

**Lazarus**

**Resaltador Configurable para SynEdit**

**TSynFacilSyn 0.8.3**

**FICHA TÉCNICA**

<b>AUTOR</b>	Tito Hinostroza – Lima Perú
<b>FECHA</b>	Rev1 terminada en 15/07/2014
<b>APLICABLE A</b>	Paquete SynEdit de Lazarus 1.2 Los ejemplos se han desarrollado sobre Windows-32.
<b>NVEL DEL DOCUMENTO</b>	Medio-Alto. Se asume conocimientos de Free Pascal, Lazarus, y manejo del control SynEdit
<b>DOCUMENTOS PREVIOS</b>	Se sugiere leer “La Biblia del SynEdit”.
<b>BIBLIOGRAFÍA</b>	Código fuente de SynEdit - Lazarus <a href="http://forum.lazarus.freepascal.org/">http://forum.lazarus.freepascal.org/</a> <a href="http://wiki.freepascal.org/SynEdit/es">http://wiki.freepascal.org/SynEdit/es</a> <a href="http://wiki.freepascal.org/SynEdit">http://wiki.freepascal.org/SynEdit</a>

## NOTAS SOBRE LA VERSIÓN 0.8.3

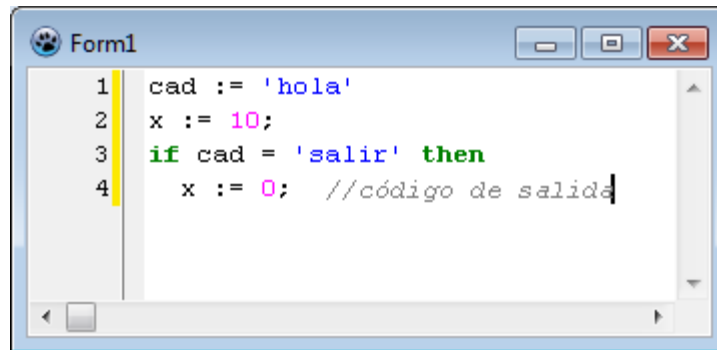
Solo se realizan cambios internos para ordenar mejor el código. Se cambian los nombres de algunos elementos de la librería. Todos ellos internos.

Muchas propiedades de tipo PRIVATE, pasan a ser ahora PROTECTED, para permitir acceder a ellas desde clases derivadas. Además algunos métodos pasan a ser ahora virtuales, para permitir que se puedan redefinir.

Se incluye la etiqueta <COMPLETION>, para que pueda ser reconocida en el archivo XML, pero no se procesa. Se mantiene así para dejar el procesamiento a una clase derivada.

# 1 Introducción

El resaltador TSynFacilSyn, para el componente SynEdit, es similar al resaltador SynAnySyn de Lazarus, pero más eficiente y mucho más configurable. Todo el código de este resaltador se encuentra definido en una sola unidad.



El resaltador TSynFacilSyn es capaz de leer una sintaxis completa definida en un archivo externo. Este archivo de sintaxis, especifica el modo de trabajo que debe usarse para el resaltador. Un archivo de sintaxis es un archivo en formato XML que solo define un lenguaje. Si se desean tener diferentes lenguajes, se deben tener diferentes XML, uno para cada lenguaje.

También se puede configurar la sintaxis del resaltador, por código, usando los métodos de la clase "TSynFacilSyn".

El resaltador TSynFacilSyn solo trabajará con el componente SynEdit de Lazarus. Implementa el resaltado de sintaxis y el plegado de código.

## 1.1.1 Características

- Es configurable. Permite definir una sintaxis para el resaltado, usando un archivo externo XML o usando instrucciones de código.
- Es de ejecución rápida y ligero en cuanto a memoria. Está optimizado para ser comparable en velocidad a un resaltador no-configurable.
- Permite configurar de forma sencilla la mayoría de elementos de una sintaxis, como identificadores, cadenas, números y comentarios, de forma que se adapta a la mayoría de lenguajes.
- Está orientado al manejo de tokens y atributos. Cada token contiene un único atributo.
- Permite definir los caracteres que son válidos para los identificadores. Así se puede adaptar a la definición específica de cada lenguaje.

- Permite manejar subconjuntos de identificadores (palabras reservadas, variables, directivas, etc) y subconjuntos de símbolos (operadores, separadores, etc).
- Permite definir comentarios y cadenas multi-línea.
- Permite crear tokens nuevos, usando identificadores como delimitadores (similar a los bloques <<HEREDOC).
- Incluye la propiedad “CaseSensitive”, para poder reconocer identificadores ignorando la caja.
- Permite definir bloques con o sin plegado de código. Incluye diversas opciones para adecuarse a la mayoría de lenguajes de programación.
- Puede colorear el fondo de los bloques.
- El archivo de sintaxis en XML, permite formas resumidas para la definición de tokens.
- Maneja diversos atributos predefinidos, pero pueden crearse más dinámicamente.

## 2 Resumen

### 2.1 *Uso dentro de un programa.*

Este resaltador, se compone de un solo archivo que es la unidad "SynHighLighterFile.pas". Se debe descargar este archivo y copiarlo en la carpeta donde está el proyecto de trabajo.

Una vez copiado, se debe incluir en la sentencia USES, de las unidades que requieran trabajar con este resaltador.

Como con cualquier resaltador, se debe crear una instancia del resaltador (TSynFacilSyn) y asignarla al editor que vayamos a usar:

```
uses    ... , SynHighlighterFacil;  
  
var  
    hlt : TSynFacilSyn;  
  
...  
    hlt := TSynFacilSyn.Create(self); //crea resaltador  
    editor.Highlighter := hlt;  
    hlt.LoadFromFile('SynPHP.xml');   //carga archivo de sintaxis  
...  
    hlt.Free; //libera
```

Para usar el resaltador, es necesario configurar primero la sintaxis. Esto se puede hacer usando un archivo externo (archivo de sintaxis) o por código.

### 2.2 *Configuración usando archivo de sintaxis*

Por ejemplo si deseamos una sintaxis simple de archivos PHP, podríamos crear un archivo XML y poner este contenido:

```
<Language name="php">  
  <String Start="'" End="'"></String>  
  <String Start="&quot;" End="&quot;"></String>  
  <Comment Start="//"></Comment>
```

```
<Comment Start="/" End="/" ></Comment>
<Keyword>
__file__ __line__ array and break case class const continue default die do
doubl  echo else elseif empty endfor endif endswitch endwhile eval exit extends
false float for function global highlight_file highlight_string if include int
integer isset list new object old_function or print real require return
show_source static string switch true unset var while xor
</Keyword>
</Language>
```

Luego deber amos usar el m todo LoadFromFile(), para configurar al resaltador con este archivo.

## 2.3 Configuraci n usando C digo

La sintaxis tambi n puede configurarse usando c digo, en lugar de un archivo externo. Esto nos da mayor flexibilidad, pero requiere m s cuidado. El siguiente c digo permite definir la sintaxis del archivo XML mostrado anteriormente:

```
hlt1.ClearMethodTables;
hlt1.ClearSpecials;
hlt1.DefTokIdentif('$A..Za..z_', 'A..Za..z0..9_');

hlt1.AddKeyword('__file__');
hlt1.AddKeyword('__line__');
hlt1.AddKeyword('array');
hlt1.AddKeyword('and');
...
hlt1.AddKeyword('while');
hlt1.AddKeyword('xor');

hlt1.AddTokenString('',' ',false, false);
hlt1.AddTokenString('"','"',false, false);
hlt1.AddTokenComment('//',' ',false);
hlt1.AddTokenComment('/*','*/',false);
hlt1.Rebuild;
```

## 2.4 Cambiando los atributos

Los atributos se pueden cambiar en el archivo XML o usando c digo.

El siguiente archivo XML, muestra cómo se definen las propiedades de los atributos:

```
<Language name="SQL Oracle" ext="psql" ColorBlock="Block">
  <Attribute Name="Keyword" ForeCol="#40a040" Bold="False" BackCol="#404040">
  </Attribute>
  <Attribute Name="Number" FrameCol="#4040a0" Italic="true" StrikeOut="true">
  </Attribute>

  ...

</Language>
```

Para mayor información sobre la configuración de atributos ver la sección 4.3 - Configurando los atributos

Para cambiar los atributos por código, se deben acceder a las propiedades de atributos:

```
//carga sintaxis
hlt := TSynFacilSyn.Create(self);
hlt.LoadFromFile('SynPHP.xml');
//cambia el color del texto de Keywords
hlt.tkKeyword.Foreground :=clGray;
//cambia el color de fondo de las cadenas
hlt.tkString.Background :=clyellow;
//cambia el color del texto de comentarios
hlt.tkComment.Foreground :=clRed;
```

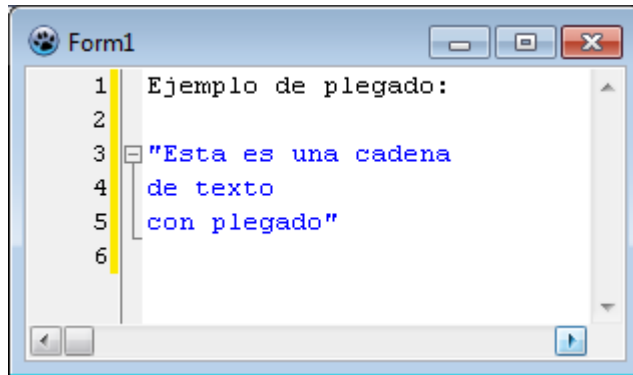
Existen diversos atributos predefinidos en el resaltador, pero solo es necesario configurar los que se van a usar.

Algunos atributos, tienen propiedades establecidas por defecto como las palabras claves y los comentarios.



## 2.5 Plegado de código

El resaltador TSynFacilSyn, incluye la funcionalidad de plegado de código (folding), y es también configurable:



El plegado de código se puede activar para tokens multilínea o para bloques de código de varios tokens.

Para la figura mostrada, se ha definido cadenas multilínea con plegado:

```
<String Start="&quot;;" End="&quot;;" Folding="true"></String>
```

Para definir bloques de plegado delimitados por tokens, se puede usar la siguiente definición:

```
<Block Start="BEGIN" End="END" Folding="True"></Block>
```

El plegado se puede configurar, tanto desde el archivo de sintaxis, como por código. Más adelante se indicará como activar el plegado.

## 3 Entendiendo una sintaxis

### 3.1 Elementos de la sintaxis

Una sintaxis puede verse a diferentes niveles. Sabemos que a bajo nivel, los lenguajes son solo una serie de caracteres, e inclusive a más bajo nivel, los lenguajes son solo conjunto de bytes (y bits).

Consideremos una secuencia sencilla de un lenguaje de programación:

bytes	7B	78	3D	31	32	3B	63	61	64	3D	22	74	C3	BA	22	7D	3B	32	32	2F	2F	63	6F	6D	65	6E	74
caracteres	{	x	=	1	2	;	c	a	d	=	"	t	ú	"	}	;				/	/	c	o	m	e	n	t
tokens	simbolo	identificador	simbolo	número	simbolo	identificador	simbolo	cadena					simbolo	simbolo	espacio	comentario											

Observar que la correspondencia entre bytes y caracteres, no siempre es de uno a uno, ya que dependerá de la codificación que se haga. En nuestro caso, SynEdit maneja la codificación UTF-8, que para algunos caracteres maneja más de un byte. Además las tabulaciones que se representan como un byte, ocupan varios caracteres en pantalla.

El resaltador TSynFacilSyn procesa los caracteres como si fueran bytes únicos. Los caracteres acentuados los procesa como si fueran dos caracteres (porque en realidad se guardan como dos bytes en UTF-8).

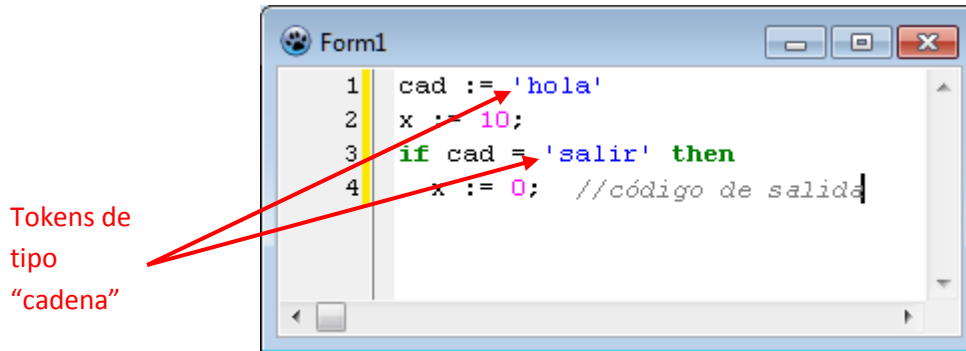
Para el resaltado de sintaxis, el resaltador identifica tokens. Ese es el nivel necesario para definir atributos de texto.

Es necesario definir dos conceptos claves para entender el funcionamiento del resaltador TSynFacilSyn:

Un token es uno o más caracteres agrupados por reglas específicas. Cada token tiene asignado uno y solo un atributo, que puede ser un color o fondo diferente.

Los atributos son las propiedades de resaltado que se aplican a un token particular. En SynEdit, un atributo está representado por un objeto de la clase “TSynHighlighterAttributes” y permiten definir propiedades como el color del texto, el color del fondo, el tipo de letra, o la activación de negrita.

Las cadenas y comentarios se identifican siempre como un solo token y tienen un único atributo.



Un resaltador es en cierta forma, un analizador léxico (lexer), porque analiza el texto e identifica “tokens”, que son los elementos léxicos de un lenguaje<sup>1</sup>.

Los tokens, se clasifican en diversos tipos dentro del resaltador<sup>2</sup>. Los tipos se pueden crear dinámicamente, pero existen 8 tipos pre-definidos:

1. Fin de línea
2. Símbolos
3. Espacios
4. Identificadores
5. Números
6. Palabras clave
7. Cadenas
8. Comentarios

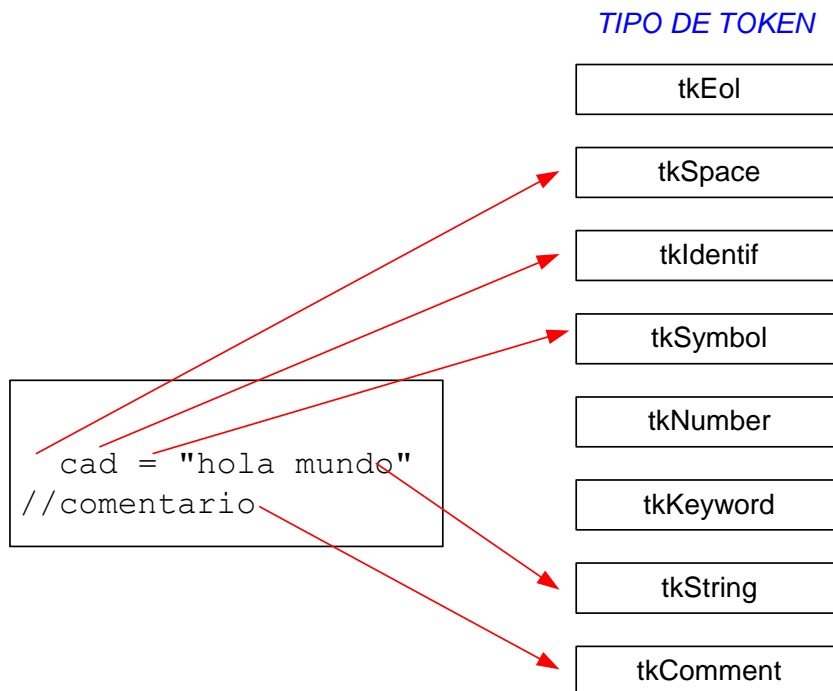
Estos tipos de tokens siempre existen dentro del resaltador, y sirven para el manejo interno y para cubrir las necesidades básicas de resaltadores sencillos.

<sup>1</sup> Por encima del nivel léxico, está el nivel sintáctico y semántico. Usualmente un resaltador no se mueve por estos niveles, excepto que sea fácil el análisis o sea muy importante (como en el caso de las palabras claves.)

<sup>2</sup> Otros resaltadores de Lazarus pueden manejar otros tipos de tokens.

Los otros tipos de tokens como variables, constantes, funciones, procedimientos, tipos, etc, deben crearse primero antes de usarse dentro del resaltador. En la sección 4 y 5 se indicará como se pueden crear los tipos adicionales de tokens.

Cuando se aplica el resaltador al texto de un editor, todos los tokens pasan a ser ubicados en alguno de los tipos existentes:



Un token solo puede pertenecer a un tipo de token.

Los diversos tipos de token se identifican por una variable interna del resaltador:

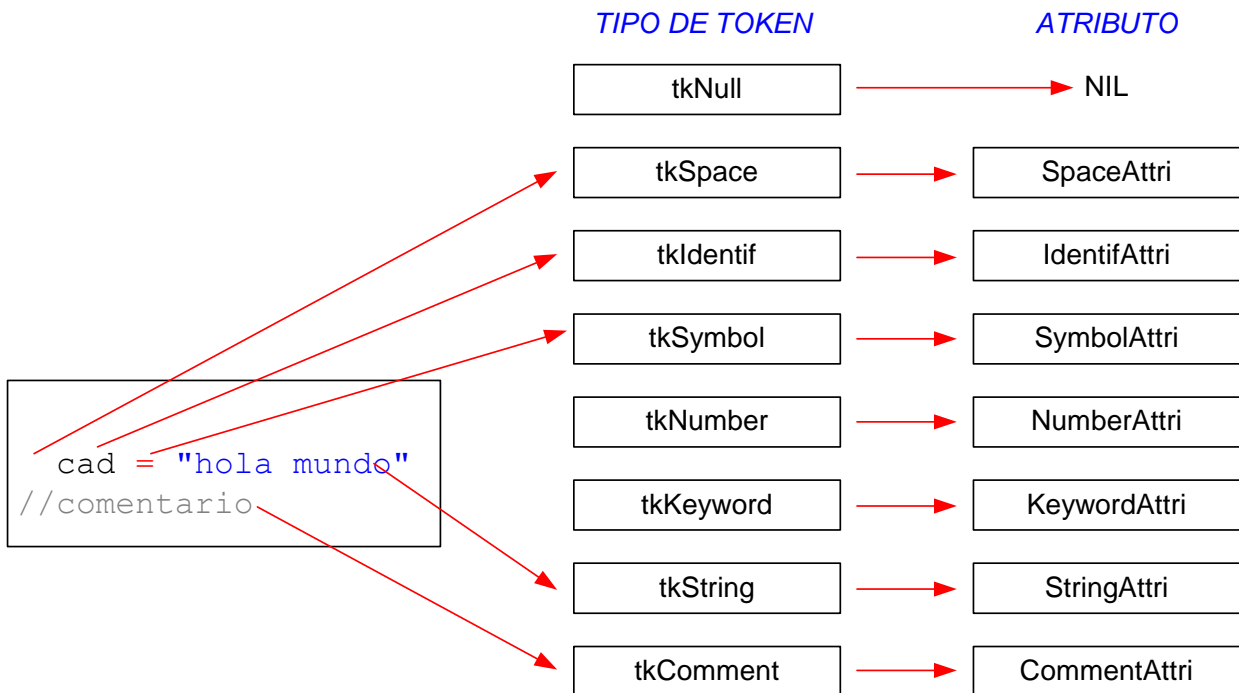
```
//atributos predefinidos
tkEol      : TSynHighlighterAttributes; //id para los tokens salto de línea
tkSymbol   : TSynHighlighterAttributes; //id para los símbolos
tkSpace    : TSynHighlighterAttributes; //id para los espacios
tkIdentif  : TSynHighlighterAttributes; //id para los identificadores
tkNumber   : TSynHighlighterAttributes; //id para los números
tkKeyword  : TSynHighlighterAttributes; //id para las palabras claves
tkString   : TSynHighlighterAttributes; //id para las cadenas
tkComment  : TSynHighlighterAttributes; //id para los comentarios
```

Y se usan para tipificar a los tokens, dentro del programa. Cada uno de estas variables son referencias a objetos reales dentro del resaltador. Estas referencias a “TSynHighlighterAttributes”, son las que SynEdit espera recibir del resaltador cuando llama al método GetTokenAttribute().

A diferencia de los resaltadores existentes en Lazarus, en TSynFacilSyn no se usan enumerados para identificar a los tipos de tokens. Se usa la referencia al mismo atributo.

Saber a qué tipo pertenece un token es necesario para poder aplicarle el atributo que corresponda. Un atributo configura propiedades como el color de texto o de fondo.

La norma es que cada tipo de token, está asociado a un único atributo, como se aprecia en el siguiente diagrama:



Los tipos de tokens pre-definidos se listan en la siguiente tabla:

TIPO DE ATRIBUTO	NOMBRE EN EL CÓDIGO	NOMBRE EN XML	DESCRIPCIÓN
Fin de línea	tkEol	EOL	Usado para detectar el fin de línea.
Símbolo	tkSymbol	SYMBOL	Representa a los símbolos que no son identificadores, números, espacios, cadenas o comentarios.
Espacios	tkSpace	SPACE	Representa a los caracteres no imprimibles en un lenguaje. Incluye a los saltos de línea y caracteres de control.
Identificador	tkIdentif	IDENTIFIER	Representar a los identificadores en un lenguaje. Estos identificadores pueden ser nombres de variables, funciones, etc.
Números	tkNumber	NUMBER	Representa a las constantes numéricas en un lenguaje, como 99, 0.243, 0xA0, \$FF
Palabra clave	tkKeyword	KEYWORD	Son identificadores especiales. Suele representar a las palabras reservadas en un lenguaje, como FOR, BEGIN, RETURN.
Cadenas	tkString	STRING	Representa a las constantes de tipo cadena, usadas en la mayoría de lenguajes, como "hola", 'mundo'.
Comentarios	tkComment	COMMENT	Representa a los comentarios dentro del código de un programa.

La categoría de Palabra Clave, se han creado para aplicarse a ciertos tipos de identificadores, llamados "Identificadores Especiales". Así, formalmente podríamos haberlas definido, como una subcategoría de Identificadores, pero se les define de forma separada, porque en algunos lenguajes, podrían constituir elementos sintácticos totalmente diferentes, como es el caso de las variables en PHP, que se pueden identificar como tokens especiales porque empiezan con el carácter \$.

La cantidad de tipos distintos de tokens (atributos disponibles), indica la cantidad de características visuales distintas que se pueden aplicar a distintos elementos del texto de una sintaxis. Cuando los tipos predefinidos no son suficientes, es cuando se deben crear tipos personalizados de tokens.

Usualmente se asociará un atributo, a un tipo de elemento similar en un lenguaje, es decir, que los atributos de números se aplicarán a los números (constantes numéricas) de un lenguaje. Pero el resaltador TSynFacilSyn, permite tal flexibilidad, que podrían aplicarse atributos de números a identificadores o incluso cadenas.

Para ser precisos, se debe indicar que el resaltador aplicará, siempre, los atributos de acuerdo al tipo de token. Y es posible definir tokens de forma diversa, sin necesidad de seguir lo establecido.

En términos generales, el tipo de token determina que atributo a aplicar a todos los tokens que pertenezcan a ese tipo. Determinar que tokens pertenecen a tal o cual tipo, es parte de la definición de tokens (Ver Sección 3.2 - Identificadores especiales).

### **3.1.1 Identificadores**

Un identificador es un conjunto de caracteres unidos, generalmente alfanuméricos, que suelen empezar con una letra o símbolo y que están seguidos de letras, números o símbolos.

Cada lenguaje tiene su definición de identificador, pero la mayoría de ellos comparten las mismas características, como incluir a todos los caracteres alfabéticos.

Por lo general, los identificadores se consideran como un solo token, pero es posible definir tokens más extensos que tengan como delimitador a un identificador.

### **3.1.2 Números**

En muchos lenguajes, se consideran como tokens numéricos a los que contienen dígitos numéricos o algún carácter auxiliar. Por ejemplo: 5943, 0.123, .05, etc

También las formas \$FF y 0x1a, pueden ser consideradas números. En general, todas las constantes numéricas de un lenguaje deberían entrar en la categoría de tokens numéricos.

Cada lenguaje puede permitir caracteres diversos dentro de los números, pero por lo general serán: .0123456789

### 3.1.3 Espacios

Los espacios se definen como todos los caracteres con código menor a #32. Esto incluye a los verdaderos espacios, tabulaciones, caracteres de control, etc. A estos caracteres se les suele llamar caracteres blancos.

Un conjunto de caracteres blancos se consideran como un solo token de tipo espacio.

Por ejemplo las siguientes combinaciones son tokens espacio:

```
' '
```

```
#9#32#32
```

```
#1#2#3
```

Los caracteres considerados como espacios, son fijos en el resaltador y no se pueden cambiar. Estos tokens, también pueden admitir atributos pero se encuentran desactivados en la clase, para ganar velocidad. Si se desea activar atributos para los espacios, se debe modificar el método TSynFacilSyn.GetTokenAttribute().

También los caracteres #10 y #13, se consideran como espacios dentro del resaltador, si bien lo usual es que nunca se procesen dentro de una línea, porque deberían eliminarse al momento de cargarse en el editor.

### 3.1.4 Cadenas

Las cadenas son tokens especiales que están entre dos delimitadores de cadena. Los delimitadores de cadena suelen ser los símbolos comillas o apóstrofo, como "Hola" o 'mundo'. Son identificados en un lenguaje como constantes de tipo cadena.

Pueden haber definidos varios tipos de cadenas, con delimitadores diferentes. Por lo general, los delimitadores inicial y final suelen ser iguales, pero no necesariamente.

El siguiente ejemplo muestra un trozo de código en donde se han definido dos tipos de cadenas, entre comillas y entre apóstrofes:



bytes	78	3D	27	32	3D	3D	32	3D	20	27	3B	63	3D	22	74	C3	BA	22	7D	3B
caracteres	x	=	'	2	=	"	2	"		'	;	c	=	"	t	ú	"	}	;	
tokens	identificador	símbolo	cadena								símbolo	identificador	símbolo	cadena				símbolo	símbolo	

Se tienen algunas observaciones sobre las cadenas:

- Las cadenas pueden ocupar una o varias líneas.
- Los delimitadores también forman parte del token de la cadena.
- Las cadenas de una sola línea pueden contener cualquier carácter en su interior, incluyendo delimitadores de otro tipo de cadena (los cuales se ignoran).
- Las cadenas no se pueden anidar. Una cadena del mismo tipo no puede estar dentro de otra cadena.
- Las cadenas se componen de un solo token, excepto cuando ocupan más de una línea.

Esta última observación, se explica de la siguiente forma. Imaginemos que tenemos un texto con una cadena que ocupa varias líneas:

```
cadena_larga = "En un lugar de
la macha, de cuyo nombre no
quiero acordarme ;
```

La parte sombreada corresponde a la extensión de la cadena, que debería considerarse como un solo token. Pero debido a la forma de trabajo de SynEdit (exploración por líneas), un token solo puede extenderse hasta el final de la línea, de modo que para SynEdit, esta cadena la leerá en realidad como 3 tokens.

### 3.1.5 Comentarios

Los comentarios son tokens especiales que están entre dos delimitadores de comentario. Los delimitadores de comentario suelen ser símbolos de uno o dos caracteres.

Pueden haber definidos varios tipos de comentarios, con delimitadores diferentes. Los tipos comunes de comentarios son:

- Comentarios de línea.- So lo ocupan una línea. Se suele especificarlos indicando el delimitador inicial y se supone que se extiende hasta el fin de la línea actual.
- Comentarios de múltiples líneas.- Pueden ocupar varias líneas de texto. Se les suele especificar indicando el delimitador inicial y final.

Algunas observaciones sobre los comentarios son:

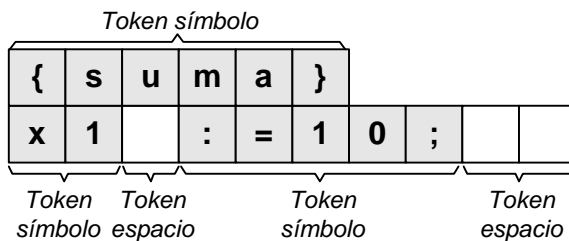
- Los comentarios pueden ser de una o varias líneas.
- Los delimitadores también forman parte del token del comentario.
- Los comentarios de una sola línea pueden contener cualquier carácter en su interior, incluyendo delimitadores de otro tipo de comentario (los cuales se ignoran).
- Los comentarios no se pueden anidar. Un comentario del mismo tipo no puede estar dentro de otro comentario.
- Los comentarios se componen de un solo token, excepto cuando ocupan varias líneas.

### 3.1.6 Símbolos

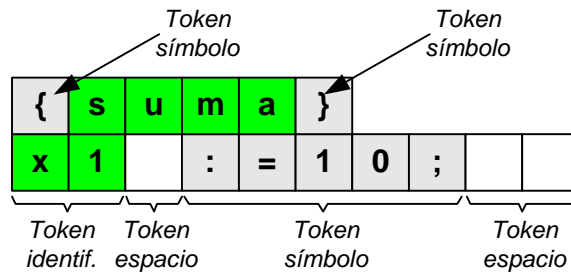
Dentro del resaltador se considera como símbolos a los tokens que están compuestos de todos los caracteres que no sean parte de cualquier otro token (cadenas, comentarios, identificadores, números, etc).

Comúnmente, se espera que los símbolos sean aquellos tokens compuestos de caracteres no alfanuméricos, pero el resaltador podría considerar como símbolos inclusive a caracteres numéricos o alfabéticos. Todo depende de cómo se defina la sintaxis.

Por ejemplo, en el siguiente texto, se ha configurado un lenguaje que solo reconoce espacios (celdas blancas), de forma tal que todos los demás caracteres (en fondo gris claro) se consideran como parte de tokens de tipo símbolo:



En este otro ejemplo se ha configurado un lenguaje que reconoce además a los identificadores (color verde), de forma tal que todos los demás caracteres (en gris claro) se consideran como parte de tokens de tipo símbolo:



Como se aprecia, los tokens símbolo, están formados por todos los caracteres que no sean parte de cualquier otro símbolo. Esta es la forma como trabaja el resaltador.

### 3.2 Identificadores especiales

Los identificadores son elementos fundamentales en la mayoría de sintaxis. Son como las palabras de un lenguaje. Y como suelen ser los elementos más abundantes, se les suele categorizar en la mayoría de lenguajes.

Los identificadores pueden representar a variables, constantes, funciones, directivas, tipos, etc. Por ejemplo, consideremos los siguientes elementos de un lenguaje:

x	-> variable (identificador)
VAL_INI	-> constante (identificador)
TObjeto	-> clase (identificador)
max()	-> función (identificador con paréntesis)

Como vemos, todos estos elementos son identificadores.

TSynFacilSyn, permite manejar diferentes categorías de identificadores, llamados Identificadores Especiales. Básicamente una categoría es un subconjunto de identificadores a los que se le asignará el mismo atributo.

Como ejemplo consideremos un trozo de código en el que existen diversos identificadores (que cumplen la definición de identificadores):

b	e	g	i	n			
x	1	:	=	1	0	;	
c	a	d	:	=	'	'	;
e	n	d	;				

Al cambiar el atributo de los identificadores, se cambiará la apariencia de todos los identificadores del texto. Pero si sabemos que existen identificadores especiales, como las palabras reservadas, entonces, podemos identificar un sub-conjunto de estos identificadores y asociarlos con cualquier otro atributo. Estos vendrían a ser los identificadores especiales.

En nuestro ejemplo de código en Pascal, podríamos elegir los identificadores BEGIN y END; como identificadores especiales y asignarle el atributo de palabras reservadas (KEYWORD).

Los identificadores especiales pueden ser de cualquier tipo predefinido KEYWORD, NUMBER, o algún tipo personalizado.

### 3.3 Símbolos especiales

Así como se pueden definir subconjuntos de identificadores para asignarles un token diferente, también se puede elegir algunas combinaciones de caracteres, de los que forman los símbolos, para definir tokens especiales, que pueden tener cualquier otro atributo.

A estos tokens, definidos a partir de los símbolos, se les llama Símbolos Especiales, y pueden usarse para representar a tokens especiales (como Operadores o Delimitadores), en algún lenguaje de programación.

Como ejemplo consideremos una sintaxis en la que se han definido identificadores (verde), y números (naranja):

{	s	u	m	a	}						
x	1		:	=	1	0	;				

La parte gris representa a los tokens de tipo símbolo. Pero dentro de estos tokens, podemos identificar al operador de asignación “:=”. Entonces, es posible definir el token “:=” como un símbolo especial y asignarle algún otro atributo.

Los símbolos especiales pueden ser del tipo KEYWORD, NUMBER, etc, o de algún tipo personalizado.

