Lazarus

Coloreado de sintaxis con SynEdit

Por Tito Hinostroza
Lima - 2013

Gran parte de este trabajo está basado en la experiencia de uso, la escasa documentación que existe en la web, Ingeniería inversa, y hasta a veces, analizando el código fuente de las unidades.

1.1 Coloreado de Sintaxis.

El componente SynEdit soporta el coloreado de sintaxis, esto significa que podemos resaltar con colores ciertos grupos de palabras (identificadores, variables, números, etc) o bloques de comentarios o cadenas.

Se pueden diferenciar 3 tipos de elementos en el coloreado

- Coloreado de palabras claves. Es el coloreado común en el que ciertas palabras o identificadores claves del texto se ponen de un color determinado. Normalmente se definen diversas categorías como palabras reservadas, palabras claves, identificadores, espacios, variables, macros, etc. Se puede definir diversos atributos de color de texto, color de fondo, marco, etc, para cada categoría.
- Coloreado de comentarios de una línea. Este coloreado es típico de los comentarios de una línea de la mayoría de lenguajes. Implica poner de un color específico, el texto de un comentario, desde el inicio hasta el final de la línea.
- Coloreado de rangos de texto o de contexto. Este coloreado se aplica también a los comentarios, o las cadenas. El coloreado del texto, afecta a un grupo de palabras, que pueden estar en una misma línea u ocupar varias líneas consecutivas.

Se puede elegir realizar el coloreado de sintaxis en un lenguaje predefinido como C++ o Java, (lo que significa que se resaltaran las palabras claves y elementos de ese lenguaje en particular) o se puede elegir un lenguaje totalmente nuevo.

1.2 Coloreado de Sintaxis Usando Componentes Predefinidos

Este tipo de coloreado de sintaxis implica el uso de componentes predefinidos de sintaxis. Estos componentes vienen preparados para trabajar en un lenguaje predefinido.

1.2.1 Usando un lenguaje predefinido

Estos componentes de sintaxis vienen ya instalados en el entorno Lazarus. Existen componentes para la mayoría de lenguajes comunes como: Pascal, C++, Java, HTML, Phyton, etc.

El método es simple: Arrastramos el control al formulario, donde está el control SynEdit que vamos a usar. Luego lo enlazamos, usando la propiedad "highlighter", del SynEdit.

Posteriormente podemos elegir los colores y atributos de las palabras claves, comentarios, números, y otras categorías, accediendo a las propiedades del objeto "TSynXXXSyn" (donde XXX representa el lenguaje elegido). Cada control "TSynXXXSyn", está representado por una unidad.

El objeto "TSynXXXSyn" viene optimizado para trabajar en su lenguaje predefinido y responde bien en velocidad. Sin embargo, las palabras claves y su detección están empotradas ("hardcoded") en el código de su unidad. Intentar agregar una palabra reservada más, no es tan sencillo, por la forma como se optimiza la búsqueda de palabras claves.

El uso de componentes predefinidos nos ahorra todo el trabajo de tener que procesar una sintaxis completa de un lenguaje conocido.

1.2.2 Usando un lenguaje personalizado

Adicionalmente a los componentes de los lenguajes predefinidos, existe un componente que se puede personalizar para un lenguaje diferente. Este es el componente "TSynAnySyn".

Para usar este componente, solo basta con incluirlo en el formulario, como cualquier otro componente de sintaxis. Pero antes de usarlo se le debe indicar cuales son las palabras claves que componen su sintaxis.

SI bien este componente soporta la definición de varios lenguajes, simples, no permite mucha flexibilidad a la hora de manejar los comentarios.

Además este componente no tiene un desempeño bueno en cuanto a velocidad, porque su diseño multi-lenguajes, no permite optimizarlo adecuadamente. No es recomendable el uso de este componente con un lenguaje de muchas palabras claves o en textos grandes.

Si se desea implementar un lenguaje nuevo con un buen desempeño y muy personalizado, es recomendable hacerlo por código.

1.3 Coloreado de Sintaxis Usando Código

Este tipo de coloreado permite personalizar de forma eficiente una sintaxis específica. Debe usarse cuando el lenguaje a usar no existe ya en uno de los componentes predefinidos, o no cumple con el comportamiento requerido.

