermedio generado**. Para que el compilador, en caso de no encontrar ningún error, genere un archivo de texto con el código intermedio generado pero de cara a ser examinado por un programador. Habrá que utilizar el carácter e. Así, si escribimos:

  ```
  CSCompiler –e Globals.cst
  ```

  se creará un archivo llamado "`CSOpcodes.txt`" en donde se motrará un código intermedio "entendible". Esta opción interesará únicamente a los programadores que usen directamente el código fuente de *Crisol* quieran realizar sus propios ajustes, o para realizar trazas por si han existido errores en el compilador no localizados en la etapa de testeo.

  Un ejemplo del contenido de este archivo podría ser:

  ```
  .global Globals.cst
   [    0] npush 0.000000
   [    1] nstore address_0
  ```

```
 [    2] npush 0.000000
 [    3] nstore address_1
.end_global

.script Script_OnClickHourPanel.cst
.event on_click_hour_panel (V)N [Index: 0]
.vars 1
.first_offset 2
.max_stack_size 2
 [    0] npush 1213.000000
 [    1] nstore address_2
 [    2] spush "Son las "
 [    3] world.get_hour
 [    4] nscast
 [    5] sadd
 [    6] spush " y "
 [    7] sadd
 [    8] world.get_minute
 [    9] nscast
 [   10] sadd
 [   11] game.write_to_console
 [   12] npush 1.000000
 [   13] nreturn

.import_functions
.end_import_functions

.local_functions
.end_locals_functions

.end_script
```