### 3.4 Definición de tokens

Puede haber diversas formas de definir un token, pero en TSynFacilSyn, existen solo dos formas para definir un token:

POR CONTENIDO	POR DELIMITADORES
Se definen especificando el delimitador inicial (de uno o varios caracteres), y los caracteres siguientes. En muchos lenguajes, se les suele usar para representar identificadores, o números. NO pueden ser multilínea. Se les llama “tokens por contenido”	Se definen especificando un delimitador inicial (de uno o varios caracteres), y un delimitador final. En muchos lenguajes, se les suele usar para representar cadenas y comentarios. Pueden ser multilínea. Se les llama “tokens delimitados”.

#### 3.4.1 Tokens por contenido

Cuando se define por contenido, se indica siempre el delimitador inicial y un conjunto de caracteres posibles para los siguientes caracteres. Deben seguir estas reglas:

- Que empiecen con el delimitador inicial indicado o que empiecen con un carácter que esté en la lista de “caracteres de inicio válidos”.
- Que contenga caracteres que estén en la “lista de caracteres siguientes válidos”.

La definición por contenido se usa comúnmente para definir a identificadores y números (pero pueden definirse otros tokens por contenido adicionales). Por ejemplo en el archivo de sintaxis, se puede encontrar definiciones como estas:

```
<Identifiers CharsStart= '$A..Za..z_' Content = 'A..Za..z0..9_ '>
</Identifiers>

<Token CharsStart="0..9" Content = "0..9" Attribute='NUMBER'>
</Token>

<Token Start="0x" Content = "0..9a..fA..F" Attribute='NUMBER'>
</Token>
```

A estas definiciones se les reconoce porque incluyen siempre el campo “Content”.

Los tokens por contenido, no pueden ser multi-línea.

### 3.4.2 Tokens delimitados

Cuando se definen tokens por delimitadores, se trabaja con elementos más básicos llamados delimitadores.

La definición por delimitadores es común usarse para definir a cadenas y comentarios, porque estos elementos están siempre delimitados por secuencias fijas de caracteres.

Una definición típica de cadenas es la siguiente:

```
<Token Start='"' End='"' Attribute='STRING'> </Token>
```

A estas definiciones se les reconoce porque incluyen siempre los campos “Start” y “End”.

Los tokens delimitados, pueden ser multi-línea.

## 3.5 Delimitadores de Token

Los delimitadores son un conjunto de caracteres que sirven para determinar dónde empieza y/o donde termina un token. Los tokens por contenido empiezan con un delimitador, mientras que los tokens delimitados, empiezan y terminan con un delimitador.

Un delimitador es a la vez un token completo. Por ejemplo, los siguientes son delimitadores válidos:

```
{  
}  
BEGIN  
/*  
END  
*/  
"""
```

```
<!--  
REM  
#A      (no es recomendable)
```

No se recomienda combinar caracteres de identificadores y símbolos en un identificador, porque puede dar lugar a errores en la interpretación de los delimitadores, por la forma como se optimiza la exploración de la sintaxis en el resaltador.

Dentro del resaltador, se consideran dos tipos básicos de delimitadores:

- Delimitador identificador. Cuando es un identificador especial como BEGIN o END.
- Delimitador de Símbolo. Cuando es un símbolo especial, por ejemplo las llaves: { }.

Los delimitadores tienen la siguiente restricción: Si empiezan con carácter inicial de identificador, entonces todo el delimitador debe ser un identificador. Dicho de otra forma, los siguientes, son delimitadores erróneos:

```
REM%  
A<  
X?X  
INI$   (solo será válido, si se ha definido $ como carácter de identificador)
```

De acuerdo a su ubicación dentro de un bloque, los delimitadores pueden ser:

- De inicio.- Es el delimitador que indica el inicio de un bloque, por ejemplo /\*.
- De fin.- Es el que indica el fin de un bloque, por ejemplo \*/.
- De inicio y fin.- Es cuando el delimitador se usa tanto para el inicio como para el fin de un bloque. Ejemplos pueden ser los delimitadores de cadena.

También se pueden clasificar a los delimitadores de acuerdo a su similitud con otros delimitadores.

- Exclusivos. Cuando no existe otro delimitador en la sintaxis que tenga alguno empiece con los mismos caracteres. Por ejemplo en el lenguaje C, el delimitador de bloque "{", es exclusivo porque el carácter "{" no está incluido en los delimitadores de cadena, ni en los de comentarios //, /\* o \*/.
- Inclusivos. Cuando existe otro delimitador en la sintaxis que empieza con el mismo carácter con el que empieza este delimitador. Este es el caso, por ejemplo del delimitador de



comentario `/**` del lenguaje C, cuyo carácter inicial está incluido en el delimitador de comentario `/*`.

Esta clasificación es importante cuando se requiere optimizar el procesamiento de la sintaxis. Mientras menos delimitadores exclusivos tenga una sintaxis, más lento se hará el procesamiento.

### 3.6 *Bloques de código*

Una característica potente, en el resaltador, es el manejo de bloques de código. Estos nos permiten crear bloques plegables de código, pero no solo eso, sino que nos permiten definir regiones especiales dentro del texto.

Un bloque de texto es una estructura lógica que representa a una extensión de texto que está claramente delimitada por tokens especiales (delimitadores de bloque).

Consideremos por ejemplo el siguiente código en C:

```
void main()
{
    for(int i=2; i<=10; i++)
    {
        fact = fact * i;
    }
}
```

El cuerpo de la función principal `main()`, que está delimitada entre llaves, se puede considerar como un bloque, cuyos delimitadores son las llaves<sup>3</sup>: `{}`.

También podríamos considerar el cuerpo del lazo “for” como otro bloque, también delimitado por llaves. Este sería un caso simple de bloques anidados.

Yendo más lejos, podríamos considerar la parte del “for”, que está entre paréntesis, como otro bloque.

Los bloques pueden ser de una o varias líneas, aunque lo común es que ocupen más de una línea.

Todos estos bloques se pueden definir dentro del resaltador, como bloques válidos. ¿Pero con qué fin?

---

<sup>3</sup> Este bloque es también sintácticamente el cuerpo de la función `main()`, pero un bloque en el resaltador, no necesariamente debe coincidir con estructuras sintácticas del lenguaje, aunque lo ideal sería, que así fuera.

La primera utilidad de los bloques es que pueden ser plegados. Esta sería la principal utilidad, de los bloques de código en el resaltador. No todos los bloques tienen que ser plegables, pero por lo general se usan bloques cuando se quiere que un código sea plegable.

Pero, analizando un poco más la utilidad de los bloques, vemos que podríamos hacerlos similares a bloques sintácticos del lenguaje, de forma que nos permitan hacer un análisis de mayor nivel que la simple extracción de tokens.

Las siguientes reglas se aplican a los bloques:

1. Los bloques pueden ser de una o varias líneas.
2. Los bloques pueden anidarse, pero no traslaparse. Un bloque interior siempre termina antes (o al mismo tiempo) que el bloque que lo contiene.
3. Los delimitadores de bloques son siempre tokens. No se pueden crear bloques si antes sus delimitadores no existen como tokens claramente definidos.
4. Los bloques se definen a partir de tokens. No pueden empezar o terminar en medio de un token. Es decir que siempre contendrán una cantidad entera de tokens. Esta es una consecuencia de la primera regla.
5. Por temas de diseño, el bloque empieza, justo después del delimitador inicial. Es decir no incluye al token inicial.
6. El bloque se extiende hasta el final del delimitador final. Es decir que incluye al delimitador final.
7. Los tokens multilínea, no pueden ser delimitadores de bloques.
8. La creación de bloques no afectan a la estructura de tokens. Un bloque solo agrupa a tokens consecutivos, pero no cambia para nada sus atributos o definiciones<sup>4</sup>. Es decir la apariencia visual de los tokens dentro de un bloque no cambiará para nada con el bloque.

Más adelante se describirá con detalle, la creación de bloques.

### ***3.7 Secciones***

Adicionalmente a los bloques, el resaltador TSynFacilSyn, introduce el concepto de “Secciones”.

---

<sup>4</sup> Se tiene previsto para versiones futuras que los bloques tengan cierto nivel de influencia en los atributos de los tokens que contiene.

Una Sección es un bloque de código que se origina en un delimitador inicial, y termina donde empieza otra sección o donde termina el bloque que lo contiene.

Un ejemplo de secciones son las definiciones de clases de Pascal:

```
class
  private
    fx, fy: integer;
  public
    procedure Iniciar(x,y);
    procedure Mover(dx,dy);
end
```

Diagram illustrating sections in Pascal code:

- A blue bracket labeled "Sección" groups the `private` section (from `private` to `fx, fy: integer;`).
- A blue bracket labeled "Sección" groups the `public` section (from `public` to `procedure Mover(dx,dy);`).

Esta estructura de código, presenta 2 bloques de código claramente definidas, la primera es la que empieza con la etiqueta “private”; y la otra es la que empieza con la etiqueta “public”.

Estos bloques de código son especiales, porque no tienen un delimitador definido. Pueden terminar cuando empieza otro bloque (similar o distinto), o también pueden terminar cuando termina el bloque principal.

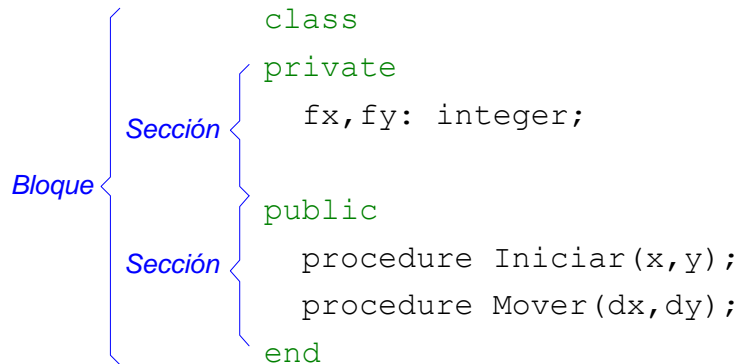
Para manejar este tipo de bloques, el resaltador implementa el concepto de “secciones”. No es conveniente manejarlos como bloques anidados, porque tienen un comportamiento un tanto diferente.

Las mismas reglas que se aplican a los bloques, se aplican también a las “secciones”, excepto que:

1. Las secciones no pueden anidarse. Una sección no puede contener otras secciones.
2. Una sección puede contener bloques.
3. Una sección termina cuando empieza otra sección.
4. Una sección, termina cuando termina el bloque que lo contiene.
5. Si un delimitador es delimitador de sección, y delimitador de bloque. Su característica de delimitador de sección tendrá prioridad sobre su característica de delimitador de bloque.

Una limitación de las secciones, con respecto a los bloques es que las secciones no pueden contener a otras secciones, solo a otros bloques. Un bloque sin embargo puede contener bloques y secciones.

Casi siempre las secciones están encerradas dentro de bloques:



```
class
private
    fx,fy: integer;
public
    procedure Iniciar(x,y);
    procedure Mover(dx,dy);
end
```

The diagram illustrates a code block structure. A large blue bracket on the left, labeled "Bloque", spans the entire code snippet. Inside this block, there are two smaller blue brackets labeled "Sección". The first "Sección" bracket covers the "private" section, which contains the declaration "fx,fy: integer;". The second "Sección" bracket covers the "public" section, which contains two procedure declarations: "procedure Iniciar(x,y);" and "procedure Mover(dx,dy);". The code snippet ends with the keyword "end".

Por definición las secciones, terminan antes de que empiece otra sección o cuando termina el bloque que lo contiene. Así, la sección "public", del ejemplo no incluye al token "end", que sin embargo, sí es parte del bloque.

A pesar del tratamiento especial que hay, las secciones son básicamente, bloques. Internamente comparten la misma estructura de datos, y solo los diferencia una bandera. Así que cuando hablemos de secciones, frecuentemente nos referiremos a ellos como bloques.

Más adelante se describirá con detalle, la creación de secciones.

### 3.8 Niveles léxico, sintáctico y semántico

En el análisis de lenguajes, con fines de procesamiento por intérpretes o compiladores, se suele separar los niveles de detalle. Generalmente, los niveles que se suelen aceptar son:

- Nivel Léxico.- Referido a la identificación y extracción de tokens de un texto, a partir de los caracteres. Se usa básicamente como el nivel base, para el análisis sintáctico.
- Nivel Sintáctico.- Usa los tokens para formar estructuras de acuerdo a reglas predefinidas (Gramática). Se suelen identificar elementos más complejos como Operando, Variable, Constante, Expresiones, etc.

- Nivel Semántico.- Se basa en reglas de mayor nivel aplicadas al lenguaje. Da un sentido a las expresiones o sentencias del código. Un código puede tener validez sintáctica, pero ser semánticamente errónea.

Al hablar de un resaltador, solemos entenderlo como una máquina para identificar tokens y para asignarles atributos de color, fondo, tipo de letra, etc.

El resaltador TSynFacilSyn, ha sido diseñado para ser una eficiente máquina extractora de tokens. En este sentido es también lo que se llama un Analizador léxico o “Lexer”.

Pero TSynFacilSyn, va más allá, porque permite el manejo de bloques de código que se mueven en el nivel sintáctico.

Si solo usamos el resaltador, para extraer tokens, estamos definitivamente moviéndonos en el nivel léxico. Pero manejar bloques de código, nos permite un acercamiento mayor al nivel sintáctico.

Por lo tanto, al definir los bloques de nuestro lenguaje, debemos tener en mente, no solo las facilidades de plegado que queremos otorgarle al texto, sino también que estas, deben estar de acuerdo a la sintaxis del lenguaje con el cual estamos trabajando.

## 4 Configurando la Sintaxis con archivo externo.

El archivo de sintaxis es un simple archivo XML, que define la sintaxis a usar para el resaltador. Se puede editar a mano o con una herramienta de software.

Es necesario asegurarse de que el archivo de sintaxis en XML, no presente ningún error de estructura (bloques sin cerrar, caracteres inválidos, etc.), de otra forma el programa generaría error. De todas formas el resaltador hace una verificación previa de la estructura del XML, antes de cargarlo, y dará un mensaje de error si encuentra alguna inconsistencia.

Si hubiera algún error, con la estructura del archivo, el resaltador mostrará un mensaje en pantalla y detendrá la carga.

Una rutina típica para cargar una sintaxis por código sería:

```
uses ... , SynHighlighterFacil;  
  
var  
    hlt : TSynFacilSyn;  
  
...  
    hlt := TSynFacilSyn.Create(self); //crea resaltador  
    editor.Highlighter := hlt;  
    hlt.LoadFromFile('SynPHP.xml'); //carga archivo de sintaxis  
    editor.Invalidate;  
...  
    hlt.Free; //libera
```

La llamada a Invalidate(), no es necesaria, si solo se va a cargar una vez el archivo de sintaxis. Sin embargo, si es que se va a cambiar varias veces la sintaxis de un editor, es necesario llamar a invalidate(), para que el contenido se adecúe a la nueva sintaxis.

### 4.1 El archivo de sintaxis

La estructura general de un archivo de sintaxis es:

<Language name="nombre de lenguaje" ext="lista de extensiones">

Definición de atributos.

Definición de identificadores e identificadores especiales.

Definición de símbolos y símbolos especiales.

Definición de números.

Definición de cadenas

Definición de comentarios

Definición de tokens por contenido.

Definición de tokens delimitados.

Definición de bloques y secciones.

</Keywords>

No es necesario usar todas las definiciones para construir una sintaxis. Se pueden prescindir de la mayoría, de las configuraciones y algunas son tomadas por defecto.

Por ejemplo una sintaxis sencilla para PHP, podría ser:

```
<Language name="php" ext="php">
  <Keywords>
array and break case class const continue default die do double echo
else elseif empty endfor endif endswitch endwhile eval exit extends
false float for function global if include int integer isset list
new object old_function or print real require return show_source
static string switch true unset var while xor
  </Keywords>
</Language>
```

En este caso, no se especifica la definición de identificadores, pero se asume una definición por defecto.

Este ejemplo solo indica las palabras reservadas en el bloque <Keywords>. La lista de palabras indicadas puede especificarse en una o varias líneas, siempre que se use solo espacios o saltos de línea para separar las palabras.

### 4.1.1 Caracteres especiales

Como todo archivo XML, el archivo de sintaxis requiere tener en cuenta algunas consideraciones en cuanto a los caracteres que se pueden incluir.

Para los caracteres especiales, en XML se usan:



SÍMBOLO	DESCRIPCIÓN	CODIFICACIÓN EN XML
&	Y comercial	&amp;
<	Abrir corchete angular	&lt;
>	Cerrar corchete angular	&gt;
“	Comilla recta	&quot;
‘	Apóstrofo	&apos;

Por ejemplo la siguiente declaración:

```
Start="<"
```

No es una forma válida para definir un campo en XML. Se debe usar en su lugar:

```
Start = "&lt;,"
```

También se puede usar la forma:

```
&#<código ASCII del caracter>;
```

Así para representar al carácter de tabulación, escribiremos: &#009

#### 4.1.2 Uso de intervalos y comodines

En el archivo de sintaxis, es común el uso de intervalos para evitar tener que escribir una lista larga de caracteres.

Los siguientes intervalos son ejemplos de formas simplificadas de escritura:

```
A..Z = ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
a..z = abcdefghijklmnopqrstuvwxyz
```

```
0..9 = 0123456789
```

```
a..f = abcdef
```

Por ejemplo si se quiere simplificar esta declaración:

```
Start = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"
```

Se puede usar la forma corta:

Start = "A..Za..z"

Existe adicionalmente, dos comodines predefinidos:

%HIGH%        -> Caracteres ASCII del #128 al #255 (no implementado)

%ALL%         -> Todos los caracteres, excepto #0.

El manejo de intervalos y comodines, no forman parte del estándar de XML, sino que son interpretados por el propio resaltador.

### 4.1.3 Uso de formas simplificadas

La mayoría de las definiciones en el archivo de sintaxis, serán de tipo:

```
<token attribute="NOMBRE_ATRIBUTO" valor1="XXX" valor2="YYY" ... >  
</token>
```

Para evitar tener que escribir demasiado texto, se permite la forma simplificada:

```
<NOMBRE_ATRIBUTO valor1="XXX" valor2="YYY" ... >  
</NOMBRE_ATRIBUTO>
```

Es decir que se usa el nombre del atributo como etiqueta o marcador, y de esa forma, ya no es necesario especificar el atributo.

Otra de las formas simplificadas, es cuando no se quiere cambiar los atributos de una etiqueta. Si por ejemplo una definición tiene la forma:

```
<ETIQUETA valor1="XXX" valor2="YYY" >  
</ETIQUETA>
```

Si escribimos la definición en su forma simplificada:

```
<ETIQUETA>  
</ETIQUETA>
```

Estamos indicando que definimos explícitamente la etiqueta, pero que no deseamos cambiar ninguno de sus valores. Esta forma es útil cuando se requiere la etiqueta para crear otras definiciones o para expresar listas de valores (Ver 4.5.1 - Identificadores Especiales).

#### 4.1.4 Uso de espacios, mayúsculas y minúsculas.

Los espacios en blanco, así como los saltos de línea se ignorarán en la definición de un lenguaje. Por ejemplo, las siguientes definiciones son equivalentes:

```
<ETIQUETA valor1="XXX" valor2="YYY" <</ETIQUETA>
```

```
<ETIQUETA valor1="XXX" valor2="YYY" >  
</ETIQUETA>
```

```
<ETIQUETA valor1="XXX"  
valor2="YYY" >  
</ETIQUETA>
```

A pesar de que XML es sensible a la caja por definición, el resaltador ignorará el tipo de caja de las definiciones. Así por ejemplo, las siguientes definiciones son similares.

```
<etiqueta Valor1="XXX" Valor2="YYY" >  
</etiqueta>
```

```
<Etiqueta valor1="XXX" valor2="YYY" >  
</Etiqueta >
```

```
<ETIQUETA VALOR1="XXX" VALOR2="YYY" >  
</ETIQUETA>
```

Sin embargo, por definición de XML, la etiqueta de inicio y fin deben coincidir en caja. La siguiente declaración es un caso de error:

```
<Etiqueta Valor1="XXX" Valor2="YYY" >  
</etiqueta>
```

### 4.1.5 Comentarios en XML

Los comentarios en XML, se definen encerrados entre los delimitadores `<!--y -->` . Por ejemplo:

```
<?xml version="1.0"?>
<!--
Este es un comentario
-->
```

Tener cuidado con la combinación de caracteres: `-->` o `--">` porque se consideran como delimitador de comentario en XML.

## 4.2 Definición del Lenguaje

La etiqueta principal <Language> tiene los siguientes atributos:

Name	Obligatorio	Nombre de la sintaxis
Ext	Opcional	Lista de extensiones de archivos asociados al lenguaje.
CaseSensitive	Opcional	Indica si se aplicará diferenciación entre mayúscula y minúscula, para ubicar identificadores y símbolos especiales.
ColorBlock	Opcional	Permite activar el coloreado de bloques. Por defecto está en "None". (Desactivado).

Solo puede haber una definición de lenguaje por archivo.

Una definición mínima de lenguaje típica sería:

```
<Language name="pascal">
</Language>
```

Otra definición de lenguaje podría ser:

```
<Language name="php" ext="php php3 phtml">
</Language>
```

Ambos ejemplos son definiciones vacías, pero son el marco en donde deben crearse todas las definiciones del lenguaje.

Si no se especifica el atributo "CaseSensitive", se asume que está en falso, es decir que no habrá distinción entre mayúsculas y minúsculas. Para activarlos, se le debe asignar la cadena "TRUE":

```
<Language name="pascal" ext="pas pp" CaseSensitive="TRUE">
</Language>
```

El parámetro "ColorBlock", permite mostrar los bloques con un color de fondo distinto de acuerdo al valor que contenga. Los valores que puede tomar son:

"None"	Desactiva el coloreado de bloques. Es la opción por defecto.
"Level"	Colorea el fondo del bloque, por nivel. Muestra los bloques con un color más o menos oscuro, de acuerdo al nivel del bloque o sección.
"Block"	Colorea el fondo del bloque, usando el color definido para cada bloque, en su parámetro "BackCol". (Ver Sección 4.10 - Definición de Bloques)

La siguiente definición es un ejemplo, de cómo activar el coloreado de bloques por nivel:

```
<Language name="php" ext="php php3 phtml" ColorBlock="Level">
</Language>
```

Para más información sobre el coloreado de bloques, ver la Sección 4.10 - Definición de Bloques.

### 4.3 Configurando los atributos

Los atributos de los tokens, se pueden configurar o crear desde dentro del archivo XML, así se puede definir completamente la apariencia que tendrán cada uno de las categoría de tokens dentro del resaltador.

Para configurar un atributo, se usa la etiqueta <ATTRIBUTE>. Así por ejemplo, para cambiar la apariencia de todos los identificadores de un texto, se puede usar la siguiente definición:

```
<Attribute Name="Identifier" ForeCol="#800000" Bold="True">
</Attribute>
```

Esta definición pondrá a todos los identificadores, con color de texto rojo y en negrita. Los parámetros de color, deben seguir el formato de 3 componentes: #RRGGBB, donde RR, GG y BB, corresponden a los valores de los componentes Rojo, Verde y Azul, expresados en dos dígitos hexadecimales.

Si el nombre del atributo ya existe, se actualiza sus propiedades. Si el atributo especificado no existe, se crea primero el atributo.

La etiqueta <ATTRIBUTE> tiene los siguientes campos:

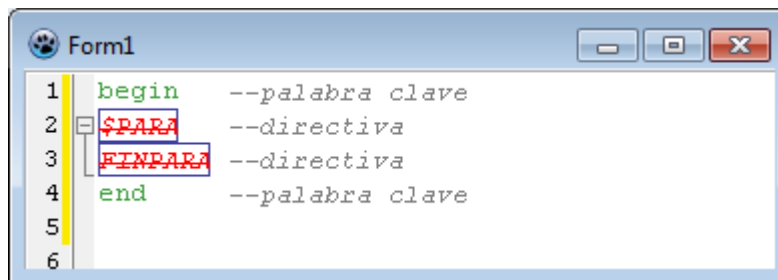
Name	Obligatorio	Indica el nombre del atributo que se desea configurar.
------	-------------	--

BackCol	Opcional	Es el color de fondo del token
ForeCol	Opcional	Es el color del texto del token
FrameCol	Opcional	Es el color del borde del token
Bold	Opcional	Valor booleano. Pone el texto en negrita
Italic	Opcional	Valor booleano. Pone el texto en cursiva
Underline	Opcional	Valor booleano. Pone el texto con subrayado
StrikeOut	Opcional	Valor booleano. Pone el texto tachado

Se pueden expresar múltiples definiciones de atributos en un archivo de sintaxis:

```
<?xml version="1.0"?>
<Language name="SQL Oracle" ext="psql" ColorBlock="Block">
  <Attribute Name="Keyword" ForeCol="#40a040" Bold="False">
  </Attribute>
  <Attribute Name="Directive" FrameCol="#4040a0" Italic="true" StrikeOut="true">
  </Attribute>
  ...
</Language>
```

Estas definiciones de atributos podrían tener el siguiente resultado:



Si no se especifican las propiedades de los atributos, se mantienen los que tienen por defecto.

La mayoría de atributos tienen sus propiedades por defecto desactivadas, es decir, no se aplican colores ni para el texto, ni para el fondo, ni para el borde. Además todos los atributos de tipo booleano están en falso.

Sin embargo, algunos atributos, tienen propiedades establecidas por defecto como las palabras claves y los comentarios. Así las cadenas se mostrarán inicialmente con texto azul, los comentarios en letra cursiva y en color gris, mientras que las palabras claves se mostrarán en verde.

Un atributo nuevo, se crea de la misma forma a como se configura un atributo ya existente. En la siguiente definición, creamos el atributo “Especiales” y configuramos dos identificadores especiales con el atributo “Especiales”:

```
<?xml version="1.0"?>
<Language name="Pascal" ext="pas inc pp" ColorBlock="Level">
  <Attribute Name="Especiales" ForeCol="#008000" Bold="True"> </Attribute>
  <Identifiers CharsStart= "A..Za..z_" Content = "A..Za..z0..9_">
    <Especiales>
      fulano mengano
    </Especiales>
  </Identifiers>
</Language>
```

El nombre del atributo se especifica en la propiedad “Name=”. Puede tener cualquier longitud y contener casi cualquier carácter, pero se recomienda que sea un nombre corto, para mejorar la velocidad de las búsquedas.

El nombre de un atributo nuevo, no debe ser igual a alguno de los atributos pre-definidos en el resaltador:

1. EOL
2. SYMBOL
3. SPACE
4. IDENTIFIER
5. NUMBER
6. KEYWORD
7. STRING
8. COMMENT

No existe un límite práctico para la cantidad de atributos que se pueden crear. Solo está limitado por la cantidad de memoria disponible en el sistema.



## 4.4 Definición de Símbolos

Dentro del resaltador, los tokens que nunca faltarán, serán los símbolos. Esto es debido a la forma como trabaja el resaltador internamente.

Los símbolos son los primeros elementos que se deben definir en una sintaxis. Sin embargo, el resaltador hace una definición por defecto, que sería algo como esta:

```
<Symbols Content = "%ALL%" >
</Symbols>
```

Y es la primera definición que se aplica, ya que se espera, que las definiciones siguientes vayan restringiendo el alcance de los símbolos.

En realidad, la definición de símbolos, excluye a los caracteres blancos #1..#31 y al carácter nulo #0, por fines de seguridad.

No es necesario definir explícitamente a los símbolos, y tampoco se puede hacer. El resaltador aceptará la definición de símbolos, pero no la procesará. Solo aceptará esta definición, para permitir definir símbolos especiales. Así una sintaxis típica de símbolos especiales, podría ser:

```
<Symbols>
  <Token Attribute = "OPERATOR">
    + - * /
  </Token>
</Symbols>
```

### 4.4.1 Símbolos Especiales

Los símbolos especiales se definen dentro de la etiqueta SYMBOLS:

```
<Symbols>
  <Operator>
    lista de símbolos
  </Operator>

  <Delimiter>
    lista de símbolos
  </Delimiter>
```

```
...  
</Symbols>
```

Cada una de estos grupos pueden tener un atributo diferente, si es que se le define de forma separada.

Un uso común para los Símbolos especiales, es para definir Operadores:

```
<Extra1>  
+ - * / =  
</Extra1>
```

Aquí se usa el atributo EXTRA1, porque no se tiene un atributo especial para operadores, pero en teoría se podría usar cualquier atributo que no se esté usando en la sintaxis.

No es necesario, poner los símbolos en un orden específico. Basta con ponerlos separadas por espacios o saltos de línea. Esta definición tiene como limitación que no permite incluir el carácter espacio, como parte de un símbolo.

Los símbolos, comúnmente no contienen caracteres alfabéticos, pero si los tuviera, la comparación se hará considerando la opción CaseSensitive.

Todos los símbolos especiales definidos, deben ser estrictamente símbolos, es decir que deben cumplir con la definición de los símbolos. Por ejemplo no se podrá definir “\$\$” como un símbolo especial, si es que el carácter \$ ya ha sido definido como carácter inicial válido, por ejemplo para los identificadores.

No es recomendable tampoco incluir un número grande de símbolos especiales, porque se puede obtener un deterioro en el rendimiento del resaltador. La búsqueda de símbolos no está optimizada como la búsqueda de identificadores especiales.

Es importante que estas listas de símbolos especiales, sean exclusivas, de otra forma, se generará un error en la carga.

Al igual que en la definición de identificadores especiales, también se puede usar el parámetro “TokPos”, para indicar el orden en que se debe encontrar el token para ser considerado como token especial.

En la siguiente definición, condicionaremos a que el símbolo “/”, sea considerado como token especial, solo cuando aparezca como el primer token de la línea.

```
<Symbols>
  <Extra1 TokPos="1">
    /
  </Extra1>
  . . .
</Symbols>
```

## 4.5 Definición de Identificadores

Los identificadores del resaltador, se definen por contenido, indicando el carácter inicial y los siguientes caracteres.

Se define con la etiqueta <Identifiers>.

Tiene los siguientes atributos:

CharsStart	Obligatorio	Lista de caracteres considerados válidos como carácter inicial de un identificador.
Content	Obligatorio	Lista de caracteres considerados válidos como caracteres siguientes de un identificador.

Por ejemplo una definición típica sería:

```
<Identifiers CharsStart= "A..Za..z" Content = "A..Za..z0..9_">
</Identifiers>
```

Los caracteres iniciales y siguientes pueden ser cualquier carácter imprimible.

Es importante definir correctamente los caracteres para un identificador, porque los demás elementos de la sintaxis, como los delimitadores, dependen de esta definición.

Los caracteres a usar, como caracteres iniciales válidos para los identificadores, no deben entrar en conflicto con la definición otros tokens por contenido (como números). De darse el caso, se usará la última definición encontrada, o la aplicada por defecto.

Si no se especifica la definición de identificadores se tomará una definición por defecto. Esta es:

```
<Identifiers CharsStart= "A..Za..z$_" Content = "A..Za..z0..9_">
</Identifiers>
```

Si por algún motivo, se quisiera eliminar la definición de identificadores por defecto, se deberá escribir la siguiente línea:

```
<Identifiers CharsStart= "" Content = "">
```

```
</Identifiers>
```

Los identificadores aceptan la definición de subconjuntos de tokens (Ver 4.5.1 - Identificadores Especiales).

No se debe confundir la definición de identificadores (que son elementos especiales en el resaltador), con la forma simplificada de tokens por contenido (Ver sección siguiente).

### 4.5.1 Identificadores Especiales

Los identificadores especiales se definen dentro de la etiqueta IDENTIFIERS:

```
<Identifiers>
  <Keyword>
    lista de identificadores
  </Keyword>

  <Variable>
    lista de identificadores
  </Variable>

  ...

</Identifiers>
```

Cada una de estos grupos pueden tener un atributo diferente, si es que se le define de forma separada.

Las palabras reservadas se identifican como “Keyword”, en el archivo de sintaxis:

```
<Keyword>
  lista de palabras claves
</Keyword>
```

Estas palabras reservadas se definen para cada lenguaje. Por ejemplo para Pascal se podría definir:

```
<Keyword>
  absolute abstract and array as asm assembler begin break case class const
  constructor continue default destructor dispose div do downto else end exit
  false far finalization finally for forward function goto if implementation in
```

```
index inherited initialization inline interface is label mod name new nil not  
object of on operator or override private procedure program property protected  
public published read record register repeat saveregisters self set string then  
to true try type unit until uses var virtual while with xor  
</Keyword>
```

No es necesario, poner las palabras en un orden específico. Basta con ponerlas separadas por espacios o saltos de línea.