Antes de diseñar nuestra clase para el manejo del coloreado, debemos entender un poco el funcionamiento del coloreado de sintaxis:

Para el coloreado de sintaxis, se debe crear una clase descendiente de “TSynCustomHighlighter”, a esta clase la llamaremos clase resaltador, u clase sintaxis, de la cual instanciaremos un resaltador. Esta resaltador debe asociarse luego al editor SynEdit que implementará el coloreado de sintaxis.

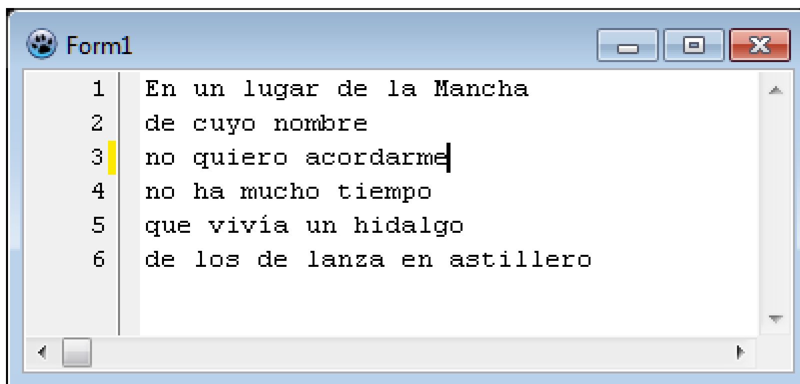
Cuando se asocia un resaltador, a un editor “TSynEdit”, este empezará a llamar a las rutinas del resaltador, cada vez que requiera información sobre el coloreado de la sintaxis.

TSynEdit no guarda información del coloreado del texto en alguna parte. Siempre que requiera información de coloreado, llamara al resaltador, para obtener, “sobre la marcha”, la información del coloreado.

Este resaltador, debe implementar el análisis del texto y la identificación de los elementos del texto para dar la información de los atributos del texto a SynEdit.

La exploración del texto a colorear lo hace SynEdit, procesando línea por línea. Cuando se modifica, una línea, se repinta, la línea modificada y las siguientes líneas, a partir de la línea modificada, que se muestren en la ventana visible. Este comportamiento en normal si se considera que puede incluirse en una línea un elemento de sintaxis que afecte a las demás líneas del texto (como el inicio de un comentario de varias líneas).

Consideremos el caso de un editor con el siguiente texto:



Al mostrar la ventana, después de haber etado oculta, se generarán los siguientes eventos:

```
1. SetLine: En un lugar de la Mancha
2. SetLine: de cuyo nombre
3. SetLine: no quiero acordarme
4. SetLine: no ha mucho tiempo
5. SetLine: que vivía un hidalgo
6. SetLine: de los de lanza en astillero
```

La etiqueta “SetLine”, indica que se extá explorando la línea indicada.

Al modificar la línea número 3 del mismo texto (un cualquier parte), la secuencia de exploracoión cambia un poco:

```
1. SetLine: no quiero acordarme,
2. SetLine: no ha mucho tiempo
3. SetLine: que vivía un hidalgo
```

```
4. SetLine: de cuyo nombre
5. SetLine: no quiero acordarme,
6. SetLine: no ha mucho tiempo
7. SetLine: que vivía un hidalgo
```

Podemos ver que el editor hace una exploración de las siguientes dos líneas, y luego hace una exploración nuevamente, pero empezando una línea anterior.

1.3.1 Exploración de líneas

Cada línea se asume que está dividida en elementos llamados "tokens". No hay parte de una línea que no sea un token. Un token puede ser un identificador, un símbolo, un carácter de control o un carácter en blanco.

Un token puede tener uno o más caracteres de longitud. Cada tokens o tipo de token, puede tener atributos particulares.

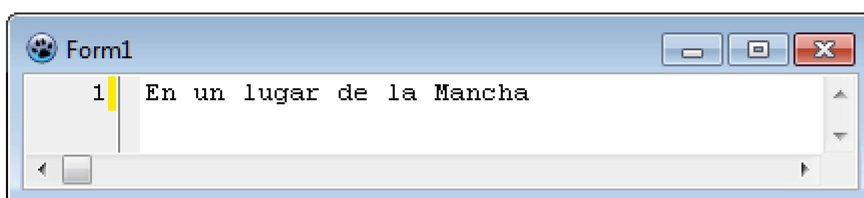
Cada vez que SynEdit necesita procesar una línea, se produce una secuencia de llamadas al resaltador (en realidad a "TSynCustomHighlighter", pero este los canaliza al resaltador que se esté usando), para obtener los atributos de cada elemento de la línea:

- Primeramente se llama el método "SetLine", para indicar que se está empezando la exploración de una nueva línea. Después de esta llamada espera que el método "GetTokenEx" o "GetTokenAttribute" devuelvan información sobre el token actual.
- Después de la llamada a "SetLine", se generarán múltiples llamadas al método "Next", para acceder a los siguientes tokens.
- SynEdit, espera que después de cada llamada a "Next", los métodos "GetTokenEx" y "GetTokenAttribute" devuelvan información sobre el token actualmente apuntado.
- Cuando SynEdit quiere verificar si se ha llegado al final de la línea de trabajo, hará una llamada a "GetEol". Esta debe estar funcional desde que se llama a "SetLine".

Las llamadas a estos método se producen repetidamente y en gran cantidad para cada línea. Por ello estos métodos, deben ser de respuesta rápida y de implementación eficiente. La demora en el procesamiento de alguno de estos métodos afectará al rendimiento del editor.

Para tener una idea del trabajo que hace SynEdit, en cuanto a coloreado sintaxis, presentamos a continuación, la secuencia de métodos llamados cuando se muestra una ventana de editor que estaba oculta.

La ventana ejemplo contiene este texto:



La secuencia de eventos llamados al mostrar esta ventana es:

1. SetLine: En un lugar de la Mancha
2. Next:0
3. GetEol
4. GetTokenEx
5. GetTokenAttribute
6. Next:2
7. GetEol
8. GetTokenEx
9. GetTokenAttribute
10. Next:3
11. GetEol
12. GetTokenEx
13. GetTokenAttribute
14. Next:5
15. GetEol
16. GetTokenEx
17. GetTokenAttribute
18. Next:6
19. GetEol
20. GetTokenEx
21. GetTokenAttribute
22. Next:11
23. GetEol
24. GetTokenEx
25. GetTokenAttribute
26. Next:12
27. GetEol
28. GetTokenEx
29. GetTokenAttribute
30. Next:14
31. GetEol
32. GetTokenEx
33. GetTokenAttribute
34. Next:15
35. GetEol
36. GetTokenEx
37. GetTokenAttribute
38. Next:17
39. GetEol
40. GetTokenEx
41. GetTokenAttribute
42. Next:18
43. GetEol
44. GetTokenEx
45. GetTokenAttribute
46. Next:24
47. GetEol

El valor indicado después del Evento Next, corresponde al carácter inicial que se explora. Se pude observar que el editor comprueba siempre si se ha llegado al final, después de cada llamada a "Next".

La llamada al método "GetTokenAttribute", es cuando el editor solicita el atributo a aplicar al texto.

Sin embargo, si se realiza una modificación, como insertar una coma al final de la línea, se genera la siguiente secuencia:

1. SetLine: En un lugar de la Mancha,
2. Next:0
3. GetEol
4. Next:2
5. GetEol
6. Next:3
7. GetEol
8. Next:5
9. GetEol
10. Next:6
11. GetEol
12. Next:11
13. GetEol
14. Next:12
15. GetEol
16. Next:14
17. GetEol
18. Next:15
19. GetEol
20. Next:17
21. GetEol
22. Next:18
23. GetEol
24. Next:24
25. GetEol
26. Next:25
27. GetEol
28. SetLine: En un lugar de la Mancha,
29. Next:0
30. GetEol
31. GetTokenEx
32. GetTokenAttribute
33. Next:2
34. GetEol
35. GetTokenEx
36. GetTokenAttribute
37. Next:3
38. GetEol
39. GetTokenEx
40. GetTokenAttribute

```
41.  Next:5                    60.  GetTokenAttribute
42.  GetEol                    61.  Next:15
43.  GetTokenEx                62.  GetEol
44.  GetTokenAttribute         63.  GetTokenEx
45.  Next:6                    64.  GetTokenAttribute
46.  GetEol                    65.  Next:17
47.  GetTokenEx                66.  GetEol
48.  GetTokenAttribute         67.  GetTokenEx
49.  Next:11                   68.  GetTokenAttribute
50.  GetEol                    69.  Next:18
51.  GetTokenEx                70.  GetEol
52.  GetTokenAttribute         71.  GetTokenEx
53.  Next:12                   72.  GetTokenAttribute
54.  GetEol                    73.  Next:24
55.  GetTokenEx                74.  GetEol
56.  GetTokenAttribute         75.  GetTokenEx
57.  Next:14                   76.  GetTokenAttribute
58.  GetEol                    77.  Next:25
59.  GetTokenEx                78.  GetEol
```

Se puede notar que el editor realiza primero una exploración previa, de toda la línea, antes de aplicar los atributos.