Por cuestiones de optimización, los identificadores especiales solo pueden tener como carácter inicial, a los caracteres alfabéticos en inglés, y los caracteres `_`, `$`, `%`, `@` y `&`. Si el identificador especial, no empieza con estos caracteres, no se podrá definirlo (a menos que se modifique al resaltador).

Por ejemplo no se podrá agregar el identificador especial `"#uno"`, aun cuando el carácter `"#"`, sea un carácter inicial válido para los identificadores.

Si se usa la opción `CaseSensitive` en `FALSE` (opción por defecto), será indiferente si se ponen las palabras en mayúscula o minúscula, de otra forma se leerá la lista considerando la caja.

Todos los identificadores especiales definidos, deben ser estrictamente identificadores, es decir que deben cumplir con la definición de los identificadores. Por ejemplo no se podrá definir `"$INCLUDE"` como un identificador especial, si es que el carácter `$`, no forma parte de los caracteres iniciales válidos para un identificador.

No es recomendable tampoco incluir un número grande de identificadores especiales, porque a pesar de estar optimizada la búsqueda de identificadores especiales, se puede obtener un deterioro en el rendimiento del resaltador.

Es importante que estas listas de identificadores especiales, sean exclusivas, de otra forma, se generará un error en la carga.

Si bien el orden en que se declaran los identificadores especiales, en el archivo XML, no afecta a la sintaxis, si puede afectar el rendimiento del resaltador. Para un procesamiento más rápido, se recomienda poner primero las palabras que en el lenguaje tengan más probabilidad de aparecer, respetando o no el orden alfabético del carácter inicial (internamente se ordenarán por el carácter inicial).

Así por ejemplo, si en un lenguaje determinado, la palabra clave END, aparece con más frecuencia que la palabra ECHO, su declaración debería ser:

```
<Keyword>
    ...
end echo
    ...
</Keyword>
```

Existe una forma alternativa para declarar los identificadores especiales, desde fuera de la etiqueta <IDENTIFIERS>:

```
<Identifiers>
    ...

</Identifiers>

<Keyword>
    lista de identificadores
</Keyword>
```

El resaltador asumirá automáticamente que estas definiciones de palabras especiales pertenecen a la etiqueta <IDENTIFIERS>. Sin embargo, esta forma solo es válida para los atributos KEYWORD.

Es posible condicionar un identificador especial, usando su posición dentro de una línea. El parámetro "TokPos", indica el orden en que se debe encontrar el token para ser considerado como token especial.

En la siguiente definición, condicionaremos a que el identificador "INICIO", sea considerado como token especial, solo cuando aparezca como el primer token de la línea.

```
<Identifiers>
    <Keyword TokPos="1">
        inicio
    </Keyword>

</Identifiers>
```

La diferencia está, en que ahora se usa el parámetro "TokPos", para indicar que el identificador especial, debe estar al inicio de la línea (token número 1), para ser considerado como

tal. En cualquier otra posición, en que se encuentre el token, no será considerado como identificador especial.

De la misma forma se puede indicar que el token ocupe la posición 2, 3 o cualquier otra. El primer token que aparezca en una línea será siempre el número 1, y así sucesivamente. No olvidar que, inclusive los espacios son también un token.

Por defecto, el valor de “TokPos”, es cero, lo que significa que no se considera el orden que ocupe el token. Pero al definir “TokPos”, se le está dando un parámetro más, de modo que, puede haber en el resaltador, dos identificadores especiales con la misma cadena pero con distinta posición de token. Es decir, la siguiente definición no generará error:

```
<Identifiers>
  <Keyword TokPos="1">
    inicio
  </Keyword>
  <Keyword TokPos="3">
    inicio
  </Keyword>
</Identifiers>
```

Sin embargo, la siguiente definición si dará error, por cuanto hay un cruce en las definiciones:

```
<Identifiers>
  <Keyword TokPos="1">
    inicio
  </Keyword>
  <Keyword>
    inicio
  </Keyword>
</Identifiers>
```

## 4.6 Definición de Tokens por contenido

Adicionalmente a los identificadores, TSynFacilSyn, admite la declaración de otros tokens por contenido.



Se definen con la etiqueta <Token ... Content = ... >.

Tiene los siguientes parámetros válidos:

CharsStart	Opcional	Lista de caracteres considerados válidos como carácter inicial.
Start	Opcional	Cadena que se usará como delimitador inicial del token.
Content	Obligatorio	Lista de caracteres considerados válidos como caracteres siguientes de un identificador.
CharsEnd	Opcional	Lista de caracteres considerados válidos como carácter final.
Attribute	Opcional	Indica el atributo a usar para el token. Si no se especifica, se asume el atributo NULL.

La definición de tokens por contenido, puede tomar dos formas:

```
<Token CharsStart= ... Content = ... > </Token>
```

```
<Token Start= ... Content = ... > </Token>
```

El parámetro “Content” es obligatorio, pero los parámetros “CharsStart” y “Start”, se pueden intercambiar.

Usamos “CharsStart”, cuando queremos indicar que un carácter (tomado de una lista) es suficiente para determinar el inicio de un token delimitado.

Por ejemplo una definición de este tipo sería:

```
<Token CharsStart="0..9" Content = '0..9' Attribute='NUMBER'>
</Token>
```

“CharStart” indicará la lista de caracteres válidos que determinan el inicio de un token delimitado.

Tanto “CharsStart”, “Content” y “CharsEnd” pueden incluir cualquier carácter imprimible, incluyendo caracteres alfabéticos.

Cuando se quiere definir tokens por contenido que empiezan con una cadena de más de un carácter se debe usar la forma alternativa:

```
<Token Start="0x" Content = '0..9A..Ha.h' Attribute='STRING'> </Token>
```

Este ejemplo se puede usar para identificar a los números hexadecimales en lenguaje C.

El parámetro “Start”, en este caso es un delimitador de token y por lo tanto tiene las restricciones indicadas en la sección 3.5 - Delimitadores de Token.

Si no se especifica el campo “Attribute”, el nuevo tipo de token creado, no presentará atributos de resaltado.

Solo se pueden definir hasta 4 tokens por contenido, adicionales a los identificadores. Esta limitación se impone por criterios de velocidad.

Los caracteres a usar, como caracteres iniciales válidos, no deben entrar en conflicto con la definición de otros tokens por contenido (como identificadores o números). De darse el caso, se usará la última definición encontrada.

Si se va a definir tokens por contenido, es altamente recomendable definir también a los identificadores, ya que si no se encuentra la definición de identificadores, se asumirá al final, una definición por defecto, la cual puede sobrescribir la definición de los otros tokens por contenido.

Los tokens por contenido se suele usar para definir elementos como las constantes numéricas. Una definición típica de números sería:

```
<Token CharsStart=".0..9" Content = '0..9' Attribute='NUMBER'>  
</Token>
```

También se suelen usar tokens por contenido para definir elementos que no se pueden manejar como identificadores, porque no cumple con esa definición. Por ejemplo las cadenas en Object Pascal que se expresan como #32 o #9, se pueden expresar como tokens por contenido que empiezan con el carácter # y le siguen dígitos numéricos:

```
<Token CharsStart="#" Content = '0..9' Attribute="STRING">  
</Token>
```

El atributo “CharsEnd”, no suele usarse, pero permite definir la lista de caracteres que se permiten al final de un token por contenido.

Un uso típico, es cuando se definen números que incluyan el punto decimal, pero no queremos que considere el punto decimal si es que aparece al final. Entonces la definición sería:

```
<Token CharsStart="0..9" Content = '0..9.' CharsEnd= '0..9' Attribute='NUMBER'>
</Token>
```

Con esta definición, solo se reconocerá como parte del token, el punto decimal, solo si es que existe al menos un dígito numérico después, o dicho de otra forma. No se permitirá al carácter “.” Al final del token.

El uso de “CharsEnd”, genera una carga adicional en el procesamiento de tokens, así que debe evitarse en lo posible.

Se puede simplificar la definición, usando la forma simplificada:

```
<Number CharsStart="0..9" Content = '0..9.' CharsEnd= '0..9'>
</Number>
```

Pero no es recomendable usar formas simplificadas, porque pueden dar lugar a confusión con otras definiciones del lenguaje.

#### 4.6.1 Tokens por contenido e identificadores

La definición simplificada de tokens por contenido, puede dar lugar a confusión con la definición de identificadores.

Notar que la forma simplificada:

```
<Identifier CharsStart= "A..Za..z" Content = "A..Za..z0..9_">
</Identifier>
```

Es diferente a la definición de Identificadores:

```
<Identifiers CharsStart= "A..Za..z" Content = "A..Za..z0..9_">
</Identifiers>
```

La primera es una forma simplificada de:

```
<Token CharsStart= "A..Za..z" Content = "A..Za..z0..9_" Attribute='IDENTIFIER'>
```

**</Token>**

Pero la segunda es una definición especial para los identificadores, que son tokens por contenido, pero que tienen un tratamiento especial en el resaltador.

La definición con <Identifiers> indica al resaltador que estamos definiendo a los identificadores del resaltador. Estos identificadores permiten definir subconjuntos de tokens para asignarle otro atributo (Ver Sección de Tokens Especiales), mientras que la forma <Identifier>, es solo la forma simplificada de tokens por contenido con atributo IDENTIFIER, pero esta definición no admite definir subconjuntos de tokens.

## 4.7 Definición de Tokens delimitados

Los tokens delimitados, son otra forma de definir tokens, alternativamente a la definición de tokens por contenido.

Se definen con la etiqueta <Tokens Start= ... >.

Tiene los siguientes atributos:

Start	Obligatorio	Delimitador inicial del token. Puede ser un símbolo o identificador.
End	Opcional	Delimitador final del token. Puede ser un símbolo o identificador.
Attribute	Opcional	Indica el atributo a usar para el token. Si no se especifica, se asume el atributo NULL.
Multiline	Opcional	Indica si el token puede extenderse por varias líneas o si es de una sola línea. Por defecto es de una sola línea.
Folding	Opcional	Indica si se mostrará la marca de plegado en el editor. Solo es válido para los tokens multilínea. Por defecto, está desactivado.

Un caso típico de token delimitado, es la definición de cadenas:

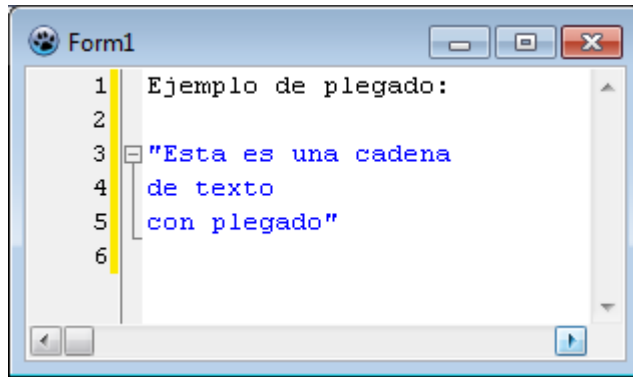
```
<Token Start="'" End="'" Attribute='STRING'>
</Token>
```

Esta sentencia asignará el atributo STRING a la secuencia de caracteres, que se encuentre entre comillas simples, incluyendo a las comillas.

Si quisiéramos que las cadenas se puedan extender por más de una línea y además se puedan plegar, el siguiente código funcionará:

```
<Token Start="'" End="'" Attribute="STRING" Multiline="true" Folding="true">
</Token>
```

Un texto, con este resaltador, se podría mostrar así:



El siguiente ejemplo permite definir las directivas de compilador de Pascal:

```
<Token Start="{ $" End="}" attribute="DIRECTIVE">
</Token>
```

Esta definición permitiría identificar a los bloques de tipo:

{ \$!+ }

Este otro ejemplo, permite otro tipo de directiva, usada en C:

```
<Token Start="#" End="&gt;" attribute="DIRECTIVE">
</Token>
```

Esta definición permitiría identificar a los bloques de tipo:

#include <windows.h>

Si no se especifica el delimitador final de un token delimitado, o se le asigna la cadena vacía, se asume que se extiende hasta el fin de la línea actual. Este comportamiento es común en la definición de comentarios de una sola línea:

```
<Token Start="//" attribute="COMMENT">
</Token>
```

Los atributos que pueden ser asignados son, los todos los que maneja el resaltador:

1. EOL
2. SYMBOL

3. SPACE
4. IDENTIFIER
5. NUMBER
6. KEYWORD
7. STRING
8. COMMENT

Se pueden definir tantos tokens delimitados como se quiera, considerando que a mayor cantidad de ellos, es mayor el procesamiento que debe hacer el resaltador.

Los delimitadores para un token son por lo general de tipo símbolo, pero pueden ser también de tipo identificador (Ver Sección 3.5 - Delimitadores).

Las cadenas y comentarios son ejemplos clásicos de tokens delimitados.

## 4.8 Definición de cadenas

Las cadenas son tokens delimitados comunes. Los delimitadores son por lo general, la comilla y doble comilla, pero dependerá del lenguaje a usar.

Para definir a las cadenas se puede usar la forma simplificada de definición de tokens:

```
<String Start="&quot;;" End="&quot;;">
</String>
```

Si no se especifica si las cadenas son de múltiples líneas, se asume que son cadenas de una sola línea. Para definir cadenas de múltiples líneas se debe especificar:

```
<String Start="&quot;;" End="&quot;;" MultiLine="True">
</String>
```

Los delimitadores de cadena, no pueden ser a la vez delimitadores de comentarios.

## 4.9 Definición de comentarios

Los comentarios son también tokens delimitados comunes, al igual que las cadenas. Pueden ser de una línea o de múltiples líneas:

Si no se especifica el delimitador de fin, se asume que es el salto de línea, es decir que es un comentario de línea.

Por ejemplo, para definir comentarios de una sola línea con el carácter #, haríamos:

```
<Comment Start="#" >  
</Comment >
```

Para definir comentarios de múltiples líneas se debe especificar el delimitador de fin y el atributo "Multiline":

```
<Comment Start="/*" End="*/" Multiline="true">  
</Comment >
```

Si no se especifica la definición de comentarios, no se crean definiciones por defecto, o dicho de otra forma, no se reconocerán comentarios.

Los delimitadores de comentarios, no pueden ser a la vez delimitadores de cadenas.

## 4.10 Definición de Bloques

Los bloques se definen con las etiquetas <Block> </Block>.

Tiene los siguientes parámetros:

Start	Opcional	Delimitador inicial del bloque. Debe ser un identificador o símbolo especial.
End	Opcional	Delimitador final del bloque. Debe ser un identificador o símbolo especial.
Name	Opcional	Es una cadena que identificará al bloque dentro del archivo de sintaxis. Es el nombre del bloque. Debe ser único para cada bloque.
Folding	Opcional	Indica si se mostrará la marca de plegado en el editor. Esta marca solo aparecerá cuando el bloque tenga más de una línea. Por defecto, está activado.
Parent	Opcional	Indica el bloque padre del bloque. Por defecto es "NONE", es decir que no está restringido a algún bloque.



BackCol	Opcional	Indica el color de fondo con que se pintarán, todos los tokens que se encuentren dentro del bloque.
---------	----------	---

Los parámetros “Start” y “End”, deberían ser en realidad, obligatorios, pero se consideran opcionales, porque se pueden definir posteriormente usando una definición alternativa (Ver Sección 4.11 - Bloques avanzados).

Un detalle saltante, es que no existe el parámetro de atributo, porque los bloques no poseen atributos visibles. El parámetro “BackCol”, permite realizar colorear el fondo de los tokens que pertenezcan a un bloque, pero este color solo tendrá efecto cuando se activa la opción ColorBlock="Block", en la definición del lenguaje.

Un caso típico de bloque es el que se usa en Pascal:

```
<Block Start="Begin" End="End" >  
</Block>
```

Este bloque se extenderá desde después del token “BEGIN” hasta el fin el token “END”. El bloque puede distribuirse en una o más líneas. No es necesario especificarlo. Los bloques son por defecto multilínea.

La propiedad “Name”, permite especificar un nombre para el bloque. Si no especificamos un nombre para el bloque, el resaltador, le asignará un nombre interno, que tiene la forma “Blk1”, “Blk2”,..., dependiendo de cuantos bloques existan.

Se le puede asignar cualquier nombre a un bloque, excepto “Main”, “MultiToken” y “None”, que son nombres reservados para el trabajo interno del resaltador.

El nombre en sí, no es importante para el trabajo del bloque, pero si es útil para cuando queramos referenciarlo, desde otra parte del archivo de sintaxis.

El siguiente caso es un ejemplo de bloque con nombre:

```
<Block Name="Begin-End-Global" Start="Begin" End="End" >  
</Block>
```

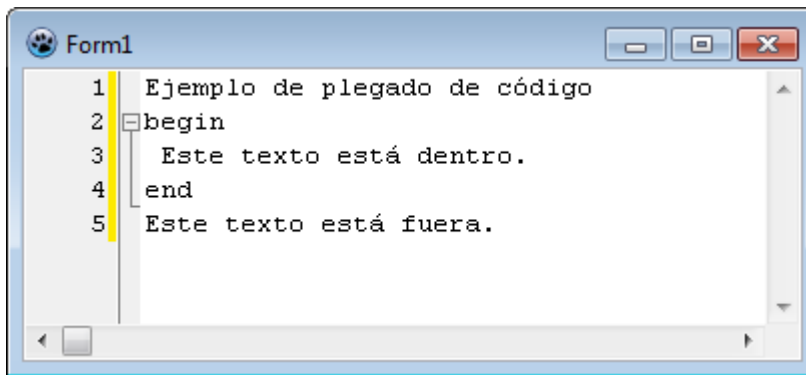
El nombre asignado al bloque, debe ser único. De otra forma se generará un error en el archivo de sintaxis.

La utilidad principal de un bloque es para el plegado de código. Por defecto, un bloque mostrará siempre la marca de plegado de código (a menos que esté contenido en una sola línea). Pero si quisiéramos especificarlo formalmente, deberíamos hacer:

```
<Block Start="Begin" End="End" Folding="True">
</Block>
```

Si quisiéramos tener un bloque definido, pero sin marca de plegado, deberíamos especificarlo con el parámetro Folding="False".

Un texto con marca de plegado en un bloque, se vería así:



Los delimitadores de bloque pueden ser también símbolos. Por ejemplo, la siguiente sentencia es válida:

```
<Block Start="{ " End="} ">
</Block>
```

Esta definición, considerará a todos los tokens dentro de las llaves como parte de un bloque, como se hace en el lenguaje C.

Los delimitadores de un bloque deben ser siempre identificadores o símbolos especiales. Si es que se define un delimitador de bloque que no se haya definido como token especial, se creará primero como token especial antes de procesar el bloque.

Por ejemplo, en la siguiente definición:

```
<Block Start="Inicio" End="Fin" Folding="True">
</Block>
```

Si el identificador “Inicio”, no se ha creado como un identificador especial, entonces, primero se creará “Inicio”, como token especial, dentro del resaltador, con el atributo “tkIdentif”.

Si el identificador especial, ya existía, no se crea de nuevo ni se modifica su atributo.

En la creación de bloques, no se validan los delimitadores. Por ejemplo, es posible escribir esta definición, sin genera errores:

```
<Block Start="Begin" End= "End"> </Block>
<Block Start="Begin" End= "End"> </Block>
```

A pesar de que usamos el mismo delimitador, en realidad estaríamos creando dos bloques totalmente independientes, pero generaríamos una confusión porque el mismo delimitador es inicio de dos bloques, de modo que cuando el resaltador encuentre el delimitador “Begin”, solo abrirá uno de ellos.

Cuando se definen bloques con el mismo delimitador de inicio y de fin, se debe restringir siempre a un bloque padre, porque de otra forma no se podrá cerrar el bloque. Consideremos la siguiente definición:

```
<Block Start="'" End="'" Name="blk_comill"></Block>
```

Tal como se ha definido, este bloque nunca se cerrará, porque cada nueva ocurrencia de la comilla, se entenderá como el inicio de otro bloque anidado. Para lograr que se cierre se podría restringir su validez al bloque principal.

```
<Block Start="'" End="'" Name="blk_comill" Parent="Main"></Block>
```

Se concluye que un bloque de este tipo (mismo delimitador inicial y final), nunca podrá ser anidado.

### 4.11 Bloques avanzados

Los bloques definidos usando los parámetros “Start” y “End”, de la etiqueta <BLOCK>, son una forma sencilla pero limitada, para especificar a los delimitadores.

Existe una forma alternativa que nos permite mayor libertad. Esta consiste en indicar los delimitadores en las etiquetas anidadas <Start> y <End>.

Para apreciar la diferencia, consideremos la siguiente declaración:

```
<Block Start="Repeat" End= "Until" >  
</Block>
```

Y la forma alternativa, equivalente:

```
<Block>  
  <Start> Repeat </Start>  
  <End> Until </End>  
</Block>
```

Esta forma de definición de bloques, nos da mayor libertad para:

- Incluir el parámetro adicional "TokPos" en los delimitadores.
- Poder incluir múltiples delimitadores.

El parámetro "TokPos", permite especificar mayor detalle con respecto a los delimitadores. Nos permite identificar completamente a un token especial que haya sido declarado con este parámetro (Ver Sección 4.5.1 y 4.4.1).

Por ejemplo, usar una definición como esta:

```
<Block>  
  <Start TokPos="2"> Inicio </Start>  
  <End> Fin </End>  
</Block>
```

Espera usar como delimitador inicial, a un identificador especial que se haya declarado así:

```
<Identifiers>  
  <Keyword TokPos="2"> Inicio </Keyword>  
  
</Identifiers>
```

Si no existiera esta definición, se procederá a crearla, porque el bloque requiere siempre que los delimitadores existan primero como tokens especiales.

La definición alternativa de delimitadores, nos permite también, definir más de un delimitador inicial o final, para un mismo bloque.

Por ejemplo si quisiéramos que el bloque BEGIN...END, se pueda abrir también con la palabra reservada INICIO, entonces la declaración sería:

```
<Block>
  <Start> Begin </Start>
  <Start> Inicio </Start>
  <End> End </End>
</Block>
```

Cada etiqueta <Start>, que se ponga dentro de la definición de bloque, permitirá definir un (y solo uno) delimitador inicial adicional.

Hay que notar que los espacios, antes y después de las etiquetas <Start></Start>, se ignoran. Es decir, que no se pueden incluir espacios como parte del delimitador.

De la misma forma, para agregar más delimitadores finales, debemos usar la etiqueta <End>, dentro de la definición del bloque:

```
<Block>
  <Start> Begin </Start>
  <Start> Inicio </Start>
  <End> End </End>
  <End> Fin </End>
</Block>
```

Los delimitadores adicionales, también admiten el parámetro “TokPos”. El siguiente ejemplo, muestra la definición de un bloque con dos delimitadores:

```
<Block Name="Consulta_Oracle" Start="Select">
  <End> End </End>
  <End TokPos="1"> / </End>
</Block>
```

Esta definición usa el símbolo “/”, como delimitador final y además especifica el parámetro “TokPos”, para especificar una posición específica del token.

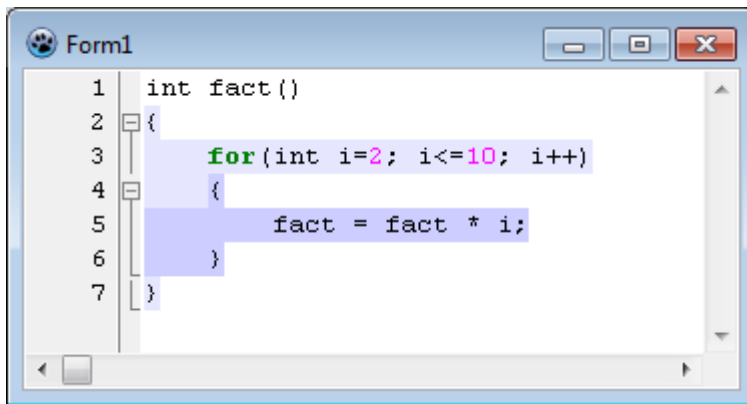
## 4.12 Coloreado de bloques

Los bloques son mayormente usados, para la implementación de plegado de código. Pero, estos bloques suelen corresponder, también, a estructuras sintácticas del lenguaje del texto.

Como una característica adicional, el resaltador, permite colorear el fondo de los bloques para proporcionar una ayuda visual, de la extensión de los bloques., Además, las marcas de plegado solo muestran la extensión de bloques, a nivel de líneas y no de columnas.

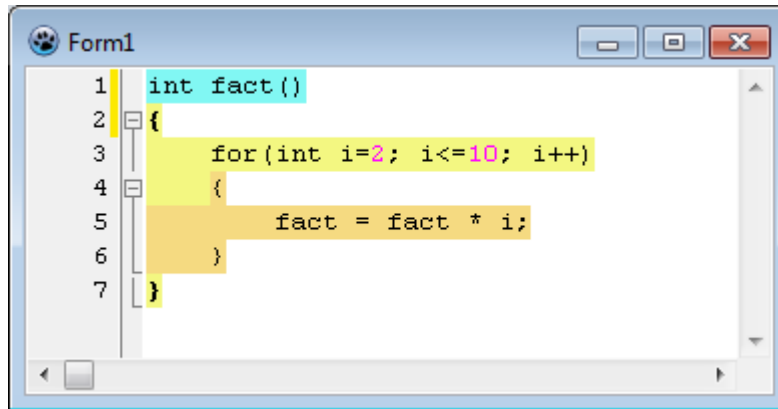
Para activar el coloreado de bloques, se debe configurar el lenguaje con el parámetro "ColorBlock" en "Level" o "Block".

Cuando ponemos ColorBlock en "Level", se procederá a realizar un coloreado automático de los bloques, usando el nivel de anidamiento de cada bloque. El siguiente ejemplo muestra cómo se vería un código sencillo de C, con coloreado de nivel:



En este estilo de coloreado, los bloques más internos, aparecerán con una tonalidad lila, tanto más oscuro, tanto más nivel de anidamiento tenga el bloque. El color base no es configurable por archivo o instrucciones, pero puede cambiarse modificando la unidad del resaltador.

El otro tipo de coloreado, es el coloreado por bloque. Si se pone el parámetro ColorBlock en "Block", el coloreado se hará usando como color de fondo, el color definido para cada bloque. La siguiente figura, muestra cómo se vería el mismo texto usando coloreado por bloques.



Este tipo de coloreado, no toma en cuenta el nivel de anidamiento, sino que usa el color personalizado que se especifica en la definición de cada bloque.

Para, crear este efecto, se ha usado la siguiente definición:

```
<?xml version="1.0"?>
<Language name="C" ext="c" ColorBlock="Block">
  <Block Name="int" Start="int" End=")" Parent="Main" BackCol="#81F7F3">
  </Block>
  <Block Name="Bloque_de_C" Start="{ " End="}" Parent="Main" BackCol="#F3F781">
    <Block Start="{ " End="}" BackCol="#F5DA81"></Block>
  </Block>
</Language>
```

El color, definido para cada bloque, se especifica en el parámetro “BackCol”. Este color, se debe especificar siempre en el formato “#RRGGBB”. Donde RR, GG y BB, corresponden a los valores de los componentes Rojo, Verde y Azul, expresados en dos dígitos hexadecimales.

Si se desea que el bloque no defina un color, sino que se muestre el color del bloque anterior, se puede usar el color transparente:

```
<Block Name="Parametros" Start="(" End=")" Folding="false"
BackCol="transparent"></Block>
```

Hay que anotar, que usar el coloreado por bloque, requiere un trabajo de procesamiento mayor en el resaltador, que cuando se usa resaltado por nivel.

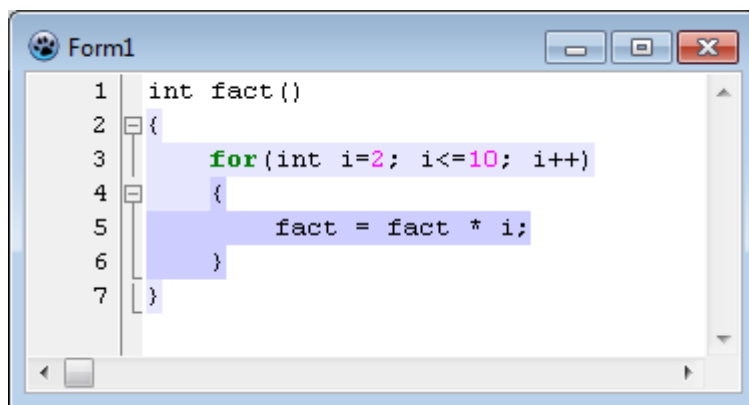
### 4.13 Alcance de un bloque

Hasta ahora, hemos definido bloques que son válidos en cualquier parte del código. Por ejemplo cuando definimos:

```
<Block Name="Bloque_de_C" Start="{ " End="}" ">
</Block>
```

Estamos diciendo que queremos que todos los tokens que se encuentren entre llaves, sean parte del bloque “Bloque\_de\_C”.

De modo que si escribimos un código de C con esta sintaxis, obtendremos la siguiente imagen:



En el resaltador se ha activado el coloreado de bloques, por nivel.

En la imagen se puede ver claramente que los dos bloques han sido reconocidos, en la sintaxis. Uno de ellos, es un bloque anidado, pero aun así es válido. Podríamos crear innumerables bloques anidados de esta forma.

Esto es así, porque en la definición del bloque, no hemos especificado el alcance del bloque en el parámetro “Parent”. Así que se asume que es el valor “None”, es decir que es bloque es válido en cualquier parte en donde aparezca.

El mismo resultado hubiéramos obtenido, si hubiéramos puesto explícitamente que el bloque es válido en cualquier parte, usando la siguiente forma:

```
<Block Name="Bloque_de_C" Start="{ " End="}" Parent="None">
</Block>
```



El bloque “None”, es en realidad una referencia a cualquier bloque. No es un bloque físico real. No tiene propiedades.

Existen 3 bloques predefinidos en el resaltador:

“None”-> Se refiere a cualquier bloque.

“Main”-> Identifica al bloque principal. El que no tiene anidamientos.

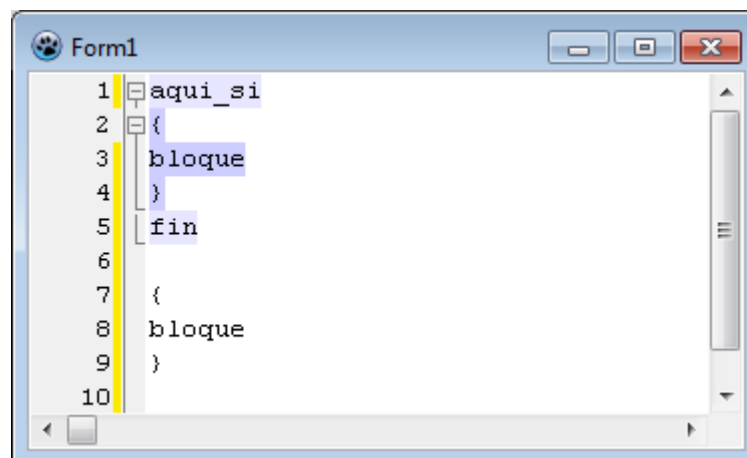
“MultiToken”-> Se refiere a los bloques que forman todos los tokens multilínea que tienen activado el plegado.

Es posible limitar la validez de un bloque, a áreas específicas del código, que pueden ser bloques o secciones.

Para definir explícitamente, que un bloque es válido, solamente dentro de los confines de otro bloque, debemos indicar el nombre, del bloque. Así por ejemplo, la siguiente definición crea primero un bloque “AQUI\_SI ... FIN”, y luego define el bloque “{ }”, para que solo sea válido, dentro de este bloque:

```
<Block Name="Aqui_si" Start="AQUI_SI" End="FIN">
</Block>
<Block Name="Bloque_de_C" Start="{" End="}" Parent="Aqui_si">
</Block>
```

Un código de ejemplo nos mostrará, el efecto de esta definición:

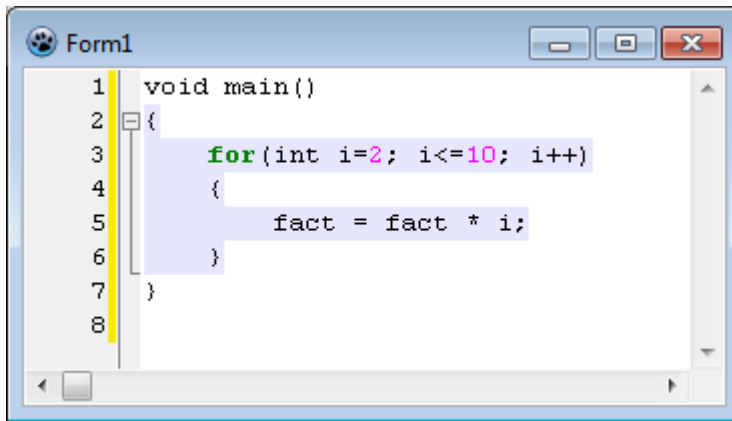


Como se aprecia, el segundo bloque, no es reconocido, porque se encuentra fuera del bloque “Aqui\_si”.