Adicionalmente al coloreado de sintaxis, SynEdit hace su propia exploración independiente para la detección y resaltado de “brackets”, (paréntesis, corchetes, llaves y comillas), cuando el cursor se encuentra apuntando a alguno de estos elementos:

```
texto |(texto entre paréntesis (otro texto)) más texto.
```

Sin embargo “SynEdit”, permite al resaltador, colaborar con la identificación de estos delimitadores. ¿Por qué?, porque el resaltador puede proporcionar información adicional para el resaltado de los “brackets”, ya que maneja los atributos de las diferentes partes del texto.

Para servir a la funcionalidad de “brackets” de SynEdit, el resaltador, debe implementar correctamente, los métodos: “GetToken”, y “GetTokenPos” y “GetTokenKind”.

¿Cómo funcionan? Para que un “bracket” de apertura, se asocie con su correspondiente “bracket” de cierre se verifica que “tokenKind” devuelve el mismo valor ambos. Si en la exploración se encontrara un “bracket” con un atributo diferente, este no se tomará en cuenta.

Si bien estos métodos no se usan para el coloreado de sintaxis, si determinan el comportamiento del resaltado de “brackets”.

Estos métodos son llamados de forma menos frecuente, que los métodos de coloreado de sintaxis. Solo se ejecutan cuando el cursor apunta a un “bracket” o cuando se agrega o quita alguno.

Si ha entendido el proceso de coloreado de sintaxis, ya estamos listos para dar los primeros pasos en la implementación de un coloreado por código.

1.3.2 Primeros pasos

Ante todo es recomendable crear una Unidad especial para almacenar el código de nuestra nueva sintaxis.

Para este ejemplo crearemos una unidad llamada “uSIntax“, e incluiremos las unidades necesarias para la creación de los objetos a usar.

En esta nueva unidad debemos crear necesariamente una clase derivada de “TSynCustomHighlighter” (definida en la unidad “SynEditHighlighter”), para la creación de nuestro resaltador:

```
{
 Unidad mínima que demuestra la estructura de una clase sencilla que será
 usada para el resaltado de sintaxis. No es funcional, es solo demostrativa.
                                       Creada por Tito Hinostroza: 04/08/2013
}
unit uSintax; {$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, Graphics, SynEditHighlighter;
type
  {Clase para la creación de un resaltador}
  TSynMiColor = class(TSynCustomHighlighter)
  protected
    posIni, posFin: Integer;
    linAct: String;
  public
    procedure SetLine(const NewValue: String; LineNumber: Integer); override;
    procedure Next; override;
    function  GetEol: Boolean; override;
    procedure GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
              override;
    function  GetTokenAttribute: TSynHighlighterAttributes; override;
  public
    function GetToken: String; override;
    function GetTokenPos: Integer; override;
    function GetTokenKind: integer; override;
    constructor Create(AOwner: TComponent); override;
  end;

implementation

constructor TSynMiColor.Create(AOwner: TComponent);
//Contsructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
```

```
{Es llamado por el editor, cada vez que necesita actualizar la información de
 coloreado sobre una línea. Despues de llamar a esta función, se espera que
 GetTokenEx, devuelva el token actual. Y también después de cada llamada a
 "Next".}
begin
  inherited;
  linAct := NewValue;   //copia la línea actual
  posFin := 1;          //apunta al primer caracter
  Next;
end;

procedure TSynMiColor.Next;
{Es llamado por SynEdit, para acceder al siguiente Token. Y es ejecutado por
 cada token de la línea en curso. En este ejemplo siempre se movera un
 caracter.}
begin
  posIni := posFin;    //apunta al siguiente token
  If posIni > length(linAct) then  //¿Fin de línea?
    exit         //salir
  else
    inc(posFin);  //mueve un caracter
end;

function TSynMiColor.GetEol: Boolean;
{Indica cuando se ha llegado al final de la línea}
begin
  Result := posIni > length(linAct);
end;

procedure  TSynMiColor.GetTokenEx(out  TokenStart:  PChar;  out  TokenLength:
integer);
{Devuelve información sobre el token actual}
begin
  TokenStart := @linAct[posIni];
  TokenLength := posFin - posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
{Devuelve información sobre el token actual}
begin
  Result := nil;
end;

{Las siguientes funciones, son usadas por SynEdit para el manejo de las
 llaves, corchetes, parentesis y comillas. No son cruciales para el coloreado
 de tokens, pero deben responder bien.}
function TSynMiColor.GetToken: String;
begin
  Result := '';
end;
```

```
function TSynMiColor.GetTokenPos: Integer;
begin
  Result := posIni - 1;
end;

function TSynMiColor.GetTokenKind: integer;
begin
  Result := 0;
end;

end.
```