Hay que tener cuidado, con restringir un bloque a un rango. Sobre todo, los bloques que puedan aparecer anidados. Por ejemplo si definimos el bloque “{}”, para que solo sea válido en el bloque principal:

```
<Block Name="Bloque_de_C" Start="{" End="}" Parent="Main">
</Block>
```

Podríamos tener el siguiente resultado:



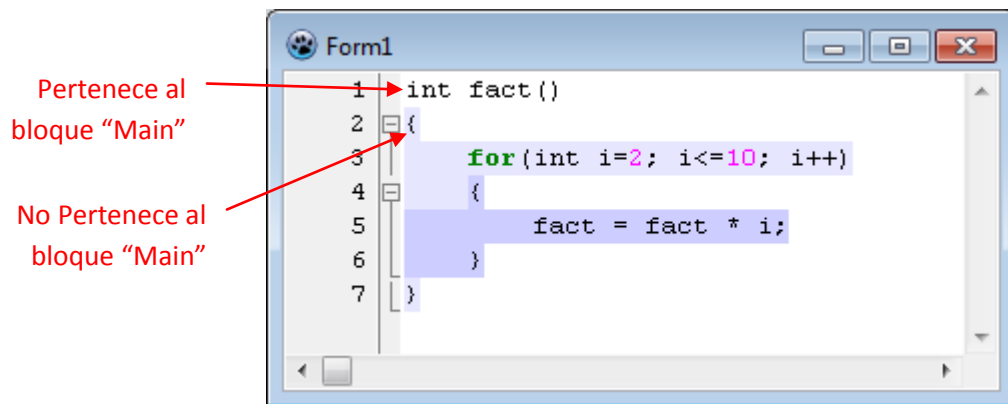
Y notaremos, que la segunda llave “{”, encontrada, no abre otro bloque, porque no está dentro del bloque principal. Sin embargo la primer llave “{”, encontrada, sí se reconoce como el delimitador final del bloque, excluyendo una parte del código y dando lugar a una definición errónea sintácticamente.

El bloque “Main” es el que incluye a todo el archivo. Todos los tokens y bloques (o secciones), que se creen estarán contenidos directa o indirectamente en el bloque “Main”.

No es necesario, crear el bloque “Main”, este bloque siempre existe y es único. Si lo hubiéramos definido, su declaración sería algo como esto:

```
<Block Name="Main" Start="" End="" Parent="">
</Block>
```

El interior de cualquier bloque, ya no es parte del bloque “Main”. Solo los tokens o estructuras que no están dentro de algún otro bloque, pertenecen directamente al bloque “Main”:



Cuando se crean bloques que solo sean válidos, en el bloque "Main", se estarán evitando que estos se puedan anidar y que solamente puedan aparecer en el cuerpo principal.

Es posible limitar el nivel de anidamiento de un bloque. Para ello se debe especificar en la estructura, cada bloque que corresponde a cada nivel.

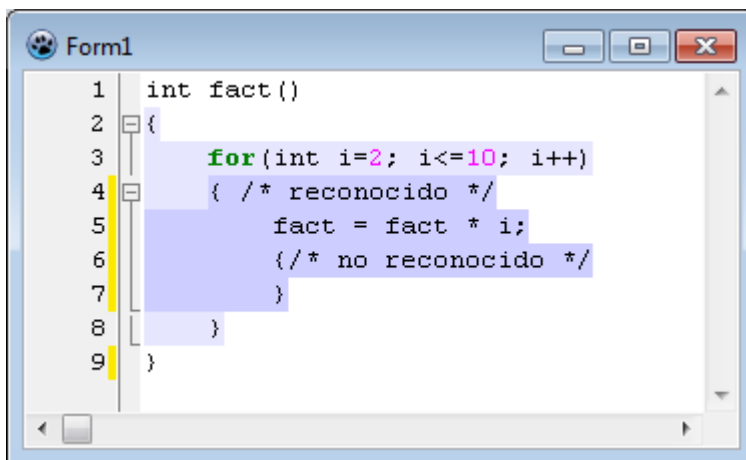
El siguiente ejemplo, es una definición para permitir un segundo anidamiento del bloque "{":

```

<Block Name="Bloque_de_C" Start="{ " End="}" Parent="Main">
</Block>
<Block Start="{ " End="}" Parent="Bloque_de_C" >
</Block>

```

Ahora con esta definición, se permitirá incluir un bloque "{", dentro de otro, pero no más. Así si se incluyera un tercer anidamiento, la sintaxis fallará:



Como hemos visto, el parámetro “Parent”, permite definir jerarquía de bloques a manera de árbol. Pero como una mejora en la sintaxis de la definición, podemos usar definiciones anidadas, para indicar esta jerarquía, sin necesidad de usar el parámetro “Parent”.

Por ejemplo, la siguiente definición:

```
<Block Name="Aqui_si" Start="AQUI_SI" End="FIN">
</Block>
<Block Name="Bloque_de_C" Start="{ " End="}" Parent="Aqui_si">
</Block>
```

Se puede escribir de forma simplificada, así:

```
<Block Name="Aqui_si" Start="AQUI_SI" End="FIN">
  <Block Name="Bloque_de_C" Start="{ " End="}"></Block>
</Block>
```

Esta forma evita usar el parámetro “Parent”, y es más clara. Por lo tanto es la que se usará, de preferencia.

## 4.14 Definición de Secciones

Las secciones se definen con las etiquetas `<Section>` `</Section>`.

Tiene los siguientes parámetros:

Start	Opcional	Delimitador inicial de la sección. Debe ser un token especial.
Name	Opcional	Es una cadena que identificará al bloque dentro del archivo de sintaxis. Es el nombre de la sección. Debe ser único para cada sección.
Folding	Opcional	Indica si se mostrará la marca de plegado en el editor. Esta marca solo aparecerá cuando la sección tenga más de una línea. Por defecto, está activado.
Parent	Opcional	Indica el bloque padre del bloque. Por defecto es "NONE", es decir que no está restringido a algún bloque.
BackCol	Opcional	Indica el color de fondo con que se pintarán, todos los tokens que se encuentren dentro de la sección. Por defecto está en "None"
Unique	Opcional	Indica que solo se permite una sección de este tipo, en el bloque que lo contiene. Por defecto está desactivado.
FirstSec	Opcional	

Los parámetros de una sección son similares a los parámetros de un bloque, excepto que no existe el delimitador Final, porque los bloques tienen otra mecánica para determinar su fin. Además existe un parámetro adicional.

Como las secciones son también bloques, la mayoría de reglas de los bloques, se aplican también aquí. También el coloreado de secciones es similar al coloreado de los bloques.

El siguiente ejemplo, permite declarar la sección INTERFACE, del lenguaje Pascal:

```
<Section Start="interface">  
</Section>
```

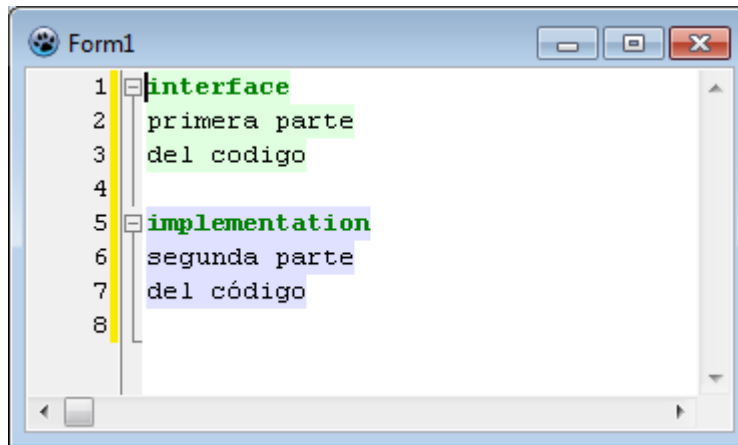
Esta sección se extenderá desde después del token "interface" hasta el fin del programa, si es que no se define alguna otra sección.

Si definimos dos secciones, entonces estaríamos dividiendo el código en dos o más partes:

```
<Section Start="interface"> </Section>  
<Section Start="implementation"> </Section>
```

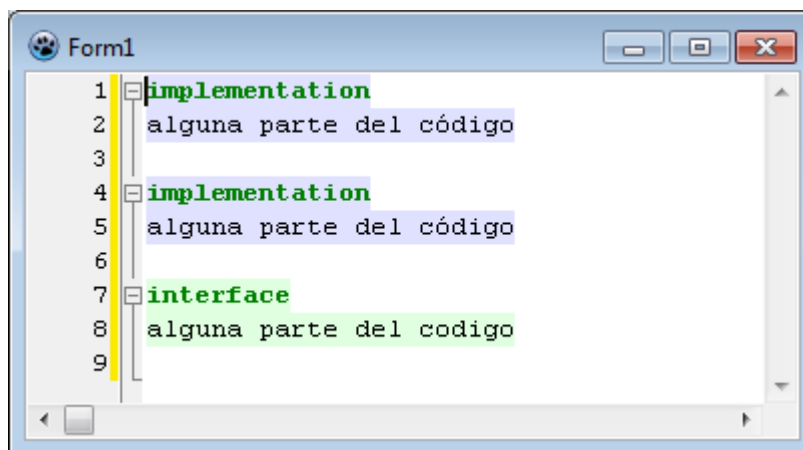
Ahora tendremos que una sección “interface” puede terminar en el inicio de una sección “implementation”, si es que existiera una.

La siguiente figura muestra un código de ejemplo, con las secciones creadas:



El orden en que se definen las secciones no es importante. No implica, un orden en el reconocimiento de las secciones.

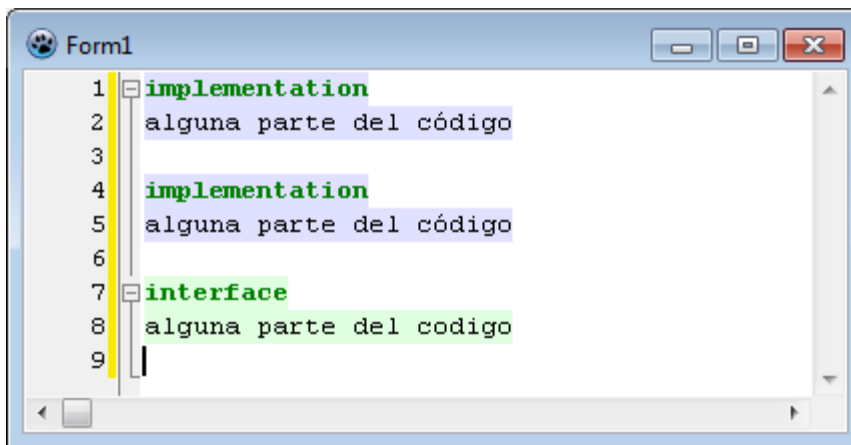
Así podríamos tener el siguiente código ejemplo:



En donde se ve, que las secciones han sido correctamente reconocidas, e inclusive, aparece una sección duplicada.

Para evitar que una misma sección, sea reconocida dos o más veces consecutivas, se debe usar el parámetro “Unique”. Si se activa, no se permitirá que la sección se repita después de ella misma. Es decir que solo se reconocerá la primera vez que aparezca.

Para graficar el efecto de “Unique”, supongamos el mismo texto anterior, pero con definición Unique=”true” e la sintaxis. El efecto sería este:



Como se puede ver, solo existe una sección “implementation” (ver marca de plegado), ya que la segunda no se considerará como una nueva sección. Sin embargo, si es posible definir nuevamente la misma sección “implementation”, después de una sección “interface”.

La propiedad “Name” de una sección, permite especificar un nombre para la sección. Si no especificamos un nombre, el resaltador, le asignará un nombre interno, que tiene la forma “Sec1”, “Sec2”, ..., dependiendo de cuantas secciones existan.

Se le puede asignar cualquier nombre a una sección excepto “Main”, “MultiToken” y “None”, que son nombres reservados para el trabajo interno del resaltador.

El nombre en sí, no es importante para el trabajo de la sección, pero si es útil para cuando queramos referenciarlo, desde otra parte del archivo de sintaxis.

El nombre de una sección debe ser diferente al de otra sección o al de otro bloque.

El siguiente caso es un ejemplo de sección con nombre:

```
<Section Name="interface" Start="interface">
</Section>
```

Las secciones también muestran la marca de plegado de código por defecto, y se puede desactivar con el parámetro “Folding”.

Las secciones suelen usarse, para representar bloques de código especiales en un lenguaje de programación, como por ejemplo las secciones “interface” e “implementation” de Pascal. Estas estructuras de lenguaje no se adaptan bien, para manejarlas como bloques.

La combinación, bloque-sección, permite representar estructuras comunes en los lenguajes de programación. Consideremos por ejemplo, la definición de procedimientos de Pascal;

```

      procedure Nombre();
      var
        x,y: integer;
      begin
        x:=1;
        y:=2;
      end;
```

Sección {

Sección {

A primera vista podría parecer que se puede expresar como bloques anidados, pero un análisis más detallado, revelará que no existe tal anidamiento, porque la parte de la declaración de variables, termina cuando empieza el cuerpo del procedimiento (o alguna declaración de tipo o constantes o más variables), que a su vez no está anidado dentro de la declaración del procedimiento, porque usa el mismo delimitador.

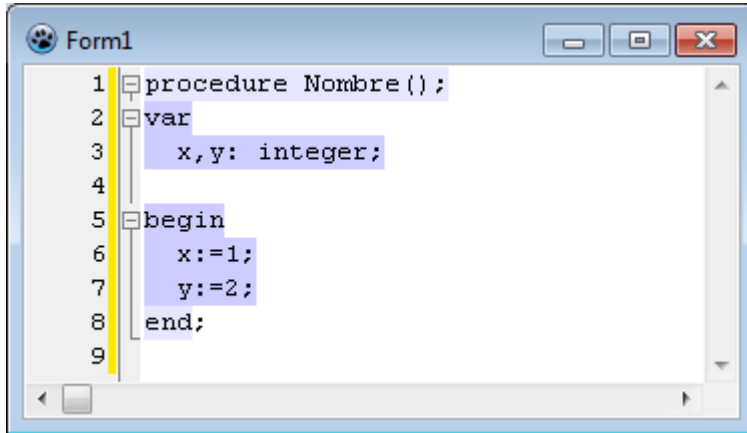
La mejor forma de expresar esta estructura, es manejando secciones dentro de un bloque que inicie con la palabra reservada “procedure” y termine con “end”:

```
<Block Name="Proc" Start="procedure" End="end" Parent="Main">
  <Section Start="var"> </Section>
  <Section Start="begin"> </Section>
</Block>
```

La declaración de variables, se maneja como una sección del bloque “Proc”.



Esta definición se puede completar agregando secciones para la declaración de tipos y constantes; y funcionará bien al menos para casos sencillos. El siguiente ejemplo muestra cómo se vería un procedimiento sencillo de pascal, usando esta definición:



Para mejorar la definición, habría que considerar que la palabra reservada “VAR”, puede aparecer también como modificador de parámetro (para indicar parámetros por referencia), y que no hay que confundirla con declaración de variables.

Para evitar esta mal interpretación, se puede crear una sección adicional, que incluya a la declaración de parámetros del procedimiento:

```
<Block Name="Proc" Start="procedure" End="end" Parent="Main">  
  <Block Name="ProcParam" Start="(" End=")" Folding="false"></Block>  
  <Section Start="var"> </Section>  
  <Section Start="begin" Unique="true"> </Section>  
</Block>
```

Ahora con esta protección, se ignorará la palabra VAR cuando se encuentre dentro de paréntesis, en el bloque del procedimiento.

Un detalle adicional, que se ha incluido en la definición anterior, es que se ha puesto el parámetro “Unique” a TRUE, en la sección “begin”. Esto evitará que se reconozca a cada palabra “Begin”, como el inicio de otra sección “Begin”. De otra forma no se podrían anidar bloques BEGIN ... END, dentro del cuerpo del procedimiento.

## 4.15 Más sobre secciones

Por lo general las secciones empiezan después de que ha empezado el bloque que lo contiene. Sin embargo, podría ser que necesitamos abrir una sección al mismo tiempo que se abre un bloque.

Esto es lo que se llama la “Primera Sección”.

Para indicar que una sección se abre con el bloque, debemos usar el parámetro “FirstSec”, en la declaración de la sección. El siguiente ejemplo muestra como estructurar una consulta SQL en un solo bloque con dos secciones:

```
<Block Start="SELECT" End=";" Parent="Main" Folding="False">
  <Section Start="SELECT" FirstSec="True"> </Section>
  <Section Start="FROM"> </Section>
</Block>
```

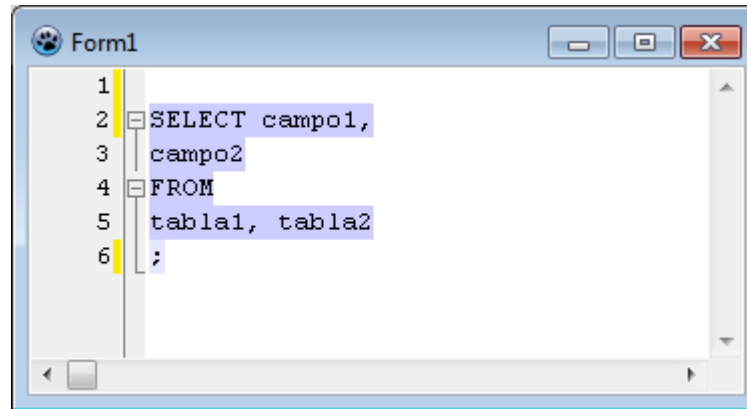
Como la sección SELECT, está declarada con FirstSec="TRUE", entonces esta sección se abrirá apenas se abra el bloque que inicia con SELECT.

Si tan solo hubiéramos hecho:

```
<Block Start="SELECT" End=";" Parent="Main" Folding="False">
  <Section Start="SELECT"> </Section>
  <Section Start="FROM"> </Section>
</Block>
```

La sección que empieza con SELECT, no se abriría al inicio porque SELECT, es también el delimitador inicial del bloque y la verificación de secciones, se hace después de reconocer la palabra SELECT.

Visualmente tendríamos la siguiente imagen:



Ahora se tienen dos secciones plegables dentro de un bloque, que no es plegable (y que tampoco sería accesible con el ratón, si es que fuera plegable).

Las secciones, al igual que los bloques, también soportan la definición alternativa para delimitadores. Por ejemplo, el siguiente código muestra cómo crear una sección con dos delimitadores iniciales:

```
<Section Name="Inicio_Sección">  
  <Start> Begin </Start>  
  <Start TokPos="1"> Inicio </Start>  
</Section>
```

## 5 Configurando la sintaxis por código

Así como se puede configurar la sintaxis, mediante el uso de un archivo externo, se puede hacer también usando métodos del resaltador.

Cuando se crea el resaltador, no inicia de cero. Algunas configuraciones están predefinidas. Estas son:

- Se crean los atributos del resaltador.
- Se crea la definición de tokens de espacios y no se podrá cambiar.
- Se configuran la definición de identificadores por defecto.
- Se crean los bloques “MainBlk” y “MulTokBlk”.

No se configuran cadenas, ni comentarios, ni palabras claves. Estos debe definirlos el usuario.

Los atributos que se crean en el resaltador, tienen asignados algunas propiedades por defecto. Por ejemplo las palabras claves se mostrarán con texto en color verde.

Una secuencia típica para el uso del resaltador, se parecerá a esta:

```
uses
    ... , SynHighlighterFacil;    //incluye unidad

procedure TForm1.FormCreate(Sender: TObject);
var
    hlt : TSynFacilSyn;    //referencia al resaltador
begin
    hlt := TSynFacilSyn.Create(self); //crea resaltador
    SynEdit1.Highlighter := hlt;    //configura el resaltador
    //configura atributos de los tokens
    hlt.KeywordAttri.Foreground:=clGreen;
    hlt.CommentAttri.Foreground:=clGray;
    ...
    //configura la sintaxis del resaltador
    ...
end;
```

Solo es necesario configurar los atributos necesarios. Los demás se pueden dejar en sus valores por defecto.

Configurar la sintaxis del resaltador, equivale a definir las reglas léxicas y sintácticas que se aplicarán para realizar el coloreado del texto. Básicamente se puede dividir en dos partes: La definición de tokens y la definición de bloques.

Para crear una nueva sintaxis, se debe seguir una secuencia estricta:

1. Limpiar la tabla de métodos: `ClearMethodTables()`
2. Definición de Identificadores: `DefTokIdentif()`
3. Definición de Números y otros Tokens por contenido.
4. Limpiar tabla de identificadores especiales: `ClearSpecials()`
5. Definición de identificadores especiales: `AddIdentSpec()`
6. Definición de símbolos especiales: `AddSymbSpec()`
7. Definición de tokens delimitados (cadenas, comentarios, etc.)
8. Definición de bloques y secciones.
9. Reconstruir la sintaxis: `Rebuild()`

El primer paso en la definición de una sintaxis, es la definición de los identificadores, porque estos tokens son la base de la mayoría de lenguajes.

La tabla de métodos, es una estructura interna del resaltador que permite seleccionar que método debe procesar un carácter cuando se realiza la exploración del texto. El objetivo de usar una tabla de métodos es acelerar la velocidad de respuesta del resaltador.

Limpiar la tabla de métodos es importante, si se van a definir los identificadores, u otros tokens por contenido, porque el resaltador aplica, al crearse, una definición por defecto para identificadores. Si se especificara una definición nueva sin limpiar la tabla de métodos, ambas definiciones se superpondrían.

Al limpiar la tabla de métodos, se borra la definición de todos los tokens de contenido (incluidos los identificadores) y de los tokens delimitados, por ello es el primer paso en la definición de cualquier sintaxis.

La primera vez que se crea una sintaxis, no es necesario limpiar las matrices de identificadores especiales, porque no se crea ningún identificador especial (como los KEYWORDS) por defecto.

Si tan solo se desea agregar identificadores especiales, bastará con ejecutar `AddKeyword()`

ClearSpecials() limpia las tablas de identificadores especiales y la tabla de símbolos especiales. Por lo tanto, borra también la definición de los tokens delimitados.

El método Rebuild() permite procesar todos los símbolos especiales y los tokens delimitados con delimitador inicial de tipo símbolo. Si no se ejecuta, no tendrá efecto las definiciones indicadas.

Las instrucciones de configuración de sintaxis, pueden generar errores internos. Cuando, se detecta un error en la definición de la sintaxis o de otro tipo, no se detiene la ejecución del programa, sino que se ignora la instrucción.

Para ver si se ha generado un error, se debe revisar siempre la propiedad "Err", del resaltador, después de cada instrucción crítica:

```
hlt.AddKeyword('array');  
if hlt.Err <> '' then begin  
    //hubo error  
    ...  
End;
```

La propiedad "Err", contiene un texto descriptivo (en español) del error encontrado.

Al generarse un error interno, no es conveniente continuar con la carga, porque se puede haber dejado una configuración en un estado inválido. En caso de error, lo más prudente es limpiar nuevamente la sintaxis y cargar todo de nuevo.

## 5.1 Tipos de tokens y atributos

Como se sabe, existen un número determinado de tipos de tokens que se crean siempre con el resaltador. Estos tipos son:

- tkEol
- tkSymbol
- tkSpace
- tkIdentif
- tkNumber
- tkKeyword
- tkString

## tkComment

Y están definidos como referencias a objetos “TSynHighlighterAttributes”. Esto significa que estos tipos de tokens son a la vez los objetos atributos, con los que se deben configurar la apariencia del texto en pantalla.

Si se quiere configurar la apariencia que tendrán los tokens “tkString” (que normalmente representan a las constantes cadenas), deberíamos hacer algo como esto:

```
hlt := TSynFacilSyn.Create(self);      //crea al resaltador
...
hlt.tkString.Foreground:=clPurple;      //color de texto
hlt.tkString.Style := [fsBold, fsUnderline]; //negrita y subrayado
...
```

Como el objeto hlt.tkString, es de tipo “TSynHighlighterAttributes”, podemos cambiar directamente sus parámetros.

También podríamos usar una variable adicional, para acceder a los atributos:

```
var
  atribString: TSynHighlighterAttributes;
...
hlt := TSynFacilSyn.Create(self);      //crea al resaltador
...
atribString := hlt.tkString;           //referencia al atributo
atribString.Foreground:=clPurple;      //color de texto
atribString.Style := [fsBold, fsUnderline]; //negrita y subrayado
```

Para crear nuevos atributos, se debe usar variables de tipo “TSynHighlighterAttributes”, de forma similar. En el siguiente código creamos dos nuevos tipos de tokens, y definimos sus atributos:

```
var
  tkOperator: TSynHighlighterAttributes;
  tkDelimiter: TSynHighlighterAttributes;
...
//crea nuevo tipo de token llamado "operador"
tkOperator := hlt.NewTokType('operador');
tkOperator.Foreground:=clRed;          //color de texto

//crea nuevo tipo de token llamado "delimitador"
```

```
tkDelimiter := hlt.NewTokType('delimitador');  
tkDelimiter.Foreground:=clRed;           //color de texto
```

Los nuevos atributos, pueden ahora ser usados como si fueran iguales a los atributos pre-definidos del resaltador. El nombre indicado, en NewTokType(), no es importante para fines de configuración por código, pero si es útil cuando algún programa acceda a los atributos del resaltador, como una ventana de configuración. Este nombre se usa también para identificar al nuevo tipo dentro del archivo XML.

El nombre del nuevo tipo debe ser único para cada tipo. De otra forma se generará una excepción en tiempo de ejecución.

Si se quiere referenciar a un atributo existente, por su nombre, se debe usar la función GetAttribByName():

```
tkOperador := lex.GetAttribByName('Operador'); //se debe haber creado
```

De esta forma se puede acceder también a los atributos que hayan sido creados desde el archivo XML, ya que es la única forma de identificarlos externamente. GetAttribByName(), no hace distinción entre mayúscula o minúscula.

## 5.2 Definición de Identificadores

La primera definición de token que se debe hacer dentro de un resaltador es la de los identificadores.

Para definir a los identificadores por código, se deben definir los caracteres inicial y siguiente, de este modo:

```
hlt.ClearMethodTables;  
...  
hlt.DefTokIdentif('[A..Za..z]', 'A..Za..z0..9');
```

En este ejemplo se supone que “hlt” es el resaltador. La instrucción “ClearMethodTables” es necesaria para limpiar la tabla de métodos del resaltador.

El método DefTokIdentif(), permite definir a los tokens identificadores. Estos se definen siempre por contenido.



El primer parámetro define los caracteres válidos como carácter inicial para los identificadores. El segundo parámetro define los caracteres siguientes.

El primer parámetro se escribe siempre encerrado entre corchetes para indicar que se trata de una lista de caracteres y no de una cadena común. Si no se escribe como lista, se generará un error interno.

Los caracteres iniciales, y siguientes pueden ser cualquier carácter, imprimible o no.

Se puede usar las formas cortas para indicar intervalo de caracteres:

```
A..Z = ABCDEFGHIJKLMNOPQRSTUVWXYZ  
a..z = abcdefghijklmnopqrstuvwxyz  
0..9 = 0123456789
```

Al crearse el resaltador, al inicio, se crea también una definición por defecto para los identificadores. Esta definición es:

```
hlt.DefTokIdentif('[A..Za..z$_]', 'A..Za..z0123456789_');
```

Esto significa que aunque, no se especifique alguna definición, se inicia siempre con una para los identificadores.

Después de la definición de identificadores, todos los tokens que pertenezcan a esa categoría, pasarán a ser del tipo “tkIdentif”, y se mostrarán con los atributos de este tipo.

Si se incluyen caracteres con tilde en la definición de identificadores, hay que considerar que la codificación usada es UTF-8. Así por ejemplo, la siguiente definición:

```
hlt.DefTokIdentif('[áA..Za..z]', 'A..Za..z0..9');
```

Habilitará el reconocimiento de identificadores con la “a” minúscula tildada, pero como la codificación UTF8 de la letra “á” es 195,161 lo que en realidad se está haciendo es incluir los caracteres 195 y 161, como caracteres válidos para el inicio de un identificador. Como los caracteres acentuados se componen de dos bytes, no habría forma de ser específico para filtrar solo algunos caracteres acentuados (El resaltador filtra solo por un byte). Si ingresamos el código 195 en la definición, estaremos permitiendo realmente, todos los caracteres de la página de

código 195 de UTF8, que contiene a todos los caracteres acentuados de los idiomas latinos: áàäëèë ...

Para los caracteres siguientes, si es recomendable definir los valores posibles indicando los caracteres válidos:

```
hlt.DefTokIdentif('[áA..Za..z]', 'áéíóúÁÉÍÓÚA..Za..z0..9');
```

Esta definición sería la más apropiada para aceptar las vocales acentuadas como parte de identificadores. Sin embargo, por la naturaleza del UTF-8 (dos bytes por carácter) y la naturaleza del resaltador (un byte por carácter), esta definición no es exacta. Por ejemplo el identificador “èco” será reconocido parcialmente porque inicia con el carácter 192, pero el byte siguiente 168 no será considerado como parte del identificador, haciendo que el resaltador genere un identificador de un byte, pero para el editor es “medio carácter”. El resultado puede ser que el carácter aparezca invisible.

### 5.3 Definición de Tokens por contenido

Los tokens por contenido se definen de la misma forma a como se definen los identificadores, pero especificando el tipo de token:

```
hlt.ClearMethodTables;  
...  
hlt.DefTokContent('[0123456789]', '0123456789', '', hlt.tkNumber);
```

En este ejemplo se supone que “hlt” es el resaltador. La instrucción “ClearMethodTables” es necesaria para limpiar la tabla de métodos del resaltador. “hlt.tkNumber”, es el tipo de token al que pertenecerán los tokens que cumplan con esta definición.

El método DefTokContent(), permite definir tokens por contenido. Su definición es:

```
procedure DefTokContent(dStart, charsCont, charsEnd: string;  
                        typToken: TSynHighlighterAttributes);
```

El parámetro “dStart” define el delimitador inicial o los caracteres válidos como carácter inicial para el token por contenido. El parámetro “charsCont” define los caracteres siguientes, el parámetro “charsEnd” define los caracteres válidos como carácter final, y el cuarto parámetro define el tipo de token.

Los caracteres iniciales, siguientes y finales pueden ser cualquier carácter, imprimible o no.

Solo se admiten 4 definiciones de tokens por contenido (aparte de los identificadores), para una misma sintaxis.

Otro ejemplo podría ser este:

```
hlt.ClearMethodTables;  
...  
hlt.DefTokContent('$', 'a..zA..Z0..9', '', tkDirective);
```

Se puede usar las siguientes formas cortas para indicar intervalos de caracteres:

```
A..Z = ABCDEFGHIJKLMNOPQRSTUVWXYZ  
a..z = abcdefghijklmnopqrstuvwxyz  
0..9 = 0123456789
```

También es posible especificar una cadena como delimitador inicial. Por ejemplo la siguiente definición permite reconocer los números hexadecimales en lenguaje C:

```
hlt.DefTokContent('0x', '0123456789', '', hlt.tkNumber);
```

Cuando el parámetro dStart, no incluye corchetes, se entiende que se trata de un único delimitador (no un conjunto de caracteres). Para ver las reglas que deben cumplir los delimitadores ver la sección 3.5 - Delimitadores de Token.

## 5.4 Identificadores Especiales

Los identificadores especiales se pueden agregar por código, usando el método AddIdentSpec(), como se muestra en el siguiente ejemplo:

```
hlt.AddIdentSpec('IF', hlt.tkKeyword);
```

El primer parámetro es el identificador a agregar. Este texto debe ser un identificador, es decir que debe cumplir con la definición de los identificadores. Por ejemplo no se podrá definir la

palabra “\$INCLUDE” como un identificador especial, si es que el carácter \$, no forma parte de los caracteres iniciales válidos para un identificador.

La caja (mayúscula o minúscula) del identificador, no se tomará en cuenta por defecto, a menos que se ponga la propiedad “CaseSensitive” a TRUE en el resaltador.

Los caracteres iniciales para los identificadores especiales solo pueden ser los caracteres alfabéticos en inglés, y los caracteres `_`, `$`, `%`, `@` y `&`. No se puede usar otros caracteres que no estén en este grupo.

Se puede asignar cualquier atributo a los identificadores especiales. El siguiente ejemplo, crea el identificador especial PI con atributo de número:

```
hlt.CaseSensitive := true;  
hlt.AddIdentSpec('PI', hlt.tkNumber);
```

Al aplicar esta definición, todos los identificadores “PI”, se pintarán en pantalla con los atributos configurados en “hlt.TkNumber”.

En este ejemplo, se ha definido que todos los identificadores se lean considerando su caja. Por lo tanto los identificadores “Pi” y “pi”, no se resaltarán como números.

Si es que hubiera palabras reservadas anteriores, se debe limpiar primero la memoria, usando el método ClearSpecials():

```
hlt.ClearSpecials;  
...  
hlt.AddIdentSpec('PI', hlt.tkNumber);  
hlt.AddIdentSpec('EPSILON', hlt.tkNumber);  
...
```

El método AddIdentSpec(), permite agregar los identificadores especiales, uno a uno. Para agregar varios identificadores en grupo se debe usar el método AddIdentSpecList():

```
hlt.ClearSpecials;  
...  
hlt.AddIdentSpecList('begin end if', hlt.tkKeyword);
```

Los identificadores en la lista, deben estar separados por al menos un espacio en blanco.

Como las palabras reservadas (atributo tkKeyword) son muy comunes, existe un método especial para agregar palabras reservadas: AddKeyword().

```
hlt.AddKeyword('array');
```

Los atributos de las palabras reservadas se cambian usando la propiedad “KeywordAttribute” del resaltador.

No es recomendable tampoco incluir un número grande de identificadores especiales, porque se puede obtener un deterioro en el rendimiento del resaltador.

Los identificadores especiales se pueden ir agregando dinámicamente, conforme a las necesidades del lenguaje. No es necesario, realizar configuraciones adicionales. Lo que si, no se puede hacer (al menos directamente) es eliminar identificadores especiales, que hayan sido agregados.

Los identificadores especiales agregados, no deben repetirse, de otra forma se generará un error interno y se ignorará la instrucción.

Si bien el orden en que se agregan los identificadores, no afecta a la sintaxis, si puede afectar el rendimiento del resaltador. Para un procesamiento más rápido, se recomienda poner primero las palabras que en el lenguaje tengan más probabilidad de aparecer.

Al observar la declaración de AddIdentSpec(), veremos que tiene un parámetro adicional:

```
procedure AddIdentSpec(iden: string; tokTyp: TSynHighlighterAttributes;  
TokPos: integer=0);
```

El parámetro “TokPos”, nos permite indicar el orden en que se debe encontrar el token para ser considerado como identificador especial.

En la siguiente definición, condicionaremos a que el identificador “INICIO”, sea considerado como token especial, solo cuando aparezca como el primer token de la línea.

```
hlt.AddIdentSpec('INICIO', hlt.tkKeyword, 1);
```

En cualquier otra posición, en que se encuentre el token, no será considerado como identificador especial.

De la misma forma se puede indicar que el token ocupe la posición 2, 3 o cualquier otra. El primer token que aparezca en una línea será siempre el número 1, y así sucesivamente. No olvidar que, inclusive los espacios son también un token.

Por defecto, el valor de “TokPos”, es cero, lo que significa que no se considera el orden que ocupe el token. Pero al definir “TokPos”, se le está dando un parámetro más, de modo que, puede haber en el resaltador, dos identificadores especiales con la misma cadena pero con distinta posición de token. Es decir, la siguiente definición no generará error:

```
hlt.AddIdentSpec('INICIO', hlt.tkKeyword, 1);  
hlt.AddIdentSpec('INICIO', hlt.tkKeyword, 2);
```

Sin embargo, la siguiente definición si dará error, por cuanto hay un cruce en las definiciones:

```
hlt.AddIdentSpec('INICIO', hlt.tkKeyword, 1);  
hlt.AddIdentSpec('INICIO', hlt.tkKeyword);
```

## 5.5 Símbolos Especiales

Los símbolos especiales se pueden agregar por código, usando el método AddSymbSpec(), como se muestra en el siguiente ejemplo:

```
hlt.AddSymbSpec('+', tkOperator);
```

El primer parámetro es el símbolo a agregar. Este texto debe ser un símbolo, es decir que debe cumplir con la definición de los símbolos. Por ejemplo no se podrá definir la palabra “\$\$” como un símbolo especial, si es que el carácter \$, no forma parte de los caracteres iniciales válidos para un símbolo (podría haber sido asignado a identificadores).

La caja (mayúscula o minúscula) del símbolo, no se tomará en cuenta por defecto, a menos que se ponga la propiedad “CaseSensitive” a TRUE en el resaltador. Sin embargo, el efecto de “CaseSensitive” no suele verse, porque los símbolos no suelen incluir caracteres alfabéticos.

El segundo parámetro es el tipo de token. Puede ser uno de los predefinidos o cualquier otro que hayamos creado. En este ejemplo suponemos que hemos creado el tipo “tkOperator”.

Se puede asignar tipo a los tokens especiales. El siguiente ejemplo, crea el símbolo especial “+A” con atributo de tkString:

```
hlt.CaseSensitive := true;  
hlt.AddSymbSpec('+A', hlt.tkString);
```

Al aplicar esta definición, todos los símbolos “+A”, se pintarán en pantalla con los atributos configurados en “tkString”. Esta definición puede causar conflictos si es que se analiza textos como “B+A”, ya que por definición la parte “+A”, se considerará como un solo token con atributo “tkString”.

Si es que hubiera símbolos especiales anteriores, se debe limpiar primero la memoria, usando el método ClearSpecials()<sup>5</sup>:

```
hlt.ClearSpecials; //limpia identificadores y símbolos especiales  
...  
hlt.AddSymbSpec('+', tkOperator);  
hlt.AddSymbSpec('-', tkOperator);  
...  
hlt.Rebuild; //es necesario para terminar la definición
```

Rebuild(), actualiza la tabla de métodos y ordena la tabla interna de símbolos especiales para que puedan ser correcta y rápidamente detectados en la exploración. Además procesa los tokens delimitados y bloques.

Como norma es conveniente hacer siempre Rebuild() al final de la definición del lenguaje.

El método AddSymbSpec(), permite agregar los símbolos especiales, uno a uno. Para agregar varios símbolos en grupo se debe usar el método AddSymbSpecList():

```
hlt.ClearSpecials;  
...  
hlt.AddSymbSpecList('+ - * / < > = :=', tkOperator);
```

---

<sup>5</sup> ClearSpecials() también elimina los identificadores especiales. Así que debe ejecutarse antes de crear tanto identificadores especiales como símbolos especiales.

Los símbolos en la lista, deben estar separados por al menos un espacio en blanco. Por lo tanto, esta definición no permitirá crear símbolos especiales que incluyan el carácter espacio.

No es recomendable tampoco incluir un número grande de símbolos especiales, porque se puede obtener un deterioro en el rendimiento del resaltador. La búsqueda de símbolos especiales no está tan optimizada como la de identificadores especiales.

Los símbolos especiales agregados, no deben repetirse, de otra forma se generará un error interno y se ignorará la instrucción.

El orden en que se agregan los símbolos especiales, no es tan importante en el desempeño, porque los símbolos especiales sufrirán un ordenamiento previo con `RebuildSymbols()`.

El método `AddSymbSpec()`, tiene un parámetro adicional:

```
procedure AddSymbSpec(symb: string; tokTyp: TSynHighlighterAttributes;  
                    TokPos: integer=0);
```

El parámetro “TokPos”, nos permite indicar el orden en que se debe encontrar el token para ser considerado como símbolo especial.

En la siguiente definición, condicionaremos a que el símbolo “.”, sea considerado como token especial, solo cuando aparezca como el primer token de la línea.

```
hlt.AddSymbSpec('.', hlt.tkKeyword, 1);
```

En cualquier otra posición, en que se encuentre el token, no será considerado como símbolo especial.

De la misma forma se puede indicar que el token ocupe la posición 2, 3 o cualquier otra. El primer token que aparezca en una línea será siempre el número 1, y así sucesivamente. No olvidar que, inclusive los espacios son también un token.

Por defecto, el valor de “TokPos”, es cero, lo que significa que no se considera el orden que ocupe el token. Pero al definir “TokPos”, se le está dando un parámetro más, de modo que, puede haber en el resaltador, dos identificadores especiales con la misma cadena pero con distinta posición de token. Es decir, la siguiente definición no generará error:

```
hlt.AddIdentSpec('.', hlt.tkKeyword, 1);
```



```
hlt.AddIdentSpec('.', hlt.tkKeyword, 2);
```

Sin embargo, la siguiente definición si dará error, por cuanto hay un cruce en las definiciones:

```
hlt.AddIdentSpec('.', hlt.tkKeyword, 1);  
hlt.AddIdentSpec('.', hlt.tkKeyword);
```

## 5.6 Definición de Tokens delimitados

Se definen con el método DefTokDelim(), que tiene la siguiente declaración:

```
procedure DefTokDelim(dStart, dEnd: string;  
    tokTyp: TSynHighlighterAttributes;  
    tipDel: TFaTypeDel=tdUniLin; havFolding: boolean=false);
```

Los parámetros “dStart” y “dEnd”, son los delimitadores del token. El parámetro “tokTyp”, indica el tipo o atributo del token. Los parámetros opcionales permiten configurar si el token es multilínea o no, y si incluirá plegado de código.

Un caso simple para definición de cadenas sería:

```
hl := TSynFacilSyn.Create(nil);  
SynEdit1.Highlighter := hl;  
hl.DefTokDelim('"', '"', hl.tkString);  
hl.Rebuild;  
...
```

El método Rebuild(), se debe ejecutar, siempre, después de ingresar todos los tokens delimitados, de otra forma no tendrá efecto, la definición.

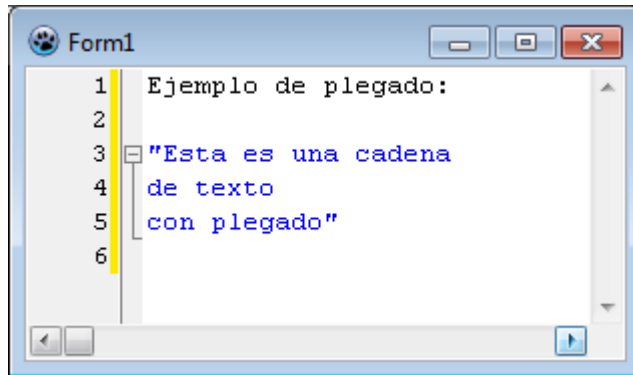
Esta sentencia asignará el atributo “tkString” a la secuencia de caracteres, que se encuentre entre comillas simples, incluyendo a las comillas simples.

Si quisiéramos que las cadenas se puedan extender por más de una línea y además se puedan plegar, el siguiente código funcionará:

```
...
```

```
hl.DefTokDelim('\"', '\"', tk.tkString, tdMulLin, true);  
hl.Rebuild;  
...
```

Un texto, con este resaltador, se podría mostrar así:



La opción de plegado de código solo es válida cuando se manejan tokens multi-línea.

Si no se especifica el delimitador final de un token delimitado, o se le asigna la cadena vacía, se asume que se extiende hasta el fin de la línea actual. Este comportamiento es común en la definición de comentarios de una sola línea. El siguiente código crea cadenas y comentarios de una sola línea, en la sintaxis del resaltador:

```
...  
hl.DefTokDelim('\"', '\"', hl.tkString);  
hl.DefTokDelim('/*', '*/', hl.tkComment);  
hl.Rebuild;  
...
```

Los atributos que pueden ser asignados son, los todos los que tiene definidos el resaltador o cualquier otro que se haya creado.

Se pueden definir tantos tokens delimitados como se quiera, considerando que a mayor cantidad de ellos, es mayor el procesamiento que debe hacer el resaltador. Los tokens con delimitadores de un carácter, son los que se procesan más rápido.

Las cadenas y comentarios son ejemplos clásicos de tokens delimitados.

Al definir tokens delimitados, se debe considerar que los delimitadores de un token no sean a la vez delimitadores, de otro token. De otra forma se generará un error interno y se ignorará la definición, que entra en conflicto.

Los delimitadores para un token son por lo general de tipo símbolo, pero pueden ser también de tipo identificador (Ver Sección 3.5 - Delimitadores).

Un ejemplo de token delimitado con tokens identificadores puede ser:

```
hl.DefTokDelim('REM', '', hl.tkComment);
```

Esta sintaxis corresponde a los comentarios del lenguaje de archivos Batch del DOS, que empiezan con la palabra reservada REM y se extienden hasta el fin de la línea.

## 5.7 Consideraciones sobre tokens delimitados

Al definir un token delimitado, se está especificando también la definición de “tokens delimitadores”. Un token delimitador, es un token especial, (símbolo o un identificador), que se usa para identificar a un token delimitado.

La siguiente declaración define un token cadena, a partir de dos identificadores:

```
hl.DefTokDelim('inicio', 'fin', hl.tkString, tdMulLin);
```

Al declarar “inicio”, como delimitador de token, se define al token “inicio” como identificador especial y se le define con la propiedad de delimitador de token de contenido, de modo que pierde su identidad como token único, y pasa a ser siempre parte de un token más grande de atributo tkString.

Si el token “inicio”, ya se hubiera declarado antes, como token especial, se ignorará su atributo, y pasará a ser un token delimitador. Si no se hubiera definido antes, se define al momento de crear el token delimitado.

El token “fin”, no se crea ni actualiza como token especial. Las rutinas de búsqueda encontrarán al identificador “fin”, por otro medio, sin que tenga que existir como token especial.

De hecho la rutina de búsqueda para el token final de tipo identificador, es bastante sencilla, de modo tal que hace una búsqueda exacta, considerando siempre la caja. La propiedad "CaseSensitive" no tiene efecto en este caso (Ver 6.9 - Limitaciones).

Un delimitador final, de token delimitado puede ser a la vez de delimitador de otros tokens delimitados, pero el delimitador inicial es exclusivo. No puede haber dos tokens delimitados con el mismo delimitador inicial.

Cuando se crea un token multilínea, existe la opción de activar o no el plegado de código. Y no hay mayor diferencia en tokens con o sin plegado, excepto por la marca de plegado que aparece cuando se crean con plegado de código. Si se intenta esta definición doble, solo se tomará en cuenta la última.

### ***5.8 Sin embargo, hay una diferencia más. Los tokens creados con plegado de código, generan "bloques de código" (Ver sección 5.9 - Ejemplo de configuración***

El siguiente código, crea una sintaxis sencilla para PHP:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  hlt1: TSynFacilSyn;
begin
  hlt1 := TSynFacilSyn.Create(self); //crea resaltador
  SynEdit1.Highlighter := hlt1;     //asigna al editor
  hlt1.ClearMethodTables;           //limpia tabla de métodos
  hlt1.ClearSpecials;               //para empezar a definir tokens
  //crea tokens por contenido
  hlt1.DefTokIdentif('[A..Za..z_]', 'A..Za..z0..9_');
  hlt1.DefTokContent('[0..9.]', '0..9xabcdefXABCDEF', '', hlt1.tkNumber);
  if hlt1.Err<>'' then ShowMessage(hlt1.Err);
  //define palabras claves
  hlt1.AddIdentSpec('__file__', hlt1.tkKeyword);
  hlt1.AddIdentSpec('__line__', hlt1.tkKeyword);
  hlt1.AddIdentSpec('array', hlt1.tkKeyword);
  hlt1.AddIdentSpec('and', hlt1.tkKeyword);
  hlt1.AddIdentSpec('break', hlt1.tkKeyword);
  //forma alternativa de declaración
  hlt1.AddIdentSpecList('case class const continue', hlt1.tkKeyword);
  hlt1.AddIdentSpecList('default die do double', hlt1.tkKeyword);
  hlt1.AddIdentSpecList('else elseif echo empty endfor endif', hlt1.tkKeyword);
  hlt1.AddIdentSpecList('endswitch endwhile eval exit extends', hlt1.tkKeyword);
```

```
hlt1.AddIdentSpecList('function false float for', hlt1.tkKeyword);  
//forma alternativa de declaración  
hlt1.AddKeyword('global');  
hlt1.AddKeyword('highlight_file');  
hlt1.AddKeyword('highlight_string');  
hlt1.AddKeyword('if');  
hlt1.AddKeyword('int');  
hlt1.AddKeyword('include');  
hlt1.AddKeyword('integer');  
hlt1.AddKeyword('isset');  
hlt1.AddKeyword('list');  
hlt1.AddKeyword('new');  
hlt1.AddKeyword('object');  
hlt1.AddKeyword('old_function');  
hlt1.AddKeyword('or');  
hlt1.AddKeyword('print');  
hlt1.AddKeyword('real');  
hlt1.AddKeyword('require');  
hlt1.AddKeyword('return');  
hlt1.AddKeyword('show_source');  
hlt1.AddKeyword('static');  
hlt1.AddKeyword('string');  
hlt1.AddKeyword('switch');  
hlt1.AddKeyword('true');  
hlt1.AddKeyword('unset');  
hlt1.AddKeyword('var');  
hlt1.AddKeyword('while');  
hlt1.AddKeyword('xor');  
//crea tokens delimitados  
hlt1.DefTokDelim('',' ', hlt1.tkString);  
hlt1.DefTokDelim(' ',' ', hlt1.tkString);  
hlt1.DefTokDelim('//',' ', hlt1.tkComment);  
hlt1.DefTokDelim('/*','*/', hlt1.tkComment, tdMulLin);  
hlt1.Rebuild; //reconstruye  
end;
```

En esta sintaxis solo se define el tratamiento de tokens. Solo se usan los tipos de tokens que ya existen en el resaltador, no se usan tipos de tokens personalizados.

Definición de bloques). De esta forma existen algunas propiedades adicionales que se pueden aplicar a estos tokens. Para más información ver la sección 5.11 - Bloques como objetos.

## 5.9 Ejemplo de configuración

El siguiente código, crea una sintaxis sencilla para PHP:

```
procedure TForm1.FormCreate(Sender: TObject);
var
    hlt1: TSynFacilSyn;
begin
    hlt1 := TSynFacilSyn.Create(self); //crea resaltador
    SynEdit1.Highlighter := hlt1;     //asigna al editor
    hlt1.ClearMethodTables;            //limpia tabla de métodos
    hlt1.ClearSpecials;                //para empezar a definir tokens
    //crea tokens por contenido
    hlt1.DefTokIdentif('$A..Za..z_', 'A..Za..z0..9_');
    hlt1.DefTokContent('[0..9.]', '0..9xabcdefXABCDEF', '', hlt1.tkNumber);
    if hlt1.Err <> '' then ShowMessage(hlt1.Err);
    //define palabras claves
    hlt1.AddIdentSpec('__file__', hlt1.tkKeyword);
    hlt1.AddIdentSpec('__line__', hlt1.tkKeyword);
    hlt1.AddIdentSpec('array', hlt1.tkKeyword);
    hlt1.AddIdentSpec('and', hlt1.tkKeyword);
    hlt1.AddIdentSpec('break', hlt1.tkKeyword);
    //forma alternativa de declaración
    hlt1.AddIdentSpecList('case class const continue', hlt1.tkKeyword);
    hlt1.AddIdentSpecList('default die do double', hlt1.tkKeyword);
    hlt1.AddIdentSpecList('else elseif echo empty endfor endif', hlt1.tkKeyword);
    hlt1.AddIdentSpecList('endswitch endwhile eval exit extends', hlt1.tkKeyword);
    hlt1.AddIdentSpecList('function false float for', hlt1.tkKeyword);
    //forma alternativa de declaración
    hlt1.AddKeyword('global');
    hlt1.AddKeyword('highlight_file');
    hlt1.AddKeyword('highlight_string');
    hlt1.AddKeyword('if');
    hlt1.AddKeyword('int');
    hlt1.AddKeyword('include');
    hlt1.AddKeyword('integer');
    hlt1.AddKeyword('isset');
    hlt1.AddKeyword('list');
    hlt1.AddKeyword('new');
    hlt1.AddKeyword('object');
    hlt1.AddKeyword('old_function');
    hlt1.AddKeyword('or');
    hlt1.AddKeyword('print');
    hlt1.AddKeyword('real');
```

```
hlt1.AddKeyword('require');
hlt1.AddKeyword('return');
hlt1.AddKeyword('show_source');
hlt1.AddKeyword('static');
hlt1.AddKeyword('string');
hlt1.AddKeyword('switch');
hlt1.AddKeyword('true');
hlt1.AddKeyword('unset');
hlt1.AddKeyword('var');
hlt1.AddKeyword('while');
hlt1.AddKeyword('xor');
//crea tokens delimitados
hlt1.DefTokDelim('',' ', hlt1.tkString);
hlt1.DefTokDelim('"','"', hlt1.tkString);
hlt1.DefTokDelim('//',' ', hlt1.tkComment);
hlt1.DefTokDelim('/*','*/', hlt1.tkComment, tdMulLin);
hlt1.Rebuild; //reconstruye
end;
```

En esta sintaxis solo se define el tratamiento de tokens. Solo se usan los tipos de tokens que ya existen en el resaltador, no se usan tipos de tokens personalizados.



## 5.10 Definición de bloques

Los bloques se pueden entender como conjuntos de tokens consecutivos, con significado sintáctico, claramente definidos por delimitadores, que también son tokens.

La forma más sencilla para definir bloques es usando el método `AddBlock()`, que tiene la siguiente declaración:

```
function TSynFacilSyn.AddBlock(  
    dStart, dEnd: string;  
    showFold: boolean = true;  
    parentBlk: TFaSynBlock = nil;  
): TFaSynBlock;
```

La descripción de los parámetros es:

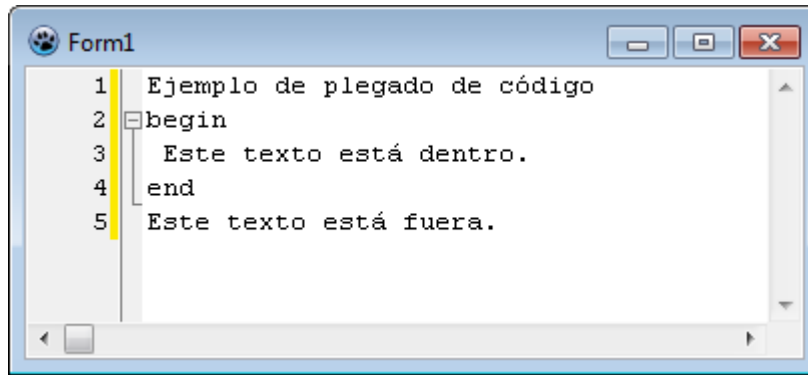
dStart	Obligatorio	Delimitador inicial del bloque. Debe ser un token especial.
dEnd	Obligatorio	Delimitador final del bloque. Debe ser un token especial.
showFold	Opcional	Indica si se mostrará la marca de plegado en el editor. Esta marca solo aparecerá cuando el bloque tenga más de una línea. Por defecto, está activado.
parentBlk	Opcional	Indica el bloque padre del bloque. Por defecto es NIL, es decir que es válido dentro de cualquier bloque o sección.

Un caso sencillo de definición de bloque sería:

```
hlt.AddBlock('begin', 'end');           //agrega el bloque  
if hlt.Err<>'' then ShowMessage(hlt.Err); //verifica el error  
...  
Rebuild;
```

El método `Rebuild()`, se debe ejecutar, siempre, después de ingresar todos los bloques, de otra forma podría no tener efecto, la definición.

Por defecto un bloque definido mostrará la marca de plegado:



El bloque se extenderá desde el token “BEGIN” hasta el fin el token “END”. El bloque puede distribuirse en una o más líneas. No es necesario especificarlo. Los bloques son por defecto multilínea.

Si quisiéramos que el bloque no muestre marca de plegado, deberíamos indicarlo en el tercer parámetro:

```
hlt.AddBlock('begin', 'end', false);
```

Los delimitadores de bloque pueden ser también símbolos. Por ejemplo, la siguiente sentencia es válida:

```
hlt.AddBlock('(', ')');
```

Esta definición, considerará a todos los tokens dentro de paréntesis, como parte de un bloque.

Los delimitadores de un bloque deben ser siempre identificadores o símbolos especiales. Si es que se define un delimitador de bloque que no se haya definido como token especial, se creará primero como token especial antes de procesar el bloque.

Por ejemplo, en la siguiente definición:

```
hlt.AddBlock('Inicio', 'Fin');
```

Si el identificador “Inicio”, no se ha creado como un identificador especial, entonces, primero se creará “Inicio”, como token especial, dentro del resaltador, con el atributo “tkIdentif”.

Si el identificador especial, ya existía, no se crea de nuevo ni se modifica su atributo.

Es importante notar que el resaltador intentará identificar siempre a los delimitadores del bloque como identificador especial o símbolo especial. La regla de identificación es sencilla:

- Si el primer carácter del delimitador (inicial o final) está en la lista de caracteres válidos de un identificador, se asume que todo el delimitador debe ser un identificador (Por reglas de tokens de tipo identificador). De no ser así se generará un error interno.

Por ejemplo el siguiente código dará un error al momento de intentar ejecutarlo, si es que el carácter %, no pertenece a los caracteres válidos para un identificador:

```
hlt.AddBlock('begin%', 'end');
```

- Si el primer carácter del delimitador no está en la lista de caracteres válidos de un identificador, se asume que todo el delimitador debe ser un símbolo. No se harán validaciones sobre la validez del delimitador como tal. Es responsabilidad del programador asegurar que así sea.

Considerar por ejemplo la siguiente definición:

```
hlt.AddBlock('"hey"', '"you"');
```

El resaltador intentará codificar “hey” (incluido las comillas) como símbolo especial, pudiendo dejar sin efecto alguna definición previa de cadenas.

Se pueden definir tantos bloques como se quiera, considerando que a mayor cantidad de ellos, es mayor el procesamiento que debe hacer el resaltador.

### 5.11 Bloques como objetos

Dentro del resaltador, los bloques son objetos de la clase “TFaSynBlock” que tienen la siguiente definición:

```
TFaSynBlock = class //clase para manejar los bloques de sintaxis
    name      : string; //nombre del bloque
    index     : integer; //indica su posición dentro de TFaListBlocks
    showFold  : boolean; //indica si se mostrará la marca de plegado
    parentBlk : TFaSynBlock; //bloque padre (donde es válido el bloque)
    BackCol   : TColor; //color de fondo de un bloque
```

```
IsSection    : boolean;    //indica si es un bloque de tipo sección
UniqSec      : boolean;    //indica que es sección única
end;
```

Un detalle importante es que un bloque no almacena referencia a los delimitadores. En la sección siguiente se explicará más sobre este aspecto.

Los bloques se crean en el resaltador, mediante el método AddBlock().

Cada vez que se usa AddBlock(), se está creando un nuevo objeto “TFaSynBlock”. El método AddBlock(), es en realidad, una función que devuelve una referencia al objeto “TFaSynBlock”, que representa al bloque.

Tener la referencia al objeto bloque, nos permite acceder a propiedades adicionales. En el siguiente código, se usa el acceso al objeto bloque para asignar un nombre al bloque:

```
var
  hlt      : TSynFacilSyn;    //objeto bloque
  bl_llaves : TFaSynBlock;
...
bl_llaves := hlt.AddBlock('{', '}');
bl_llaves.name := 'bloque_llaves'; //asigna un nombre
```

El nombre de un bloque, no es una propiedad importante cuando se configura el resaltador por código. Esta propiedad la usa el resaltador, cuando se configura la sintaxis con el archivo XML.

Existen dos bloques predefinidos en el resaltador, que se crean siempre. Estos son:

MainBlk -> Identifica al bloque principal. El que no tiene anidamientos.

MulTokBlk -> Se refiere a los bloques que forman todos los tokens multilínea que tienen activado el plegado.

Estos bloques, son bloques comunes, como cualquier otro, excepto que no necesitan crearse o destruirse. Con ellos se pueden cambiar algunas propiedades de la sintaxis.

El siguiente código permite desactivar el plegado de código de los tokens multilínea (sin eliminar los bloques). Algo que no se puede hacer por archivo XML, porque si no se activa el plegado, el token multilínea simplemente no crea bloques.

```
var
  hlt : TSynFacilSyn;
...
hlt.DefTokDelim('/*', '*/', tkComment, tdMulLin, true); //crea bloque multilínea
hlt.MulTokBlk.showFold:=false; //desactiva el plegado en tokens multilínea
```

Hay que notar que al cambiar las propiedades de “MulTokBlk”, se están cambiando las propiedades de todos los bloques que forman los tokens multilínea (que se hayan creado con plegado).

## 5.12 Bloques avanzados

El método AddBlock(), nos permite crear bloques de una forma sencilla, indicando un delimitador simple de inicio y uno de fin. Pero no nos permite mayor libertad para definir los delimitadores.

Existe una forma alternativa que nos permite mayor libertad. Esta consiste en usar el método de creación de bloques CreateBlock(), que tiene la siguiente declaración:

```
function CreateBlock(blkName: string;
  showFold: boolean=true;
  parentBlk: TFaSynBlock=nil): TFaSynBlock;
```

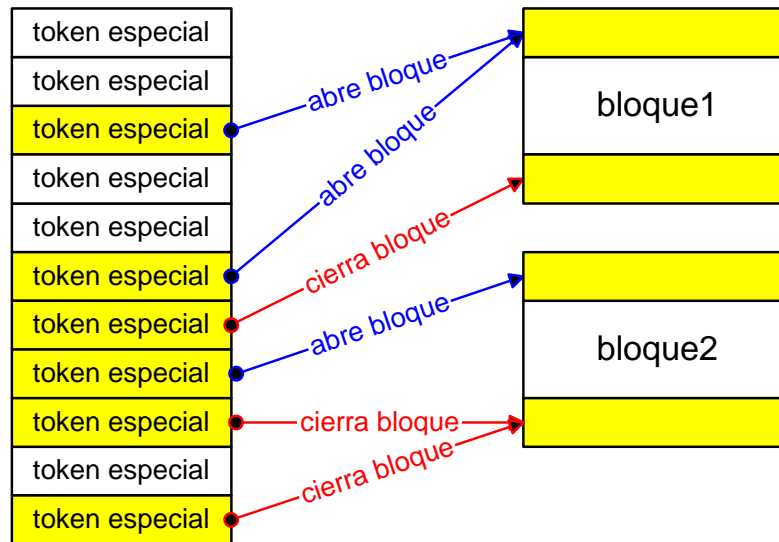
CreateBlock(), nos permite crear internamente un bloque en el resaltador (y sus propiedades principales), pero sin definir sus delimitadores.

¿Es posible tener un bloque sin delimitadores? Desde luego que sí. Sin embargo este bloque no se abrirá en la exploración normal de resaltador.

La forma común de trabajo del resaltador consiste en configurar a los tokens que sean delimitadores, para que guarden una referencia al bloque o sección que abren (pueden ser varios), y al bloque que cierran (pueden ser varios). Es la exploración de los tokens (no de bloques) los que determinan cuando se abre o cierra un bloque.

Los bloques están almacenados en la memoria del resaltador, pero ellos no tienen referencia a algún delimitador de inicio o fin (ya que puede haber varios tokens que los abran o cierren). Son los tokens delimitadores, los que guardan la referencia a los bloques que abren y cierran.

El siguiente diagrama ayudará a clarificar lo indicado:



Para asociar los delimitadores especiales, a un bloque, existen los métodos `AddIniBlockToTok()` y `AddFinBlockToTok()`, que tienen la siguiente declaración:

```
procedure AddIniBlockToTok(dStart: string; TokPos: integer;
                           blk: TFaSynBlock);
procedure AddFinBlockToTok(dEnd: string; TokPos: integer;
                           blk: TFaSynBlock);
```

Para apreciar la diferencia en la creación de bloques, consideremos la siguiente declaración de bloque:

```
blq_codigo := hlt.AddBlock('Inicio', 'Fin');
```

Y la forma alternativa equivalente, usando `CreateBlock()`:

```
blq_codigo := hlt.CreateBlock('');           //Crea bloque
//Asigna refer. de bloque al token 'Inicio', en su lista de "bloques que abre"
AddIniBlockToTok('Inicio', 0, blq_codigo);
//Asigna refer. de bloque al token 'Fin', en su lista de "bloques que cierra"
AddFinBlockToTok('Fin', 0, blq_codigo);
```

El método `AddIniBlockToTok()`, accede a la lista “bloIniL”, del token especial, para agregar la referencia del bloque indicado. Así, cuando se identifique a este token, se sabrá que es el inicio del bloque “blq\_codigo”. De la misma forma trabaja `AddFinBlockToTok()` pero con la lista “bloFinL”.

Esta nueva forma de definición de bloques, nos da mayor libertad para:

- Incluir el parámetro adicional “TokPos” en los delimitadores.
- Poder incluir múltiples delimitadores.

El parámetro “TokPos”, permite especificar mayor detalle con respecto a los delimitadores. Nos permite identificar completamente a un token especial que haya sido declarado con este parámetro (Ver Sección 4.5.1 y 4.4.1).

Por ejemplo, usar una definición como esta:

```
blq_codigo := hlt.CreateBlock('');           //Crea bloque
AddIniBlockToTok('Inicio', 2, blq_codigo);
```

Espera usar como delimitador inicial, a un identificador especial que se haya declarado de esta forma:

```
hlt.AddIdentSpec('Inicio', tkKeyword, 2);
```

Si no existiera esta definición, se procederá a crearla, porque el bloque requiere siempre que los delimitadores existan primero como tokens especiales.

La definición alternativa de delimitadores, nos permite también, definir más de un delimitador inicial o final, para un mismo bloque.

Por ejemplo si quisiéramos que el bloque BEGIN...END, se pueda abrir también con la palabra reservada INICIO, entonces la declaración sería:

```
blq_codigo := hlt.CreateBlock('');           //Crea bloque
AddIniBlockToTok('Begin', 0, blq_codigo);
AddIniBlockToTok('Inicio', 0, blq_codigo);
AddFinBlockToTok('End', 0, blq_codigo);
```

Cada llamada a `AddIniBlockToTok()`, permitirá definir un delimitador inicial adicional. Lo mismo sucede con `AddFinBlockToTok()`.

Los métodos `AddIniBlockToTok()` y `AddFinBlockToTok()`, hacen una verificación de la validez de los delimitadores, de la misma forma a cómo trabaja `AddBlock()`. Esto significa que es conveniente verificar siempre si se produce algún error en la definición de los delimitadores.

Un código típico de verificación, se vería así:

```
blq_codigo := hlt.CreateBlock(''); //Crea bloque
//Asigna refer. de bloque al token 'Inicio'.
AddIniBlockToTok('Inicio', 0, blq_codigo);
if Err<>' ' then ...; //Verifica Error
//Asigna refer. de bloque al token 'Fin'.
AddFinBlockToTok('Fin', 0, blq_codigo);
if Err<>' ' then ...; //Verifica Error
```

### 5.13 Coloreado de bloques

Como una característica adicional, el resaltador, permite colorear el fondo de los bloques para proporcionar una ayuda visual, de la extensión de los bloques.,

Para activar el coloreado de bloques, se debe configurar la propiedad “ColBlock” del resaltador. Los valores permitidos son:

cbNull	Desactiva el coloreado de bloques. Es la opción por defecto.
cbBlock	Colorea el fondo del bloque, por nivel. Muestra los bloques con un color más o menos oscuro, de acuerdo al nivel del bloque o sección.
cbLevel	Colorea el fondo del bloque, usando el color definido para cada bloque, en su parámetro “BackCol”. (Ver Sección 4.10 - Definición de Bloques)

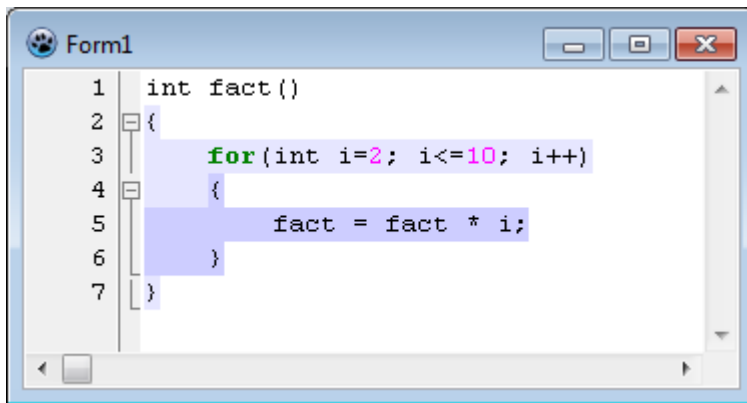
El siguiente código muestra, configura el coloreado del resaltador por nivel:

```
var
  hlt : TSynFacilSyn;
begin
  hlt := TSynFacilSyn.Create(self); //crea resaltador
  SynEdit1.Highlighter := hlt;
```



```
hlt.ColBlock:= cbLevel;      //configura coloreado
hlt.AddBlock('{', '}');      //crea bloque
hlt.Rebuild;
```

Cuando ponemos ColBlock en “cbLevel”, se procederá a realizar un coloreado automático de los bloques, usando el nivel de anidamiento de cada bloque. El siguiente ejemplo muestra cómo se vería un código sencillo de C, con coloreado de nivel:



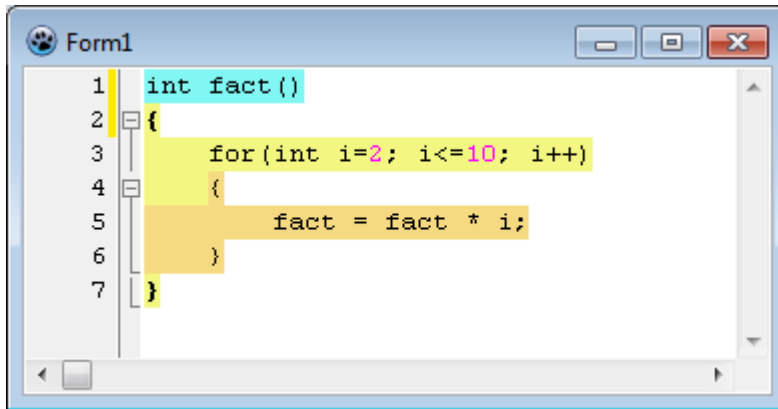
En este Estilo de coloreado, los bloques más internos, aparecerán con una tonalidad lila, tanto más oscuro, tanto más nivel de anidamiento tenga el bloque. El color base no es configurable por archivo o instrucciones, pero puede cambiarse modificando el método `TSynFacilSyn.GetTokenAttribute()` en el código del resaltador.

Observar que cuando se activa el coloreado de los bloques, los delimitadores del bloque se consideran también como parte del bloque y aparecen también coloreados.

Los bloques solo permiten cambiar el color de fondo de los tokens que se encuentren contenidos en el bloque (incluyendo los espacios). Cuando se activa el coloreado de bloques, se ignora la propiedad “Background” de los atributos de los tokens.

No es posible cambiar el color del texto o algún otro atributo visual, de los tokens dentro de un bloque. Estos, siempre corresponden a los que se haya definido en los atributos del resaltador.

El otro tipo de coloreado, es el coloreado por bloque. Si ponemos ColorBlock en “cbBlock”, el coloreado se hará usando como color de fondo, el color definido para cada bloque. La siguiente figura, muestra cómo se vería el mismo texto usando coloreado por bloques.



Este tipo de coloreado, no toma en cuenta el nivel de anidamiento, sino que usa el color personalizado que se especifica en la definición de cada bloque.

El color de fondo de un bloque se especifica en la propiedad “BackCol”, del bloque. Para cambiar esta propiedad, se debe tener la referencia al objeto bloque:

```
var
  hlt : TSynFacilSyn;
  bl_llaves: TFaSynBlock; //objeto bloque
begin
  hlt := TSynFacilSyn.Create(self); //crea resaltador
  SynEdit1.Highlighter := hlt;
  hlt.ColBlock:=cbBlock; //activa coloreado de bloques, individualmente.
  bl_llaves := hlt.AddBlock('{','}'); //crea bloque
  bl_llaves.BackCol:=clYellow; //fija color de fondo del bloque.
  hlt.Rebuild; //reconstruye
```

Si no se quiere especificar un color particular para un bloque, sino que se quiere usar el color del bloque anterior, se debe usar la constante: COL\_TRANSPAR:

```
bl_llaves.BackCol:=COL_TRANSPAR; //fija color transparente.
```

Hay que anotar, que usar el coloreado por bloque, requiere un trabajo de procesamiento mayor en el resaltador, que cuando se usa resaltado por nivel.

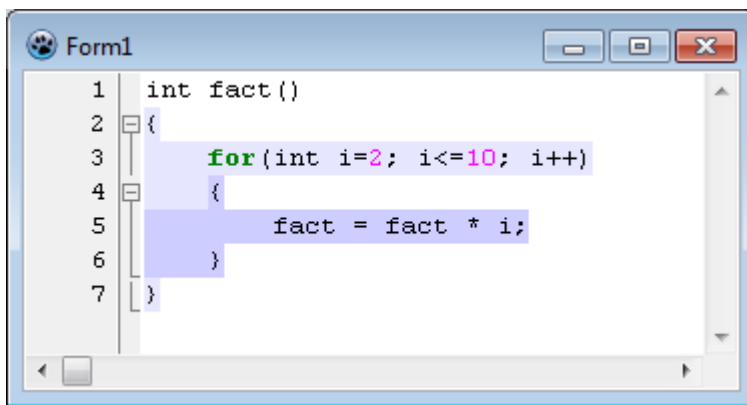
### 5.14 Alcance de un bloque

La definición de un bloque es por defecto, válida en cualquier parte del texto. Así, si definimos el bloque con:

```
bl_llaves := hlt.AddBlock('{', '}');
```

Esta definición será siempre válida, es decir, cada vez que el resaltador encuentre el símbolo especial “{”, procederá a abrir el bloque “bl\_llaves”, y de la misma forma, cada vez que encuentre el símbolo “}”, procederá a cerrar el bloque.

De modo que si escribimos un código de C con esta sintaxis, obtendremos la siguiente imagen:



En la imagen se puede ver claramente que los dos bloques han sido reconocidos, en la sintaxis, a pesar de que uno de ellos se encuentra dentro del otro. Podríamos crear innumerables bloques anidados de esta forma.

Esto es así, porque en la definición del bloque, no hemos especificado el alcance del bloque en el parámetro “parentBlk”, y este ha tomado el valor NIL, que significa que el bloque es válido en cualquier parte en donde aparezca.

El mismo resultado hubiéramos obtenido, si hubiéramos puesto explícitamente que el bloque es válido en cualquier parte, usando la siguiente forma:

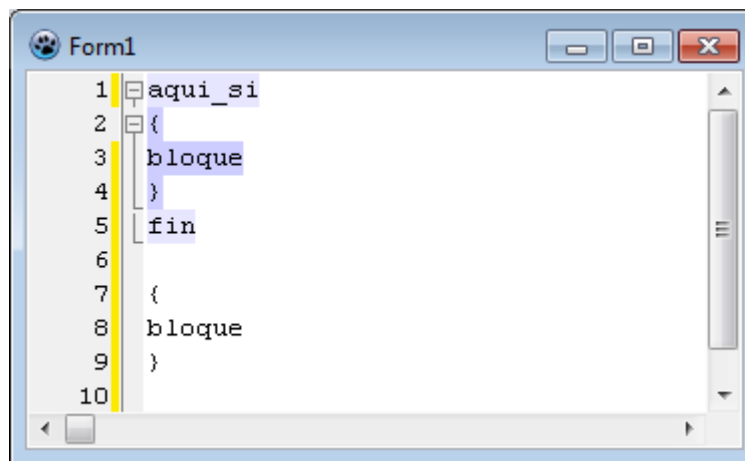
```
bl_llaves := hlt.AddBlock('{', '}', true, nil);
```

Usando el parámetro “parentBlk”, es posible limitar la validez de un bloque, a áreas específicas del código, que pueden ser otros bloques o secciones.

Así por ejemplo, la siguiente definición crea primero el bloque “aquí\_si”, y luego define el bloque “bl\_llaves”, para que solo sea válido, dentro de este bloque:

```
aqui_si := hlt.AddBlock('aqui_si','fin');  
bl_llaves := hlt.AddBlock('{','}',true, aqui_si);
```

Un código de ejemplo nos mostrará, el efecto de esta definición:

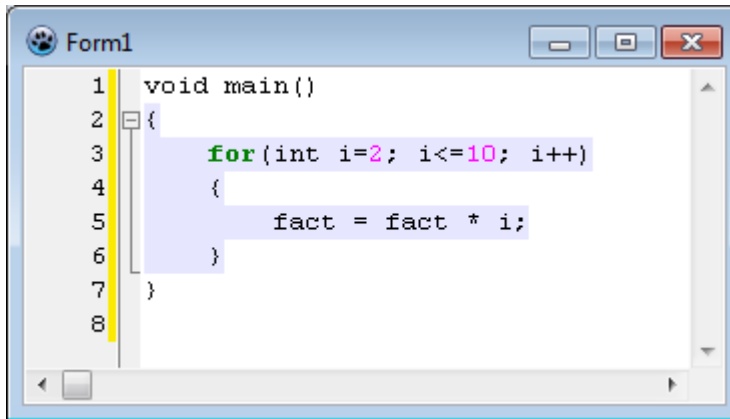


Como se aprecia, el segundo bloque, no es reconocido, porque se encuentra fuera del bloque “aquí\_si”.

Hay que tener cuidado, con restringir un bloque a un rango. Sobre todo, los bloques que puedan aparecer anidados. Por ejemplo si definimos el bloque “bl\_llaves”, para que solo sea válido en el bloque principal:

```
bl_llaves := hlt.AddBlock('{','}',true, hlt.MainBlk);
```

Podríamos tener el siguiente resultado:

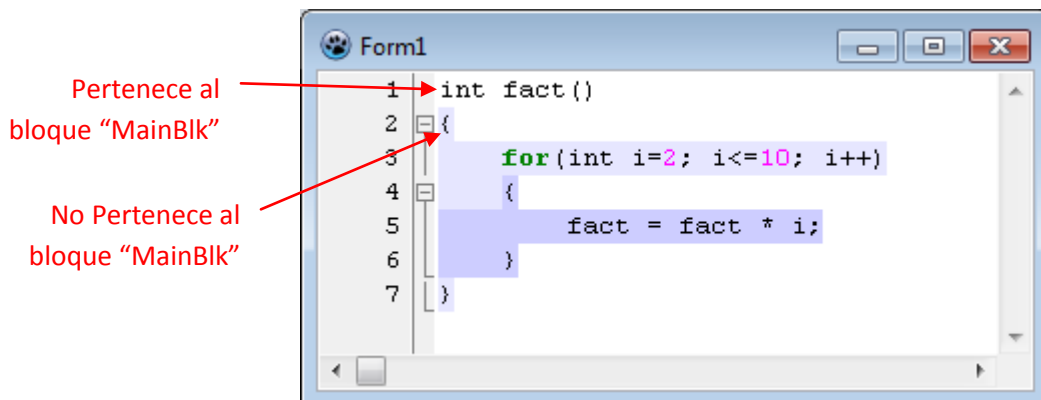


```
1 void main()
2 {
3     for(int i=2; i<=10; i++)
4     {
5         fact = fact * i;
6     }
7 }
8
```

Y notaremos, que la segunda llave “{”, encontrada, no abre otro bloque, porque no está dentro del bloque principal. Sin embargo la primer llave “{”, encontrada, sí se reconoce como el delimitador final del bloque, excluyendo una parte del código y dando lugar a una definición errónea sintácticamente.

El bloque “MainBlk”, como se recordará, es un bloque predefinido en el resaltador, que incluye a todo el archivo. Todos los tokens y bloques (o secciones), que se creen estarán contenidos directa o indirectamente en el bloque “Main”.

El interior de cualquier bloque, ya no es parte del bloque “MainBlk”. Solo los tokens o estructuras que no están dentro de algún otro bloque, pertenecen directamente al bloque “MainBlk”:



```
1 int fact()
2 {
3     for(int i=2; i<=10; i++)
4     {
5         fact = fact * i;
6     }
7 }
```

Pertenece al bloque “MainBlk”

No Pertenece al bloque “MainBlk”

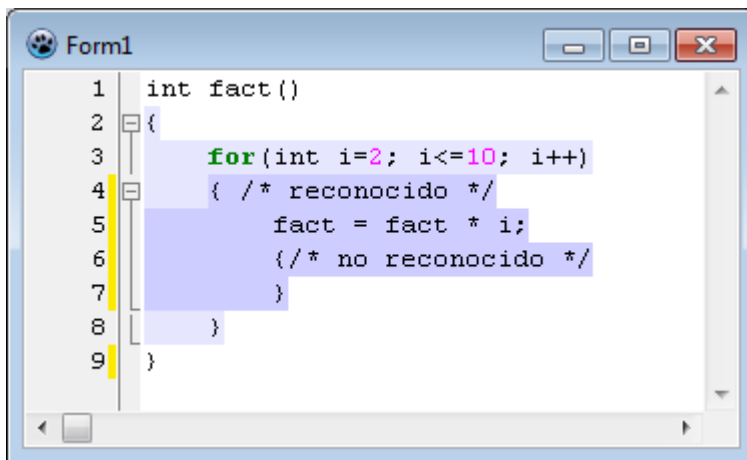
Cuando se crean bloques que solo sean válidos, en el bloque “MainBlk”, se estarán evitando que estos se puedan anidar y que solamente puedan aparecer en el cuerpo principal.

Es posible limitar el nivel de anidamiento de un bloque. Para ello se debe especificar en la estructura, cada bloque que corresponde a cada nivel.

El siguiente ejemplo, es una definición para permitir solo dos niveles de llaves:

```
//bloque válido en "MainBlk"
bl_llaves := hlt.AddBlock('{', '}', true, hlt.MainBlk);
//bloque válido en "bl_llaves"
bl_llaves2 := hlt.AddBlock('{', '}', true, bl_llaves);
```

Ahora con esta definición, se permitirá incluir un bloque "{", dentro de otro, pero no más. Así, si se incluyera un tercer anidamiento, la sintaxis fallará:



Observar que, para que esta definición funcione correctamente, es necesario que "bl\_llaves", esté definido solamente en el bloque principal "MainBlk". De otra forma no tendrá efecto, el límite de anidamientos que deseamos.

### 5.15 Definición de secciones

Se definen con el método AddSection(), que tiene la siguiente declaración:

```
function TSynFacilSyn.AddSection(  
  dStart: string;  
  showFold: boolean = true;  
  parentBlk: TFaSynBlock = nil;  
) : TFaSynBlock;
```

La descripción de los parámetros es:

dStart	Obligatorio	Delimitador inicial de la sección. Debe ser un token especial.
showFold	Opcional	Indica si se mostrará la marca de plegado en el editor. Esta marca solo aparecerá cuando el bloque tenga más de una línea. Por defecto, está activado.
parentBlk	Opcional	Indica el bloque padre del bloque. Por defecto es NIL, que significa que la sección es válida dentro del bloque principal.

Los parámetros de una sección son similares a los parámetros de un bloque, excepto que no existe el delimitador Final, porque los bloques tienen otra mecánica para determinar su fin.

Otra diferencia importante, es que el valor por defecto NIL, del “parentBlk”, no significa “válido en cualquier bloque”, como sucede en AddBlock(). Aquí NIL significa que la sección es válida solamente en el bloque principal. No se puede definir una sección que sea válida en cualquier parte, porque esto implicaría poder tener la misma sección anidada, lo cual va en contra de la definición de secciones.

Como las secciones son también bloques, la mayoría de reglas de los bloques, se aplican también aquí. También el coloreado de secciones es similar al coloreado de los bloques.

Un caso sencillo de definición de bloque sería:

```
hlt.AddSection('implementation');  
...  
hlt.Rebuild;
```

Esta sección se extenderá desde después del token “interface” hasta el fin del programa, si es que no se define alguna otra sección.

El método Rebuild(), se debe ejecutar, siempre, después de ingresar todas las secciones y bloques, de otra forma podría no tener efecto, la definición.

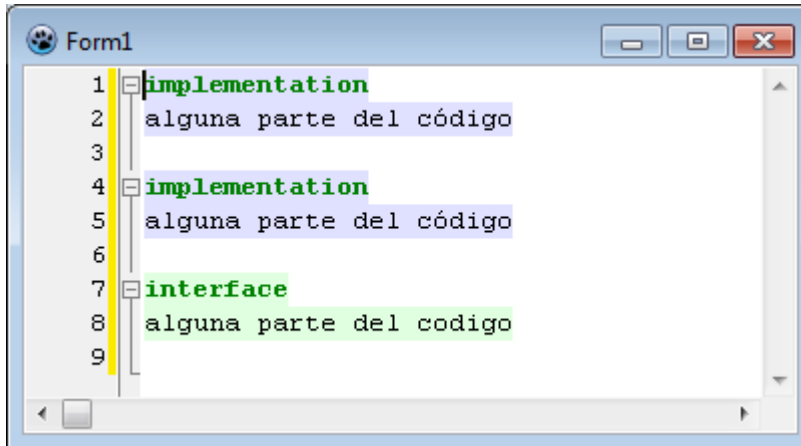
Se pueden definir tantas secciones como se quiera, considerando que a mayor cantidad de ellas, es mayor el procesamiento que debe hacer el resaltador.

Si definimos dos secciones, entonces estaríamos dividiendo el código en dos o más partes:

```
hlt.AddSection('interface');  
hlt.AddSection('implementation');
```

Ahora tendremos que una sección “interface” puede terminar en el inicio de una sección “implementation”, si es que existiera una.

La siguiente figura muestra un código de ejemplo, con las secciones creadas:



El orden en que se definen las secciones no es importante. No implica, un orden en el reconocimiento de las secciones. En nuestro ejemplo las secciones han sido correctamente reconocidas, e inclusive, aparece una sección duplicada.

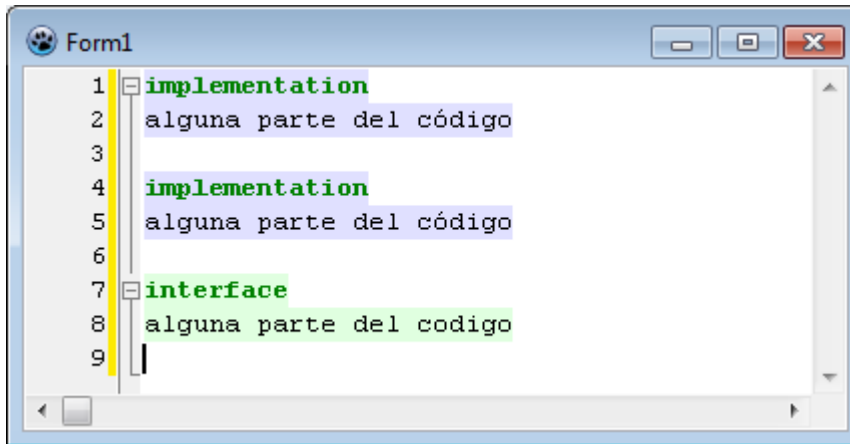
Para evitar que una misma sección, sea reconocida dos o más veces consecutivas, se debe usar la propiedad “UniqSec” del bloque. Si se activa, no se permitirá que la sección se repita después de ella misma. Es decir que solo se reconocerá la primera vez que aparezca.

En nuestro ejemplo no permitir la misma sección de forma consecutiva, tendríamos que hacer:

```
var
  sec1, sec2: TFaSynBlock;
...
sec1 := hlt.AddSection('interface');
sec2 := hlt.AddSection('implementation');
sec1.UniqSec:=true;
sec2.UniqSec:=true;
```

El efecto sería este:





Como se puede ver, solo existe una sección “implementation” (ver marca de plegado), ya que la segunda no se considerará como una nueva sección. Sin embargo, si es posible definir nuevamente la misma sección “implementation”, después de una sección “interface”.

Las secciones suelen usarse, para representar bloques de código especiales en un lenguaje de programación, como por ejemplo las secciones “interface” e “implementation” de Pascal. Estas estructuras de lenguaje no se adaptan bien, para manejarlas como bloques.

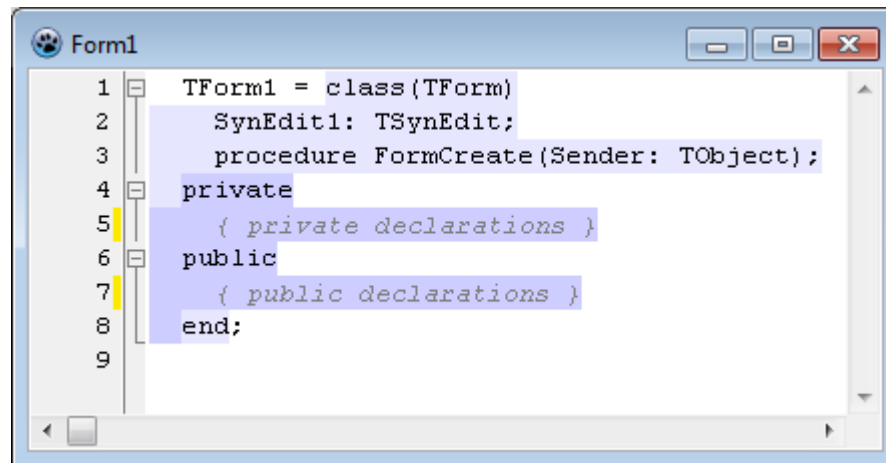
Los ejemplos vistos anteriormente, trabajaban solamente dentro del bloque principal, pero las secciones pueden trabajar también, dentro de otros bloques específicos. Así se permite segmentar regiones de código, dentro de estos bloques.

Como ejemplo consideremos la sintaxis de declaración de clases en Pascal. En ella, pueden existir las secciones “private”, “public” y “protected”.

Para definir estas secciones dentro de la declaración de una clase, primero definimos un bloque que incluya a la clase, y luego las secciones internas.

```
bloClas := hlt.AddBlock('class', 'end');  
hlt.AddSection('private', true, bloClas);  
hlt.AddSection('public', true, bloClas);  
hlt.AddSection('protected', true, bloClas);
```

Un texto de declaración de clase, se vería como se muestra en la siguiente figura:



## 5.16 Primera Sección

Por lo general las secciones empiezan después de que ha empezado el bloque que lo contiene. Sin embargo, podría ser que necesitamos abrir una sección al mismo tiempo que se abre un bloque.

Esto es lo que se llama la “Primera Sección”. Para entender este concepto, consideremos la estructura del siguiente texto:

```
SELECT campo1,  
campo2  
FROM  
tabla1, tabla2  
;
```

La tarea aquí, sería representar mediante bloques y secciones, la estructura de esta consulta SQL, de modo que tenga dos partes plegables. La primera parte sería desde la palabra SELECT, hasta antes de la palabra FROM, y la otra parte sería desde la palabra FROM hasta el final de la consulta.

La primera solución podría ser simplemente, definir dos secciones independientes que inicien con las palabras SELECT y FROM. Pero si deseáramos mantener la unidad sintáctica de la consulta completa, deberíamos incluirla en un bloque.

Como solución plantearemos la opción de definir un bloque que empiece con la palabra Reservada SELECT y termine con el punto y coma:

```
bloSelect:= hlt.AddBlock('SELECT',';');
```

De esta forma manejamos toda la consulta como un bloque. Pero ahora necesitamos dividirla en dos secciones. La sección FROM, no ofrece ningún problema, porque es una sección común.

Sin embargo, la primera sección, la que empieza con SELECT, no se puede definir directamente porque SELECT, es también el delimitador inicial del bloque. Si intentáramos definirla así:

```
bloSelect:= hlt.AddBlock('SELECT',';');  
hlt.AddSection('SELECT', true, bloSelect);  
hlt.AddSection('FROM', true, bloSelect);
```

La sección SELECT, no se abriría al inicio de la consulta, porque la verificación de secciones, se hace después de reconocer la palabra SELECT.

Para poder abrir el bloque, y a la vez una sección, debemos configurar una sección especial en el token delimitador. Esta es la llamada “Primera Sección”.

Para crear un Primera Sección, se debe usar el método AddFirstSection(), que es similar a AddSection(), en su declaración:

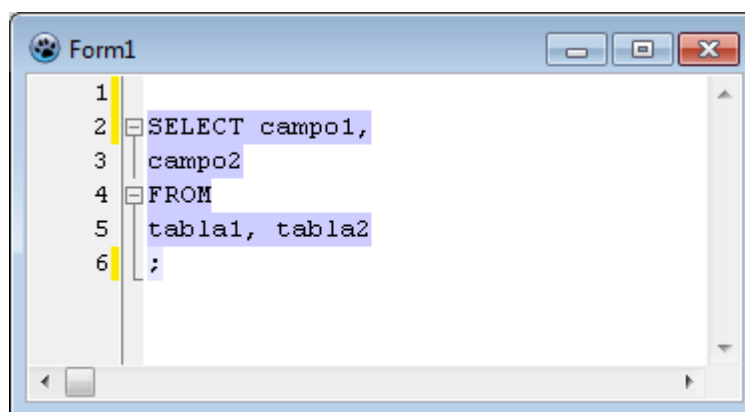
```
function TSynFacilSyn.AddFirstSection(  
    dStart: string; showFold:  
    boolean = true;  
    parentBlk: TFaSynBlock = nil): TFaSynBlock;
```

Con AddFirstSection(), podemos indicarle al resaltador, que el token indicado (que debe abrir un bloque), abrirá también la sección indicada.

De esta forma, la solución a nuestro problema, estaría en el siguiente código:

```
bloSelect:= hlt.AddBlock('SELECT',';');  
hlt.AddFirstSection('SELECT', true, bloSelect);  
hlt.AddSection('FROM', true, bloSelect);
```

Ahora, cuando se abra el bloque “bloSelect”, se abrirá una sección. El efecto es similar al mostrado:



Ahora se tienen dos secciones plegables dentro de un bloque, que también es plegable, pero que no es accesible con el ratón, por quedar “debajo”, de la marca de plegado de la sección.

## 5.17 Secciones avanzadas

El método `AddSection()`, nos permite crear secciones de una forma sencilla, indicando un delimitador simple de inicio. Pero no nos permite mayor libertad para definir este delimitador.

Existe una forma alternativa que nos permite mayor libertad. Esta consiste en usar el método de creación de bloques `CreateBlock()`, que tiene la siguiente declaración:

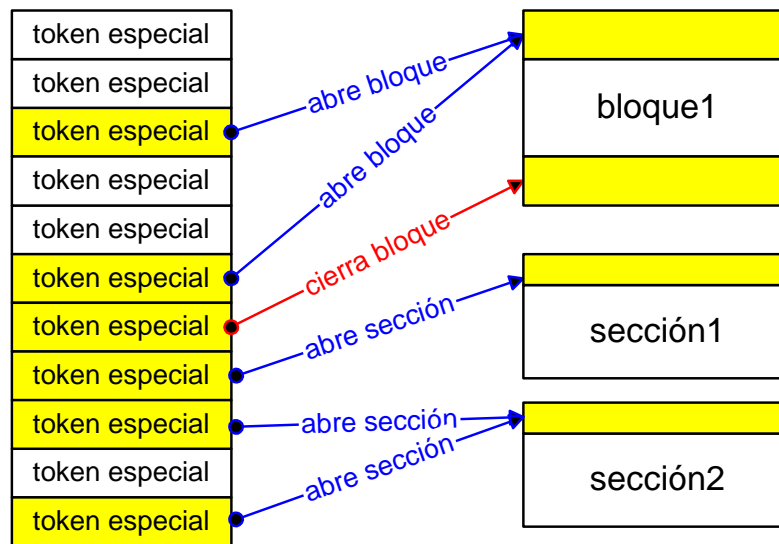
```
function CreateBlock(blkName: string;  
                    showFold: boolean=true;  
                    parentBlk: TFaSynBlock=nil): TFaSynBlock;
```

`CreateBlock()`, nos permite crear internamente un bloque o sección en el resaltador (y sus propiedades principales), pero sin definir sus delimitadores.

La forma como trabaja el resaltador consiste en configurar a los tokens que sean delimitadores, para que guarden una referencia al bloque o sección que abren (pueden ser varios), y al bloque que cierran (pueden ser varios). Es la exploración de los tokens (no de bloques) los que determinan cuando se abre o cierra una sección.

Las secciones están almacenadas en la memoria del resaltador (compartiendo el lugar con los bloques), pero ellos no tienen referencia a algún delimitador de inicio o fin (ya que puede haber varios tokens que los abran o cierren). Son los tokens delimitadores, los que guardan la referencia a las secciones que abren y cierran.

El siguiente diagrama ayudará a clarificar lo indicado:



Técnicamente no hay tokens que guarden referencias de secciones que puedan cerrar. Las secciones se cierran de forma automática, cuando se encuentra otra sección en su mismo nivel o el fin del bloque que la contiene (Ver 3.7 - Secciones).

Para asociar delimitadores especiales, a una sección, existe el método `AddIniSectToTok()`, que tiene la siguiente declaración:

```
procedure AddIniSectToTok(dStart: string; TokPos: integer; blk: TFaSynBlock);
```

Para apreciar la diferencia en la creación de secciones, consideremos la siguiente declaración:

```
sec1 := hlt.AddSection('interface', true, hlt.MainBlk);
```

Y la forma alternativa equivalente, usando `CreateBlock()`:

```
sec1 := hlt.CreateBlock('', true, hlt.MainBlk); //Crea bloque
sec1.IsSection := true; //marca como sección
//Asigna sección al token 'interface', en su lista de "secciones que abre"
AddIniSectToTok('interface', 0, sec1);
```

Aquí podemos apreciar mejor que nunca, que una sección es un simple bloque, pero con la propiedad "IsSection", en TRUE. Esto hará que el resaltador trate al bloque de una forma diferente al de los bloques comunes.

IMPORTANTE. Al crear secciones con AddBlock, se deben tener en cuenta dos consideraciones:

- Poner siempre la propiedad IsSection a TRUE.
- Especificar siempre el bloque padre. Dejar el valor por defecto (NIL), hará que la sección no se reconozca

El método AddIniSectToTok(), accede a la lista "seclNIL", del token especial, para agregar la referencia del bloque indicado. Así, cuando se identifique a este token, se sabrá que es el inicio de la Sección "sec1".

Esta nueva forma de definición de bloques, nos da mayor libertad para:

- Incluir el parámetro adicional "TokPos" en el delimitador de inicio.
- Poder incluir múltiples delimitadores de inicio.

El parámetro "TokPos", permite especificar mayor detalle con respecto al delimitador de inicio. Nos permite identificar completamente a un token especial que haya sido declarado con este parámetro (Ver Sección 4.5.1 y 4.4.1).

Por ejemplo, usar una definición como esta:

```
sec_clase := hlt.CreateBlock(' ', true, hlt.MainBlk); //Crea bloque
sec_clase.IsSection := true; //marca como sección
AddIniSectToTok('private', 1, sec_clase);
```

Espera usar como delimitador inicial, a un identificador especial que se haya declarado de esta forma:

```
hlt.AddIdentSpec('private', tkKeyword, 1);
```

Si no existiera esta definición, se procederá a crearla, porque el bloque (sección) requiere siempre que el delimitador exista primero como token especial.

La definición alternativa de delimitadores, nos permite también, definir más de un delimitador inicial, para una misma sección.

Por ejemplo si quisiéramos que la sección anterior, se pueda abrir también con la palabra reservada “private2”, entonces la declaración sería:

```
sec_clase := hlt.CreateBlock(' ', true, hlt.MainBlk); //Crea bloque
sec_clase.IsSection := true; //marca como sección
AddIniSectToTok('private', 0, sec_clase);
AddIniSectToTok('private2', 0, sec_clase);
```

Cada llamada a AddIniSectToTok(), permitirá definir un delimitador inicial adicional. Puede haber tantos como se desee.

El método AddIniSectToTok(), hace una verificación de la validez del delimitador, de la misma forma a cómo trabaja AddSection(). Esto significa que es conveniente verificar siempre si se produce algún error en la definición del delimitador.

Un código típico de verificación, se vería así:

```
sec_clase := hlt.CreateBlock(); //Crea bloque
sec_clase.IsSection := true; //marca como sección
AddIniSectToTok('private', 0, sec_clase);
if Err<>' ' then ...; //Verifica Error
```

Este método de creación de secciones, también permite definir “Primeras Secciones”, usando el método: AddFirstSectToTok(), tal como lo hace AddFirstSection().



## 5.18 Obteniendo información sobre los bloques

El resaltador TSynFacilSyn, permite definir bloques que se pueden colorear o plegar. Pero para obtener mejores beneficios del resaltador, es necesario acceder a la información interna sobre estos bloques.

Para ello, se han definido los siguientes métodos:

### 5.18.1 SetHighlighterAtXY()

Tiene la siguiente declaración:

```
function SetHighlighterAtXY(XY: TPoint): boolean;
```

Este método permite ubicar al resaltador en una posición específica del texto. Es como si se estuviera haciendo la exploración normal del texto y se detiene en la posición indicada.

Esto es útil por cuanto nos permite leer el estado del resaltador cuando ha llegado a cualquier parte del texto. Así podemos ver información sobre los tokens y los bloques.

El parámetro XY, indica la posición del texto en donde se desea poner al resaltador. Son valores que empiezan en 1, para la primera fila o para la primera columna. El siguiente ejemplo muestra cómo se puede poner al resaltador en la posición en donde se encuentra actualmente el cursor de un editor SynEdit1:

```
SynFacilSyn1.SetHighlighterAtXY(SynEdit1.CaretXY); //posiciona  
//a partir de aquí se puede obtener información sobre el resaltador  
tok := SynFacilSyn1.GetToken; //lee el token actual
```

Este ejemplo aprovecha para leer el token debajo del cursor, pero esto mismo se pudo haber hecho con TSynEdit.GetHighlighterAttriAtRowCol(). Sin embargo SetHighlighterAtXY(), solo posiciona al resaltador, a partir de allí, se puede leer toda la información disponible. GetHighlighterAttriAtRowCol(), solo da información sobre el token y el atributo. Además SetHighlighterAtXY(), está mejor optimizado para TSynFacilSyn y es, por lo tanto, más rápido.

Internamente SetHighlighterAtXY(), hace una pequeña exploración, pero solo en la línea de trabajo. No Es necesario explorar todo el texto, porque la información del estado del resaltador, se guarda internamente para cada línea.

### 5.18.2 NestedBlocks()

Tiene la siguiente declaración:

```
function NestedBlocks: Integer;
```

Esta función devuelve la cantidad de bloques abiertos actualmente. Este valor se puede leer en cualquier momento, pero por lo general se usará después de llamar a SetHighlighterAtXY(), para ver los bloques abiertos en una posición cualquiera del texto.

El siguiente ejemplo muestra como leer el nivel de anidamiento en la posición del cursor:

```
SynFacilSyn1.SetHighlighterAtXY(SynEdit1.CaretXY); //posiciona  
nivel := SynFacilSyn1.NestedBlocks;           //lee el anidamiento
```

Existe, en la clase TSynCustomFoldHighlighter, el método FoldBlockEndLevel(), que también devuelve la cantidad de bloques abiertos, pero solo da esta información para el final de una línea cualquiera y no considera a todos los bloques sino, solo los que tienen plegado de código.

### 5.18.3 NestedBlocksBegin()

Tiene la siguiente declaración:

```
function NestedBlocksBegin(LineNumber: integer): Integer;
```

Esta función devuelve la cantidad de bloques abiertos al inicio de la línea indicada.

A diferencia de NestedBlocks(), este método no necesita posicionarse primero en el texto, usando SetHighlighterAtXY(). Simplemente se le debe pasar como parámetro el número de línea y la función devolverá el número de bloques abiertos con que se inicia la exploración de una línea.

Este valor se obtiene leyendo el estado del final de la línea para la línea anterior, accediendo a la información de rangos.

### 5.18.4 TopCodeFoldBlock()

Tiene la siguiente declaración:

```
function TopCodeFoldBlock(DownIndex: Integer): TFaSynBlock;
```

Esta función devuelve el último bloque abierto en el resaltador. El parámetro “DownIndex”, indica el nivel de profundidad del bloque solicitado. Cuando es 0, se refiere al último bloque abierto; cuando es 1 se refiere al penúltimo, y así sucesivamente.

Esta misma información se podría obtener con el método TSynCustomFoldHighlighter.TopCodeFoldBlockType(), pero éste es inaccesible desde fuera de la clase, además TopCodeFoldBlock(), devuelve un objeto TFaSynBlock directamente.

Cuando el bloque actual es el bloque principal, es decir, que no se ha abierto ningún bloque, TopCodeFoldBlock() devuelve NIL.

El siguiente código muestra cómo leer el nombre del último bloque abierto, en la posición del cursor:

```
SynFacilSyn1.SetHighlighterAtXY(SynEdit1.CaretXY); //posiciona  
blk := TopCodeFoldBlock(0); //lee bloque actual  
if blk <> nil then nombre_bloque := blk.name;
```

### 5.18.5 GetBlockInfoAtXY()

Tiene la siguiente declaración:

```
function GetBlockInfoAtXY(XY: TPoint; out blk: TFaSynBlock; out level: integer):  
boolean;
```

Este método permite obtener el último bloque abierto, para una posición cualquiera del texto. Además devuelve la cantidad de bloques abiertos en esa posición, en el parámetro “level”.

Su trabajo es sencillo, pues solo ubica al resaltador en la posición indicada, y allí lee la información del último bloque, usando TSynCustomFoldHighlighter.TopCodeFoldBlockType().

Es una forma resumida para acceder al bloque activo en una posición del texto.

El parámetro XY, indica la posición del texto de donde se desea leer el bloque. Son valores que empiezan en 1, para la primera fila o para la primera columna.

El siguiente ejemplo muestra cómo se puede leer el nombre del bloque actual, donde se encuentra el cursor de un editor SynEdit1:

```
var
  blq: TFaSynBlock;
  nivel: integer;
...
SynFacilSyn1.GetBlockInfoAtXY(SynEdit1.CaretXY, blq, nivel);
if blq = nil then
  debugln('Bloque=NIL, nivel='+IntToStr(nivel))
else begin
  debugln('Bloque=' + blq.name + ', nivel='+IntToStr(nivel));
end;
```

### 5.18.6 ExploreLine()

Tiene la siguiente declaración:

```
function ExploreLine(XY: TPoint; out toks: TATokInfo; out CurTok: integer):
boolean;
```

Este método realiza una exploración de la línea indicada en “XY”, y lee información detallada de los tokens de esa línea, indicando además cuál es el token que se encuentra en la posición “XY”.

El tipo TATokInfo, es un arreglo de elementos de tipo TFaTokInfo:

```
TFaTokInfo = record
  txt      : string;           //texto del token
  TokPos   : integer;         //posición del token dentro de la línea
  TokTyp   : TSynHighlighterAttributes; //atributo de token
  posIni   : integer;         //posición de inicio en la línea
  curBlk   : TFaSynBlock;     //referencia al bloque del token
end;

TATokInfo = array of TFaTokInfo;
```

La estructura TFaTokInfo, contiene toda la información relativa a un token, que se puede necesitar en una exploración típica del texto.

Después de una llamada a ExploreLine(), los tokens de la línea indicada en “XY”, se devuelven en el parámetro toks[]. Este arreglo empieza en el índice 0.

El parámetro CurTok, indica la posición dentro de toks[], que corresponde al token que se encuentra debajo de la coordenada “XY”. Así se puede ubicar fácilmente el token de la posición actual.

Esta función se diseñó pensando en facilitar la implementación de la opción de autocompletado, usando la información de tokens y bloques del resaltador.

Las funciones SearchBeginBlock(), SearchEndBlock(), son ampliaciones que aún están en desarrollo, así que no deberían usarse, todavía.

### ***5.19 Usando al resaltador como analizador léxico - sintáctico***

El resaltador TSynFacilSyn, no es solamente un resaltador para el componente SynEdit, sino que además es un excelente analizador léxico – sintáctico, configurable.

Se comporta como un analizador léxico, porque una vez configurado para una sintaxis, puede extraer cada uno de los tokens del texto fuente.

Un lazo de exploración para la extracción de tokens, se parecerá a este:

```
for lin in SynEdit1.lines do begin
  hlt.SetLine(lin,1);      //"hlt" es el resaltador
  while not hlt.GetEol do begin
    Token := hlt.GetToken; //lee el token
    TokenType := hlt.GetTokenKind; //lee atributo
    //hace algo con el token
    hlt.Next; //pasa al siguiente
  end;
end;
```

Si además añadimos la información de bloques, estaremos ya en el nivel sintáctico.

La función GetXY(), permite obtener la posición actual de exploración del resaltador. Está declarada como:

```
function GetXY: TPoint;
```

Para que trabaje correctamente, es necesario especificar el valor correcto del parámetro "LineNumber" cuando se llame a SetLine(). Este valor de línea empieza en 0, para la primera línea.

## 6 Para programadores

### 6.1 Categorización de tokens

Para entender la clasificación de los tokens en el resaltador, consideremos que iniciamos el resaltador sin ningún elemento predefinido.

```
hlt1.ClearMethodTables;  
hlt1.ClearSpecials;
```

Una sintaxis vacía solo consideraría tres tipos de tokens: símbolos, espacios y nulos. Como los tokens nulos solo se encuentran en el fin de línea, entonces, todos los demás se considerarían símbolos o espacios. Una forma de ver la distribución de tokens es:

Nulo	Símbolo
Espacio	

Los espacios tienen una definición fija y no se puede cambiar, por ello es que siempre existirá esta categoría, y es probable que en todo texto exista por lo menos un token de tipo espacio.

Los símbolos no se definen, se asumen que son los caracteres que no se pueden procesar como cualquier otro token. En un caso de sintaxis vacía, se considerarán como símbolos a todos los tokens que no sean espacios o nulos.

Al empezar a crear los tokens por contenidos y los delimitados, empezaremos a reducir la cantidad de caracteres que se consideren como símbolos. En el siguiente ejemplo, se muestra cómo sería la distribución inicial de tokens en un texto con el resaltador sin iniciar<sup>6</sup>:

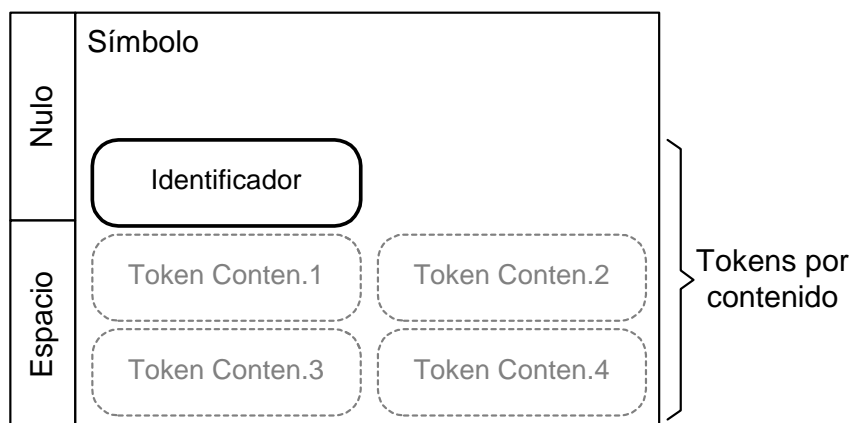
---

<sup>6</sup> En realidad, el resaltador crea una definición por defecto para los identificadores al iniciar.





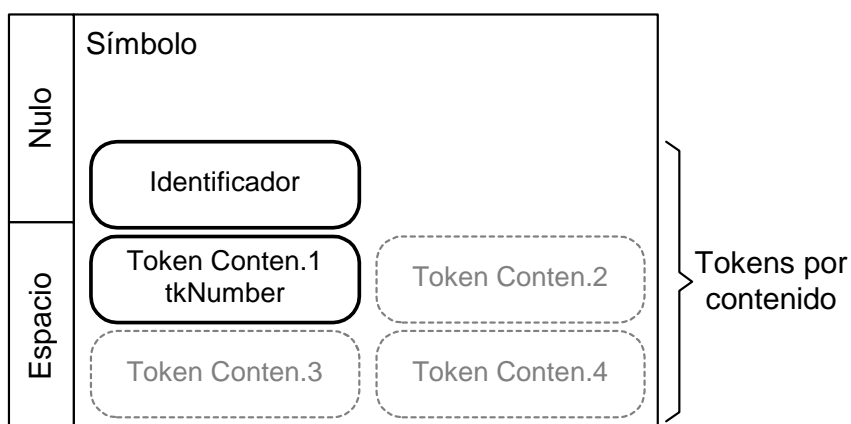
Cuando se definen identificadores, se usa el espacio especial reservado para los identificadores, y se empieza a considerar estos nuevos tipos de tokens.



Los tokens por contenido no usados, serán conjuntos vacíos, no se usaran ni se procesarán.

Cuando se definan tokens por contenido adicionales, como los números, se empezará a usar alguno de los espacios vacíos adicionales:

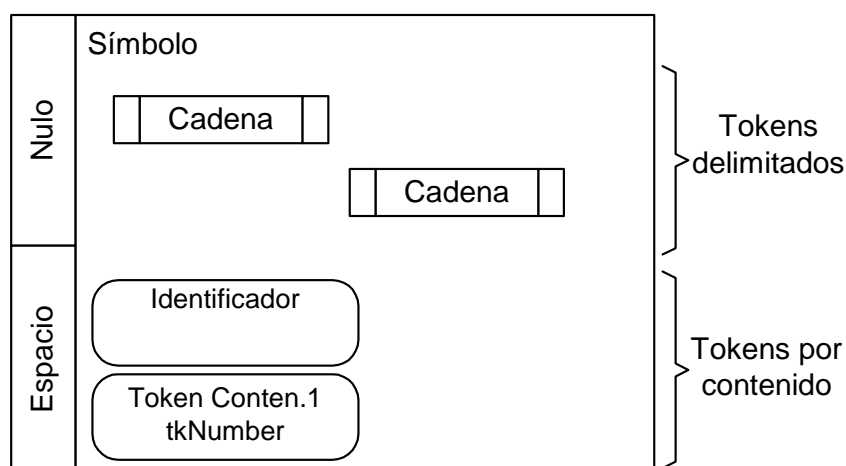
```
hlt1.DefTokContent('[0123456789]', '0123456789', tkNumber);
```



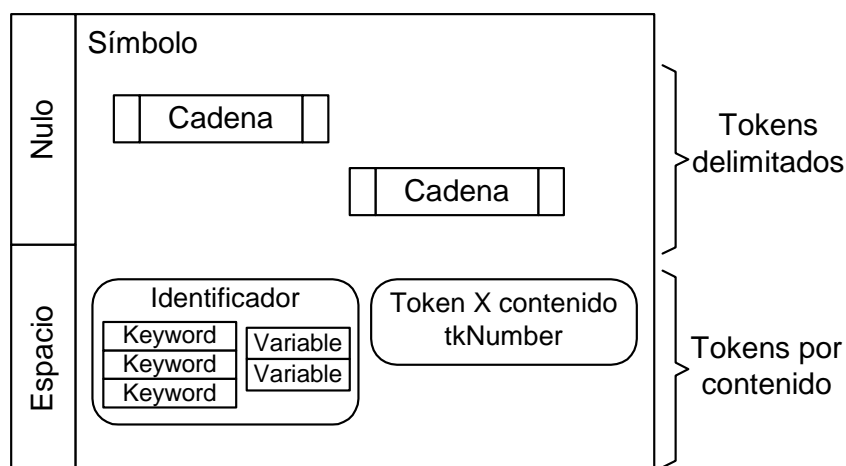
Podemos agregar también definiciones de tokens delimitados:

```
hlt.AddTokenString('\"', '\"', false, false);
hlt.AddTokenString('\"', '\"', false, false);
hlt.Rebuild;
```

Ahora el diagrama podría parecerse a este:



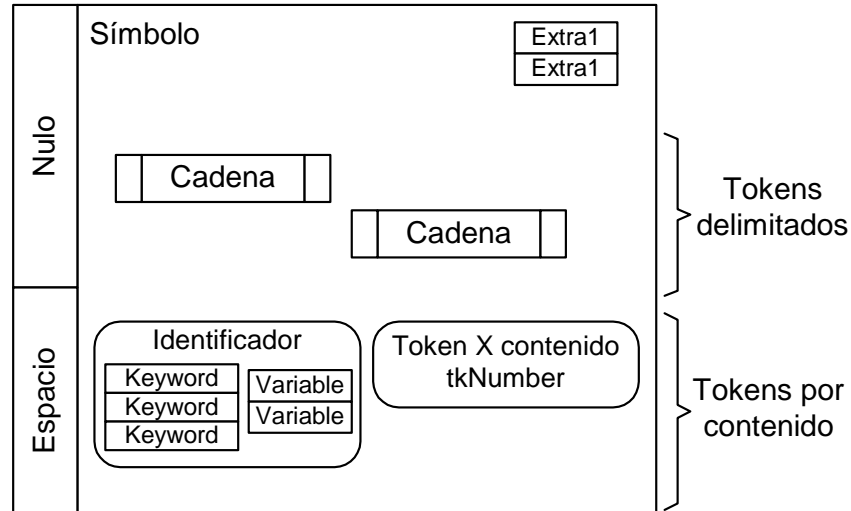
Si además definimos algunos identificadores especiales, entonces se crearán subconjuntos de identificadores con atributos distintos (identificadores especiales).



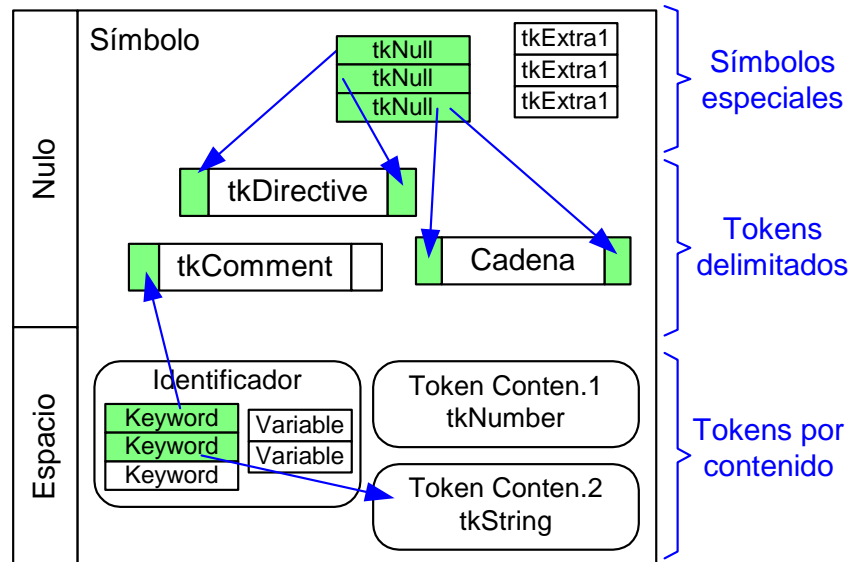
Solo el bloque de identificadores, soporta subconjuntos con atributo diferente. Los otros bloques por contenido o delimitados, no permiten manejar subconjuntos.

Pero si se puede crear subconjuntos de símbolos:

```
hlt.AddSymbSpec('+', tkExtra1);
hlt.AddSymbSpec('-', tkExtra1);
```



Conforme vayamos agregando más elementos a la sintaxis, iremos completando el diagrama. Una sintaxis más completa podría parecerse a la mostrada:



En este diagrama, la zona verde corresponde a los delimitadores, que como se puede ver, pueden ser identificadores o símbolos especiales. Los símbolos son elementos “sobrantes”, en la sintaxis, pero se les puede aplicar la definición y hacerlos delimitadores de token.

Todos los delimitadores deben ser símbolos o identificadores especiales. Un caso poco común es usar un identificador especial como delimitador de comentario, pero este caso existe en el lenguaje del DOS, en donde la palabra reservada REM especifica un comentario.

Cuando un símbolo o identificador especial, pasa a ser delimitador, pierde su naturaleza de token, y pasan más bien a ser partes de un token más grande.

## 6.2 Tratamiento de tokens

En el resaltador, los tokens por contenido, se ubican mediante el método de tabla de prefijos, usando el primer carácter de una cadena para determinar a que token pertenece.

Para ello se usa una tabla de 256 celdas con punteros a rutinas de procesamiento, llamada “Tabla de métodos”. Esta tabla tiene la siguiente declaración:

```
TProcTableProc = procedure of object;
...
TSynFacilSyn = class(TSynCustomHighlighter)
private
    ...
    fProcTable : array[#0..#255] of TProcTableProc; //tabla de métodos
    ...
end;
```

Esta tabla se modifica cuando se definen identificadores, tokens por contenido o tokens delimitados.

Para acelerar la búsqueda de identificadores especiales, se maneja una tabla por cada carácter inicial a usar. Actualmente se tienen las siguientes tablas:

```
//tablas para identificadores especiales
mA, mB, mC, mD, mE, mF, mG, mH, mI, mJ, mK, mL, mM, //para mayúsculas
mN, mO, mP, mQ, mR, mS, mT, mU, mV, mW, mX, mY, mZ,
mA_, mB_, mC_, mD_, mE_, mF_, mG_, mH_, mI_, mJ_, mK_, mL_, mM_, //para minúsculas
mN_, mO_, mP_, mQ_, mR_, mS_, mT_, mU_, mV_, mW_, mX_, mY_, mZ_,
m_, mDol, mArr, mPer, mAmp : TArrayTokEspec;
```

Los tokens delimitados, se detectan usando los delimitadores, que se guardan en la tabla mSym[], declarada así:

```
mSym : TArrayTokEspec; //tabla de símbolos especiales
```

TArrayTokEspec, es un arreglo de elementos de tipo TtokEspec:

```
TArrayTokEspec = array of TtokEspec;
```

Es una estructura que se usa para almacenar identificadores o símbolos, y que permite adicionalmente, almacenar información sobre las propiedades de un delimitador. No significa que todos los identificadores o símbolos especiales sean delimitadores, pero cuando lo son, aquí guardan sus propiedades.

La estructura TtokEspec, tiene la siguiente declaración:

```
TTokEspec = record
    txt      : string;           //palabra clave
    TokPos   : integer;         //posición del token dentro de la línea
    tTok     : TSynHighlighterAttributes; //tipo de token
    tipDel   : TFaTypeDel;      {indica si el token especial actual, es en realidad,
                                el delimitador inicial de un token delimitado}
    dEnd     : string;          //delimitador final (en caso de que sea delimitador)
    pRange   : TFaProcRange;    //procedimiento para procesar el token o rango (si
                                es multilinea)
    folTok   : boolean;         //indica si el token delimitado, tiene plegado
                                //propiedades para manejo de bloques y plegado de código
    bloIni   : boolean;         //indica si el token es inicio de bloque de
                                plegado
    bloIniL  : array of TFaSynBlock; //lista de referencias a los bloques que
                                abre
    bloFin   : boolean;         //indica si el token es fin de bloque de plegado
    bloFinL  : array of TFaSynBlock; //lista de referencias a los bloques que
                                cierra
    secIni   : boolean;         //indica si el token es inicio de sección de
                                bloque
    secIniL  : array of TFaSynBlock; //lista de bloques de los que es inicio
                                de sección
    firstSec : TFaSynBlock;     //sección que se debe abrir al abrir el bloque
end;
```

El campo “tipDel”, permite saber si el identificador o símbolo o es un delimitador, y de ser así, a qué tipo pertenece. El tipo “TFaTypeDelim” Tiene la siguiente declaración:

```
TFaTypeDelim = (tdNull,       //no es delimitado
                tdUniLin,      //es delimitador inicial de token delimitado de una línea
                tdMulLin,      //es delimitador inicial de token delimitado multilinea
                tdConten1,     //es delimitador inicial de token por contenido 1
                tdConten2,     //es delimitador inicial de token por contenido 2
                tdConten3,     //es delimitador inicial de token por contenido 3
                tdConten4);    //es delimitador inicial de token por contenido 4
```

Un token se considera como un texto con un solo atributo. Como un token puede ocupar una o varias líneas, se consideran tipos separados para acelerar el procesamiento.

Esta definición, está orientada al procesamiento de tokens delimitados. Pero también se aplica a los tokens por contenido. Hay una etiqueta para cada token por contenido disponible.

Los delimitadores de tokens se pueden agrupar de acuerdo a la siguiente tabla:

VALOR	TIPO DE TOKEN	CANTIDAD DE TOKENS	CANTIDAD DE LÍNEAS	ATRIBUTO
tdUniLin	Token uni-línea	1	1	Cualquiera
tdMulLin	Token de varias líneas	1*	1 o varias	Cualquiera
tdConten1	Token uni-línea	1	1	Cualquiera
tdConten2	Token uni-línea	1	1	Cualquiera
tdConten3	Token uni-línea	1	1	Cualquiera
tdConten4	Token uni-línea	1	1	Cualquiera

(\*) Cuando el token se extiende por varias líneas, se compondrá de varios tokens, debido a que el procesamiento de tokens se hace línea por línea.

Los comentarios y cadenas se consideran como simples tokens delimitados y pueden ser de una o varias líneas.

### 6.3 Manejo de Rangos

Para información general sobre el manejo de rangos, se recomienda leer “La Biblia del SynEdit- Rev5 – Sección 2.3.10” sobre el manejo de rangos.

A diferencia de un resaltador fijo, el resaltador “TSynFacilSyn” no puede tener una cantidad fija de rangos para manejar los elementos multilínea de la sintaxis.

Un resaltador fijo suele tener una definición como esta:

```
type
  TRangeState = (rsUnknown,
                 rsComment,  //comentario
```

```
rsComment2, //comentario
rsString); //cadena
```

Sin embargo, en nuestro resaltador, no existe tal definición para el manejo de rangos, porque puede haber muchos rangos distintos.

De hecho, cada vez que se define un token multilínea, se está creando un renglo diferente. En el resaltador, la variable “fRange”, no es un enumerado (como en la mayoría de resaltadores), sino que es un verdadero puntero:

```
fRange : ^TtokEspec; //para trabajar con tokens multilínea
```

La referencia a un registro TtokEspec, se hace para poder acceder a los campos del token especial, que define el rango. Es por ello que TtokEspec, tiene información completa sobre el bloque, cuando es un delimitador:

```
TtokEspec = record
  txt      : string;           //palabra clave
  TokPos: integer;           //posición del token dentro de la línea
  tTok  : TtkTokenKind;      //tipo de token
  tipDel: TFaTypeDel;        {indica si el token especial actual, es en realidad,
                             el delimitador inicial de un token delimitado}
  dEnd   : string;           //delimitador final (en caso de que sea delimitador)
  pRange: TFaProcRange;      //procedimiento para procesar el token o rango (si es
                             multilínea)
  folTok: boolean;           //indica si el token delimitado, tiene plegado
  ...
end;
```

De esta forma, cuando se empieza a explorar tokens multilíneas, se asigna a fRange, la referencia al registro del delimitador inicial:

```
procedure TSynFacilSyn.ProcTokenDelim(const d: TtokEspec);
...
  fRange := @d;
...
```

Cuando no se maneja rangos. fRange debe estar en NIL.

En el método Next(), se hace una verificación para ver si se está en medio de un rango, y de ser así, se usará la información del registro apuntado:

```
procedure TSynFacilSyn.Next;  
begin  
...  
  posIni := posFin;  //apunta al primer elemento  
  if fRange = nil then begin  
    carIni:=fLine[posFin]; //guardar para tenerlo disponible en el método que  
se va a llamar.  
    fProcTable[carIni];    //Se ejecuta la función que corresponda.  
  end  
  else begin  
    if posFin = tamLin then begin //para acelerar la exploración  
      fTokenID:=tkEol; exit;  
    end;  
    fTokenID:=fRange^.tTok;  
    fRange^.pRange;    //ejecuta método  
  end;  
end;
```

Este manejo “avanzado” de rangos, tiene una complicación adicional. Una de ellas es que se debe actualizar, necesariamente, la información de rangos en SynEdit, cuando se cambia la sintaxis. De otra forma tendríamos que los rangos de las líneas podrían estar apuntando a posiciones no existentes. Esta tarea se implementa fácilmente llamando a ScanAllRanges:

```
procedure TSynFacilSyn.Rebuild;  
...  
  Self.ScanAllRanges;  
...  
end;
```



## 6.4 Bloques y Secciones

Los bloques nacen como una necesidad para implementar la facilidad de plegado de código. Sin embargo, se definieron pensando en poder definir regiones especiales de significado sintáctico. Es decir, que los bloques del resaltador están orientados a representar los bloques sintácticos del propio lenguaje que manejan.

Esto significa que si configuramos al resaltador para trabajar con una sintaxis en C, entonces el bloque que va entre llaves, no solamente es una ayuda visual para poder aplicar el plegado de código, sino también que representa lógicamente a un conjunto de instrucciones, que son sintácticamente equivalentes a una sola, en el lenguaje C.

De esta forma se puede representar muchas estructuras propias del lenguaje en sí, usando bloques y secciones.

Las secciones con como una herramienta adicional, para potenciar la capacidad de crear regiones especiales de texto. Se definieron pensando en la estructura del lenguaje Pascal, pero son bastante generales, como para aplicarse a diversos lenguajes.

Si bien el concepto de “bloque”, dentro del resaltador TSynFacilSyn es similar al concepto de “Bloque” (Bloque de Plegado o Fold Block), dentro de los resaltadores de Lazarus, hay que notar que existen ciertas diferencias entre estos conceptos:

- Los bloques dentro del TSynFacilSyn, usan la maquinaria de los bloques de Lazarus para implementarse, pero complementan la funcionalidad para extender el concepto de bloque.
- Los bloques dentro del TSynFacilSyn se definen como regiones especiales de varios tokens, que pueden ocupar una o varias líneas. Los bloques de Lazarus, que ocupan una línea, no se consideran (no generan marcas de plegado).
- Los bloques de Lazarus, básicamente sirven para implementar la funcionalidad de plegado de código. Los bloques de TSynFacilSyn, permiten definir regiones de significado semántico.

Internamente los bloques y secciones de TSynFacilSyn, comparten la misma estructura. Se implementan con una clase:

```
TFaSynBlock = class //clase para manejar bloques de sintaxis
    name      : string; //nombre del bloque
    dStart, dEnd: string; //delimitadores
```

```
index      : integer;    //indica su posición dentro de TFaListBlocks
showFold   : boolean;    //indica si se mostrará la marca de plegado
parentBlk  : TFaSynBlock; //bloque padre (donde es válido el bloque)
BackCol    : TColor;     //color de fondo de un bloque
IsSection  : boolean;    //indica si es un bloque de tipo sección
UniqSec    : boolean;    //indica que es sección única
end;
```

Cada nuevo bloque creado (o sección), genera dinámicamente un nuevo objeto TFaSynBlock. La única diferencia interna, entre un bloque y una sección, es que esta tiene el campo “IsSection” en TRUE, y por lo tanto cobra sentido el campo “UniqSec”.

Para el manejo de bloques, se usan unos campos adicionales en los registros “TTokEspec”:

```
TTokEspec = record
...
    //propiedades para manejo de bloques y plegado de código
    bloIni : boolean;           //indica si el token es inicio de bloque de plegado
    bloIniL: array of TFaSynBlock; //lista de referencias a los bloques que abre
    bloFin : boolean;           //indica si el token es fin de bloque de plegado
    bloFinL: array of TFaSynBlock; //lista de referencias a los bloques que
cierra
    secIni : boolean;           //indica si el token es inicio de sección de bloque
    secIniL: array of TFaSynBlock; //lista de bloques de los que es inicio de
sección
end;
```

Cada, vez que se agrega una nueva sección o bloque, se agregan también los delimitadores, como tokens especiales (identificadores y símbolos), si es que no existían, y se configuran sus propiedades de bloque.

El manejo de bloques es independiente del manejo de tokens. El reconocimiento de tokens es una tarea independiente que se hace por rutinas especializadas. Los bloques trabajan sobre estas rutinas, sin interferir en su trabajo.

El procesamiento de bloques y secciones, introduce una carga adicional al resaltador, sin embargo en pruebas pesadas, se pudo estimar que el desempeño es tan bueno (y hasta mejor) como el de los resaltadores fijos que vienen con Lazarus.

Hay que notar que la rutina Rebuild(), puede no optimizar correctamente la sintaxis, cuando un símbolo común, es a la vez delimitador de bloque. Consideremos por ejemplo, que se definen un bloque cuyos delimitadores son los paréntesis. Entonces los símbolos “(” y “)”, se

crearán como símbolos especiales con bloques asociados, y tendrán que pasar por todas las verificaciones, cada vez que se detecte un paréntesis. En condiciones normales los paréntesis se tratarían como simples símbolos y se procesarían rápidamente. Si se definieran solo como símbolos especiales, Rebuild() podría optimizar el procesamiento usando la tabla de métodos y la rutina metSym1Car().

La idea para la implementación del plegado es simple, y está basado en el uso de estas tres funciones: StartCodeFoldBlock(), EndCodeFoldBlock() y TopCodeFoldBlockType(). De momento no se está usando las opciones de Configuraciones de Plegado<sup>9</sup>, mediante la clase "TSynCustomFoldConfig", aunque inicialmente se pensó derivar la clase "TFaSynBlock" a partir de "TSynCustomFoldConfig".

Los bloques se abren siempre, llamando a StartCodeFoldBlock(), y se cierran siempre, llamando a EndCodeFoldBlock(), independientemente si se mostrará o no la marca de plegado.

Cada vez que se encuentra un token especial, se verifica si este, es delimitador inicial de algún bloque o sección (con prioridad en la sección<sup>10</sup>), y de ser así se procede a abrir el bloque de plegado (o simplemente iniciar el bloque sin plegado, de acuerdo a la configuración). Para abrir el bloque de plegado, primero se verifica si el bloque o sección actual, es el que corresponde. Técnicamente, el bloque se abre después de reconocer el token, sin embargo, este debería abrirse antes, porque el token inicial es también parte del bloque, sin embargo, para fines de plegado (que trabaja con líneas) de código no hay perjuicio.

Para encontrar el cierre del bloque, se verifica, cada vez que se encuentra un token especial, si es que este es fin de algún bloque o sección. Usando la información del bloque o sección actual, se determina, si debe cerrarse o no el plegado. El plegado se cierra después de reconocer al token delimitador, y es así como se espera trabajar, así que por el lado de fin de bloque, no hay complicaciones.

Una tabla podría ayudar e explicar la situación:

---

<sup>9</sup> Las opciones de configuraciones de plegado permiten guardar información sobre los bloques de plegado, que sean accesibles exteriormente, desde propiedades públicas de la clase. Para más información se recomienda ver "La Biblia del SynEdit - Rev 5", en la sección sobre Plegado de Código.

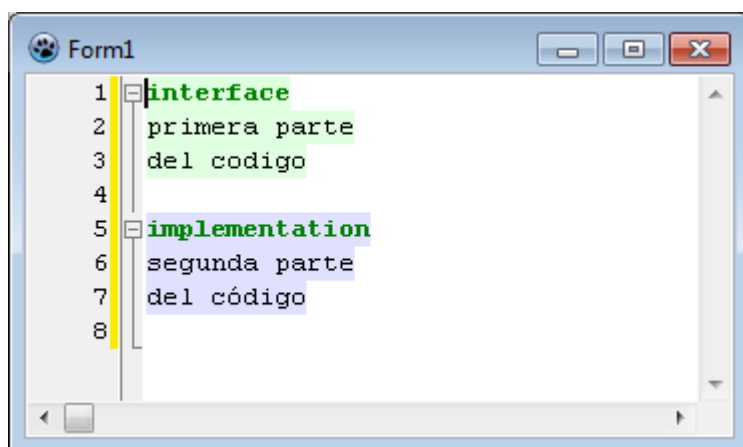
<sup>10</sup> El hecho de dar prioridad a la definición de sección, con respecto a la de los bloques, es una característica que se definió por observaciones de cómo se maneja el código en Pascal, pero debería ser consistente con otros lenguajes.

DELIMITADOR	TIPO	EL BLOQUE SE ABRE ..	EL BLOQUE DEBERÍA ABRIRSE ..
De inicio	Símbolo o Identificador	Después de identificar el token	Antes de identificar al token!!!!
Final	Símbolo o Identificador	Después de identificar el token	Después de identificar el token

Este método funciona bien, para bloques porque tiene tokens delimitadores definidos, y el desfase existente al momento de abrir el bloque, no es notorio para fines de plegado.

Para el manejo de secciones, sin embargo, la situación es un poco más problemática, porque las secciones no tienen un delimitador final definido. Así que en teoría deberían cerrarse antes de identificar al inicio de la siguiente sección o al fin del bloque que lo contiene. Sin embargo, por cuestiones técnicas el resaltador, recién detecta que debe cerrar la sección anterior, cuando ha identificado al token que inicia a la siguiente sección, y que por lo general está en la siguiente línea, del fin de la sección.

Para mejor entendimiento, veamos la siguiente figura:



Observando con detalle, se puede ver que el plegado de la sección “interface”, se extiende hasta la línea donde está “implementation”, porque recién allí, se ha identificado que la sección “interface” ha terminado. Así que tenemos un plegado que no es tan consistente con la definición. Afortunadamente, la maquinaria de plegado de SynEdit, sabrá que no debe ocultar esta línea cuando se cierre el pliegue de “interface”.

De momento, se ha mantenido este “defecto” así. Pero se sabe que es posible implementar mecanismos más elaborados, para cerrar correctamente el plegado en el caso de secciones (como se supone que se hace en el resaltador de Pascal de Lazarus).

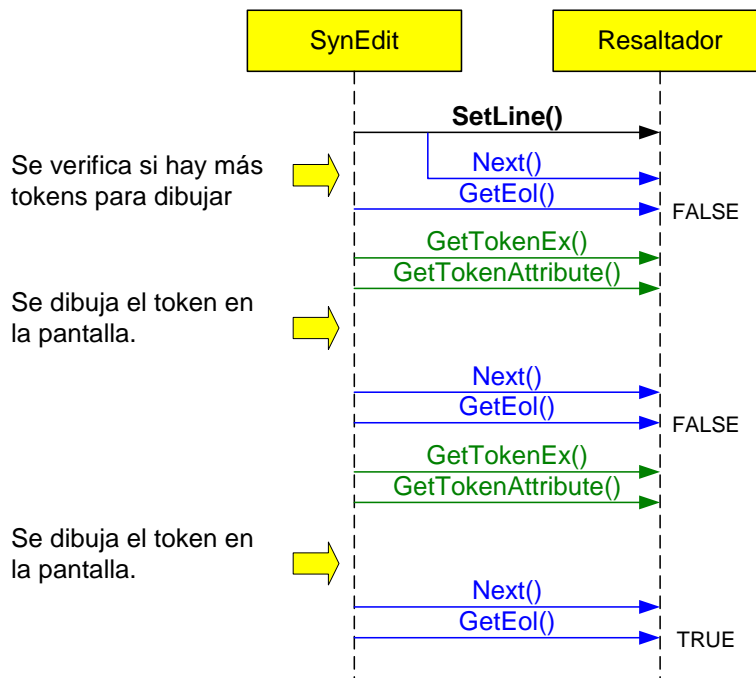
## 6.5 Coloreado de bloques

La funcionalidad de coloreado de bloques nos permite ver los límites físicos de los bloques en el editor, pero esto es básicamente una ilusión.

La posición inicial y/o final de los bloques no se almacena en ninguna parte. El resaltador no dispone de espacio de almacenamiento para definir bloques (como es la filosofía de trabajo). Toda la información referida a bloques de plegado (no bloques de TSynFacilSyn) la guarda el resaltador, junto con la información de rangos como un objeto asociado a cada línea, y básicamente lo que se guarda aquí, es el estado del resaltador al final de la exploración de la línea (Para más información, ver La Biblia del SynEdit en la sección sobre TSynCustomFoldHighlighter).

Los límites de los bloques de TSynFacilSyn, se definen en tiempo de ejecución, cuando se exploran los token, ya que los resaltadores tienen al token como la unidad mínima de exploración.

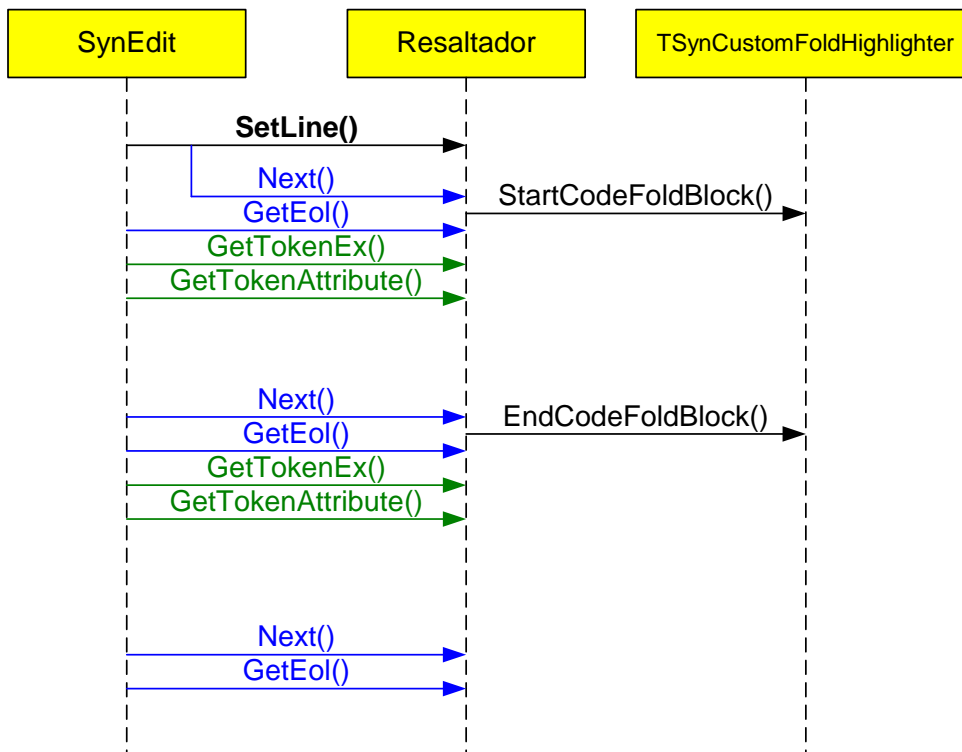
Para visualizar mejor el proceso de exploración, veamos el siguiente diagrama de secuencia:



El dibujo de un token completo, se realiza después de que SynEdit llama a `GetTokenEx()` y `GetTokenAttribute()`. Por lo tanto, si se desea cambiar el atributo del token debe hacerse antes de devolver el atributo con `GetTokenAttribute()`.

Esta es la forma en que se exploran los tokens. La pregunta lógica sería, ¿En qué momento se abren o cierran los bloques? Debido a como se ha implementado TSynFacilSyn, los bloques se abren o cierran (`StartCodeFoldBlock()` y `EndCodeFoldBlock()`), en el método `Next()`, por cuanto es aquí cuando se termina de “extraer “ los tokens del texto y se puede evaluar las condiciones necesarias.

El siguiente diagrama muestra la secuencia en la apertura y cierre de bloques:



Para lograr el coloreado de bloques, la técnica es cambiar el color de fondo en los atributos de los tokens contenidos en un bloque específico<sup>11</sup>.

El problema entonces, se reduce a poder determinar, antes de retornar el valor con `GetTokenAttribute()`, si el token pertenece o no a un bloque cualquiera. Para ello vamos a

<sup>11</sup> Desgraciadamente cambiar el atributo de fondo de los tokens, conlleva a una pérdida en la información del color de fondo de los tokens (queda como un pendiente, el solucionar esta falla).

interceptar el método TSynFacilSyn.GetTokenAttribute(), y cambiaremos el color de fondo del token, de acuerdo al color de fondo definido para el bloque actual:

```
function TSynFacilSyn.GetTokenAttribute: TSynHighlighterAttributes;  
{Debe devolver el atributo para el token actual. El token actual se actualiza  
con cada llamada a "Next", (o a "SetLine", para el primer token de la línea.)  
Esta función es la que usa SynEdit para definir el atributo del token actual}  
var topBlock: TFaSynBlock;  
begin  
    Result := fTokenID; //podría devolver "tkEol"  
    if Result<> nil then begin  
        //verifica coloreado de bloques  
        case ColBlock of  
            cbLevel: begin //pinta por nivel  
                Result.Background:=RGB(255-CodeFoldRange.CodeFoldStackSize*25,255-  
CodeFoldRange.CodeFoldStackSize*25,255);  
            end;  
            cbBlock: begin //pinta por tipo de bloque  
                topBlock := TFaSynBlock(TopCodeFoldBlockType);  
                if topBlock= nil then topBlock := MainBlk;  
                Result.Background:=topBlock.BackCol;  
            end;  
        end;  
    end;  
end;  
end;
```

La opción de coloreado "cbBlock", lee el bloque "actual" (último bloque abierto), usando TopCodeFoldBlockType(). Esto implica que el bloque actual, se mantenga abierto (no se llame a EndCodeFoldBlock) hasta que se haya llamado a GetTokenAttribute().

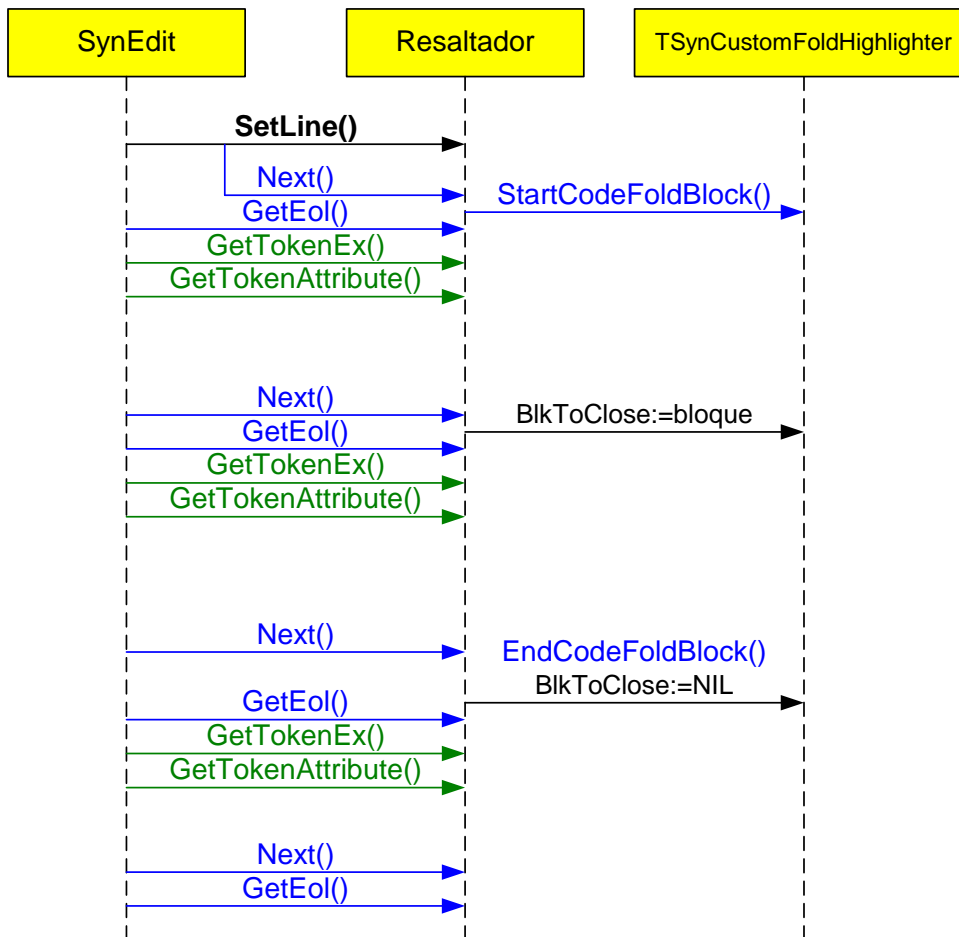
Al leer el token inicial de un bloque, el color del bloque se leerá correctamente, por cuanto el bloque se abre, antes de llamar a TSynFacilSyn.GetTokenAttribute() (se abre en Next()). Pero al momento del cierre del bloque, por el mismo efecto, el color leído, correspondería ya, al siguiente bloque (Ver diagrama de secuencia anterior).

Por eso se tendría que, para el coloreado de bloques, se produce el efecto contrario, con respecto a la detección de bloques (Ver sección anterior), en la determinación de los límites del bloque.

Para solucionar este desfase, es que el cierre de un bloque, se posterga hasta la lectura del siguiente token. Por ello es que se ha creado la bandera “BlkToClose”, y se termina de cerrar el bloque en “Next”<sup>12</sup>.

La variable “BlkToClose” es en realidad un puntero que apunta al bloque actual, que está en espera de cerrarse. Cuando no hay bloque en espera, tiene el valor NIL.

El diagrama siguiente muestra como es la secuencia actual:



Como “BlkToClose”, es una variable cuya vida no debe exceder al de una línea, se inicia siempre en SetLine().

<sup>12</sup> Inicialmente se trató de implementar el cierre del bloque en el mismo GetTokenAttribute(), después de fijar el color del bloque, pero se verificó que había errores en la coloración. No se indagó exactamente, el porqué.



Se puede ver entonces que toda la maquinaria de apertura/cierre de bloques, reside únicamente en el método `Next()`. Los límites de un bloque no se guardan en ningún lado de TSynFacilSyn. El bloque tiene un tiempo de vida en el proceso de exploración del texto.

La función de coloreado de bloques usa esta información para cambiar el color de fondo de los tokens, sobre la marcha, cuando se llama a `GetTokenAttribute()`.

## ***6.6 Bloques y Rangos***

Existe cierta interferencia entre el manejo de bloques, y los tokens multilínea (que también pueden tener información de plegado). Para mejorar esta relación, se tuvo que incluir el bloque “MulTokBlk”, para que los tokens multilínea pudieran usarlo cuando incluyeran plegado de código, y trabajar así de una forma más “estándar”.

Sin embargo, esto implica que todos los tipos de tokens multilínea comparten un solo objeto de bloque, lo que puede limitar las opciones de coloreado. Por ejemplo si se activa el coloreado del bloque MulTokBlk, este se hará efectivo para todos los tokens multilínea sin distinción, sin importar si son cadenas o comentarios.

## 6.7 Optimización de velocidad

Se sabe que la mayor dificultad en la implementación de un resaltador para SynEdit, es la velocidad de respuesta que debe tener, y eso depende de lo rápido que sea en identificar los tokens.

La optimización se ha hecho para el reconocimiento de tokens, no para la construcción de la sintaxis.

La implementación se ha hecho optimizando la velocidad, en la medida de lo posible. No se está usando “funciones hash”, como en la mayoría de resaltadores porque no son flexibles para configuración, y en mis pruebas, he verificado que hasta son más lentas en velocidad (Ver <http://forum.lazarus.freepascal.org/index.php?topic=22008.0> ).

El método usado, se asemeja a un árbol de prefijos, pero usando solo la primera letra para mejorar la identificación de los tokens. Este método de búsqueda rápida, se aplica tanto para identificadores como para delimitadores.

Considerar que los identificadores especiales, se guardan en alguna de las 57 tablas especiales que se tienen reservadas. Esta separación se hace para acelerar la identificación de identificadores específicos.

La idea es usar el carácter inicial (de la línea analizada), para determinar rápidamente a que rutina se deriva el procesamiento del token (usando la tabla de métodos). Como cada rutina, solo está especializada en identificar un grupo pequeño de identificadores, por eso se puede hacer rápidamente.

Par poder identificar rápidamente, los caracteres que pueden ser parte de un identificador, se usa una matriz de valores booleanos (CharsIdentif[]).

Los símbolos delimitadores se guardan únicamente en la matriz mSym[], y tienen un procesamiento similar al de los identificadores (para optimizar la velocidad). Para ello se configura la tabla de métodos, para que derive a la rutina de procesamiento de delimitadores (Procidnt), para cada carácter que pueda ser el inicio de un delimitador. También se hace uso de una tabla de caracteres válidos para un delimitador (CharsIdentif).

El resaltador trabaja de acuerdo a las reglas descritas en este documento, si se modifica alguna característica, este documento deberá ser actualizado.

Para ampliar la lista de caracteres disponibles para el carácter inicial de un identificador, se deben modificar “AgregaldenEspec” y crear el método apropiado (metXXX), además de crear las nuevas tablas.

## 6.8 Aligerando al resaltador

La unidad del resaltador, es un código bastante rápido y relativamente ligero en cuanto a espacio de memoria. Pero si se desea hacerlo más compacto, sea por temas de rendimiento o simplificación, se puede eliminar fácilmente algunas funcionalidades.

### 6.8.1 Funcionalidad “CaseSensitive”.

Esta opción requiere del uso de 23 tablas más, dentro de la unidad. Si no se desea esta opción y solo se quiere procesar los tokens ignorando la caja, se deben eliminar las tablas:

mA\_, mB\_, mC\_, mD\_, mE\_, mF\_, mG\_, mH\_,  
mI\_, mJ\_, mK\_, mL\_, mM\_, mN\_, mO\_, mP\_,  
mQ\_, mR\_, mS\_, mT\_, mU\_, mV\_, mW\_, mX\_, mY\_, mZ\_

Y también el campo CaseSensitive, y luego ir quitando selectivamente el código que hace uso de estos datos.

Los métodos metA\_, metB\_, ... También deben eliminarse.

La forma más sencilla, sería eliminar las variables indicadas y luego compilar para ver en qué parte del código se usan estas variables.

Para el caso de “CaseSensitive”, el trabajo sería simple por cuanto es una variable booleana, entonces procederíamos a la simplificación usando la consigna: “asumir que la variable tendrá siempre el valor FALSE”. Así por ejemplo, el siguiente código:

```
If CaseSensitive then  
  {parte TRUE}
```

```
else  
{parte FALSE}
```

Quedará como:

```
{parte FALSE}
```

Si se desea mayor simplificación se puede eliminar también la variable “tCasSen” de “FirstXMLExplor” y su código asociado.

Después de esta simplificación, el código se debe reducir en unas 100 líneas. Se estima además, que la velocidad del resaltador mejorará entre un 1 y 2% aproximadamente.

## 6.8.2 Funcionalidad “CharsEnd”.

Una característica poco usada, en el resaltador es la opción “CharsEnd”, al momento de definir los atributos por contenido.

Si se desea eliminar esta opción se deben eliminar las variables: carValFin1, carValFin2 y carValFin3, y todo el código que hace uso de estas variables (DefTokContent, metTokCont1, metTokCont2 y metTokCont3).

También se puede quitar la variable “tCharsEnd” (y su código asociado) de “LoadFromFile”.

Esta funcionalidad, sin embargo, es bastante ligera tanto en código como en procesamiento. Así que no hará mucha diferencia en quitarla, salvo que sea por fines de simplificación.

## 6.8.3 Manejo de secciones

Si solamente se desea usar el manejo de bloques, sin secciones, se puede eliminar por completo esta funcionalidad.

El código adicional para el manejo de las secciones, es considerable. Además existe una carga adicional de procesamiento.

Para eliminar el manejo de secciones, primero debemos eliminar las propiedades de “TTokEspec”, referidas a secciones: “secIni”, y “secIniL”. También las propiedades de “TFaSynBlock” referidas a secciones: “IsSection” y “UniqSec”. Todo el código asociado a estas propiedades debe ser eliminado. También se debe eliminar el método AddSection(), y el código que procesa secciones en LoadFromFile().

También se puede eliminar solamente, la característica de “Primera Sección”, quitando el campo “firstSec”, de “TTokEspec”, y todo el código asociado.

#### **6.8.4 Funcionalidad de plegado.**

Si no se requiere la funcionalidad de plegado de código de ninguna forma, se puede eliminar completamente esta funcionalidad. De ser así, se reduciría considerablemente el tamaño de la unidad así como la carga adicional de procesamiento.

Sin embargo, quitando a TSynFacilSyn, el plegado, equivale a tener un resaltador de tipo TSynFileSyn. Así que en lugar de eliminar manualmente todas las propiedades o métodos relativos al plegado, mejor convendría usar el resaltador TSynFileSyn, que es bastante más ligero.

#### **6.8.5 Lectura de archivo XML**

En el caso hipotético, de que no se vaya a configurar al resaltador mediante el archivo XML, y solo se desee hacerlo mediante instrucciones. Se puede eliminar toda la maquinaria pesada de lectura del archivo de sintaxis.

Para empezar, se deben eliminar FirstXMLExplor() y LoadFromFile().

También se debería eliminar el tipo TFaXMLatrib y las funciones LeeAtrib(), IsAttributeName() y ValidarAtribs() porque ya no serían necesarios.

El ahorro de código es considerable, de cerca de 300 líneas. La velocidad de procesamiento no se afecta, porque no se modifican las rutinas principales.

## 6.9 Limitaciones

El resaltador hace un compromiso entre velocidad y flexibilidad. Debido a esto, existen algunas limitaciones:

- a. Solo se permiten definir 4 tokens por contenido, adicionales al espacio reservado para los identificadores. Crear más, es bastante sencillo, y solo cargará ligeramente la memoria.
- b. No se pueden crear subconjuntos de tokens, para otros tokens por contenido que no sean identificadores o símbolos.
- c. El atributo `tFaTokContent.carValFin`, se han declarado como: `string[64]`, por temas de velocidad. Por lo tanto tienen un tamaño máximo de 64 caracteres.
- d. Los tokens delimitados, que tengan delimitador final de tipo identificador, siempre compararán este delimitador, considerando la caja. Esta limitación viene desde las primeras versiones.
- e. Los delimitadores de bloques no pueden ser de tipo “<identificador><espacio><identificador>”. Por ejemplo, no se puede definir “END IF”, como delimitador de bloque.
- f. Las secciones no pueden contener otras secciones. Actualmente las secciones solo pueden contener bloques, pero no es gran problema.

## ***6.10 Mejoras pendientes***

Existe siempre, posibilidades de mejora en el resaltador. Se listan solo algunas de las que estuvieron siempre en mente, cuando se desarrolló el resaltador.

- Corregir el límite de la marca de plegado para secciones (Ver 6.4 - Bloques y Secciones). Actualmente el plegado se extiende hasta una línea después del fin de la sección.
- Poder desactivar el coloreado de bloques, sin necesidad de crear de nuevo el resaltador. Actualmente, cuando se activa el coloreado de bloques, se pierde la información de color de fondo de todos los tokens.
- Creación de secciones dentro de secciones.
- Se debe dar la posibilidad de definir un bloque especial para los tokens multilínea, con plegado. Actualmente todos los tokens multilínea comparten el mismo bloque "MulTokBlk".

---

## Contenido

1	Introducción.....	4
1.1.1	Características.....	4
2	Resumen .....	6
2.1	Uso dentro de un programa. ....	6
2.2	Configuración usando archivo de sintaxis .....	6
2.3	Configuración usando Código .....	7
2.4	Cambiando los atributos.....	7
2.5	Plegado de código.....	9
3	Entendiendo una sintaxis.....	10
3.1	Elementos de la sintaxis.....	10
3.1.1	Identificadores .....	15
3.1.2	Números .....	15
3.1.3	Espacios.....	16
3.1.4	Cadenas.....	16
3.1.5	Comentarios.....	17
3.1.6	Símbolos.....	18
3.2	Identificadores especiales.....	20
3.3	Símbolos especiales .....	21
3.4	Definición de tokens .....	22
3.4.1	Tokens por contenido .....	22
3.4.2	Tokens delimitados .....	23
3.5	Delimitadores de Token .....	23
3.6	Bloques de código.....	26
3.7	Secciones .....	27
3.8	Niveles léxico, sintáctico y semántico.....	29
4	Configurando la Sintaxis con archivo externo. ....	31
4.1	El archivo de sintaxis.....	31
4.1.1	Caracteres especiales.....	32
4.1.2	Uso de intervalos y comodines .....	33
4.1.3	Uso de formas simplificadas .....	34
4.1.4	Uso de espacios, mayúsculas y minúsculas. ....	35
4.1.5	Comentarios en XML.....	36
4.2	Definición del Lenguaje.....	37
4.3	Configurando los atributos .....	38
4.4	Definición de Símbolos .....	41
4.4.1	Símbolos Especiales .....	41
4.5	Definición de Identificadores.....	44
4.5.1	Identificadores Especiales.....	45
4.6	Definición de Tokens por contenido.....	48
4.6.1	Tokens por contenido e identificadores .....	51
4.7	Definición de Tokens delimitados.....	53



---

4.8	Definición de cadenas.....	55
4.9	Definición de comentarios.....	55
4.10	Definición de Bloques .....	56
4.11	Bloques avanzados.....	59
4.12	Coloreado de bloques.....	61
4.13	Alcance de un bloque .....	63
4.14	Definición de Secciones .....	69
4.15	Más sobre secciones .....	74
5	Configurando la sintaxis por código.....	76
5.1	Tipos de tokens y atributos.....	78
5.2	Definición de Identificadores.....	80
5.3	Definición de Tokens por contenido.....	82
5.4	Identificadores Especiales.....	83
5.5	Símbolos Especiales .....	86
5.6	Definición de Tokens delimitados.....	89
5.7	Consideraciones sobre tokens delimitados .....	91
5.8	Ejemplo de configuración .....	95
5.9	Definición de bloques .....	97
5.10	Bloques como objetos .....	99
5.11	Bloques avanzados.....	101
5.12	Coloreado de bloques.....	104
5.13	Alcance de un bloque .....	107
5.14	Definición de secciones .....	110
5.15	Primera Sección .....	114
5.16	Secciones avanzadas.....	117
5.17	Obteniendo información sobre los bloques .....	121
5.17.1	SetHighlighterAtXY() .....	121
5.17.2	NestedBlocks() .....	122
5.17.3	NestedBlocksBegin() .....	122
5.17.4	TopCodeFoldBlock() .....	123
5.17.5	GetBlockInfoAtXY() .....	123
5.17.6	ExploreLine() .....	124
5.18	Usando al resaltador como analizador léxico - sintáctico .....	125
6	Para programadores .....	127
6.1	Categorización de tokens.....	127
6.2	Tratamiento de tokens .....	132
6.3	Manejo de Rangos .....	134
6.4	Bloques y Secciones.....	137
6.5	Coloreado de bloques.....	141
6.6	Bloques y Rangos .....	145
6.7	Optimización de velocidad.....	146
6.8	Aligerando al resaltador.....	147

---

6.8.1	Funcionalidad “CaseSensitive” .	147
6.8.2	Funcionalidad “CharsEnd” .	148
6.8.3	Manejo de secciones	148
6.8.4	Funcionalidad de plegado.	149
6.8.5	Lectura de archivo XML	149
6.9	Limitaciones	150
6.10	Mejoras pendientes	151
Contenido		152