Esta es, probablemente, la clase más simple que se puede implementar para un resaltado de sintaxis. Sin embargo, esta clase no resaltará ningún texto porque no contiene instrucciones para cambiar los atributos del texto. Se limita simplemente a devolver los valores por defecto a las solicitudes de “SynEdit”. No tiene utilidad, es simplemente un ejemplo minimalista de demostración.

Los métodos que aparecen como “override” son los que se requieren implementar para darle la funcionalidad de coloreado, a nuestro resaltador.

Con cada llamada a “SetLine”, se guarda una copia de la cadena en “linAct”, luego se utiliza esta cadena para ir extrayendo los tokens.

A cada petición de “Next”, esta unidad solo devuelve el siguiente carácter que se encuentra en la línea y el atributo devuelto por “GetTokenAttribute”, es siempre NIL, que significa que no hay atributos.

Los métodos “GetToken”, “GetTokenPos” y “GetTokenKind”, tampoco devuelven valores significativos, sino los valores nulos correspondientes.

La clase de resaltador que hemos creado, se llama “TSynMiColor”. No es posible usar la misma clase “TSynCustomHighlighter” como resaltador, porque esta clase es abstracta, y solo se usa para canalizar apropiadamente los requerimientos de TsynEdit, al realizar el coloreado de sintaxis.

Para usar la nueva sintaxis, debemos crear un objeto y asociarla al componente TSynEdit que vayamos a usar. Si tenemos nuestro formulario principal en Unit1 y nuestro objeto TsynEdit se llama “editor”, entonces el código para el uso de esta sintaxis podría ser:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses ... uSintax;
...
var
  Sintaxis  : TSynMiColor;
...
procedure TForm1.FormCreate(Sender: TObject);
...
  Sintaxis := TSynMiColor.Create(Self);  //crea resaltador
```

```pascal
  editor.Highlighter := Sintaxis;         //asigna la sintaxis al editor
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  editor.Highlighter := nil;  //quita resaltador
  Sintaxis.Destroy;      //libera objeto
end;
```

Entender el funcionamiento básico de este esquema de trabajo, es el primer paso para la creación de sintaxis funcionales. Si no ha entendido el funcionamiento de este ejemplo sencillo, le recomiendo que repase el código antes de pasar a las siguientes secciones.

1.3.3 Agregando funcionalidad a la sintaxis

El ejemplo anterior, se creo solo con fines didácticos. No se cumple con la funcionalidad deseada, pero se muestra la estructura que debe tener toda clase de coloreado de sintaxis.

Para comenzar debemos tener en mente que los métodos a implementar, deben ser de ejecución rápida. No deben estar cargados de mucho procesamiento, porque son llamados repetidamente para cada línea modificada del editor, así que no admiten retrasos, de otra forma el editor se volverá pesado y lento.

La primera modificación que debemos introducir, es el método de almacenamiento de la cadena. Cuando se llama a “SetLine”, se debe tener información, sobre la cadena. Pero la clase padre “TSynCustomHighlighter”, guarda ya una copia de la cadena, antes de llamar a “SetLine”.

Por ello no es eficiente crear una copia nueva para nosotros. Bastará con guardar una referencia, un puntero a esta cadena, almacenada en “TSynCustomHighlighter”.

Esto implica modificar la variable “linAct”, para que sea un “PChar”, en lugar de un string. Esto se hace en la definición de la clase. Los métodos “SetLine”, “Next”, “GetEol”, “GetTokenEx” y “GetTokenAttribute”, también deben cambiar:

Nuestro esqueleto de unidad quedaría:

```pascal
{
 Unidad mínima que demuestra la estructura de una clase sencilla que será
 usada para el resaltado de sintaxis. No es funcional, es solo demostrativa.
                                     Creada por Tito Hinostroza: 04/08/2013
}
unit uSintax; {$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, Graphics, SynEditHighlighter;
type
  {Clase para la creación de un resaltador}
  TSynMiColor = class(TSynCustomHighlighter)
```

```
  protected
    posIni, posFin: Integer;
    linAct: PChar;
  public
    procedure SetLine(const NewValue: String; LineNumber: Integer); override;
    procedure Next; override;
    function  GetEol: Boolean; override;
    procedure GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
              override;
    function  GetTokenAttribute: TSynHighlighterAttributes; override;
  public
    function GetToken: String; override;
    function GetTokenPos: Integer; override;
    function GetTokenKind: integer; override;
    constructor Create(AOwner: TComponent); override;
  end;

implementation

constructor TSynMiColor.Create(AOwner: TComponent);
//Constructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
{Es llamado por el editor, cada vez que necesita actualizar la información de
 coloreado sobre una línea. Despues de llamar a esta función, se espera que
 GetTokenEx, devuelva el token actual. Y también después de cada llamada a
 "Next".}
begin
  inherited;
  linAct := PChar(NewValue);  //copia la línea actual
  posFin := 0;                //apunta al primer caracter
  Next;
end;

procedure TSynMiColor.Next;
{Es llamado por SynEdit, para acceder al siguiente Token. Y es ejecutado por
 cada token de la línea en curso. En este ejemplo siempre se movera un
 caracter.}
begin
  posIni := posFin;    //apunta al siguiente token
  if linAct[posIni] = #0 then exit;   ///¿apunta al final?
  inc(posFin);  //mueve un caracter
end;

function TSynMiColor.GetEol: Boolean;
{Indica cuando se ha llegado al final de la línea}
begin
```

```
  Result := linAct[posIni] = #0;
end;

procedure  TSynMiColor.GetTokenEx(out  TokenStart:  PChar;  out  TokenLength:
integer);
{Devuelve información sobre el token actual}
begin
  TokenLength := posFin - posIni;
  TokenStart := linAct + posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
{Devuelve información sobre el token actual}
begin
  Result := nil;
end;

{Las siguientes funciones, son usadas por SynEdit para el manejo de las
 llaves, corchetes, parentesis y comillas. No son cruciales para el coloreado
 de tokens, pero deben responder bien.}
function TSynMiColor.GetToken: String;
begin
  Result := '';
end;

function TSynMiColor.GetTokenPos: Integer;
begin
  Result := posIni - 1;
end;

function TSynMiColor.GetTokenKind: integer;
begin
  Result := 0;
end;

end.
```

Ahora vemos que debemos iniciar “posFIn” a cero, que es donde empieza ahora la cadena, en “linAct”.

Pero aún esta clase está vacía de atributos. Lo primero que deberíamos hacer es crear nuestros atributos. Estos se deben declarar como propiedades del objeto “TSynMiColor”:

```
    fAtriComent  : TSynHighlighterAttributes;
    fAtriIdent   : TSynHighlighterAttributes;
    fAtriClave   : TSynHighlighterAttributes;
    fAtriNumero  : TSynHighlighterAttributes;
    fAtriEspac   : TSynHighlighterAttributes;
    fAtriCadena  : TSynHighlighterAttributes;
```

Todos los atributos son de tipo “TSynHighlighterAttributes”. Esta clase contiene los diversos atributos que se le pueden asociar a un token, como color de texto, color de fondo, color de borde, etc.

Luego en el constructor, debemos crear y definir las propiedades de estos atributos:

```
constructor TSynMiColor.Create(AOwner: TComponent);
//Constructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
  //atributo de comentarios
  fAtriComent   := TSynHighlighterAttributes.Create('Comment');
  fAtriComent.Style := [fsItalic];     //en cursiva
  fAtriComent.Foreground := clGray;    //color de letra gris
  AddAttribute(fAtriComent);
  //atribuuto de palabras claves
  fAtriClave    := TSynHighlighterAttributes.Create('Key');
  fAtriClave.Style := [fsBold];        //en negrita
  fAtriClave.Foreground:=clGreen;      //color de letra verde
  AddAttribute(fAtriClave);
  //atributo de números
  fAtriNumero   := TSynHighlighterAttributes.Create('Number');
  fAtriNumero.Foreground := clFuchsia;
  AddAttribute(fAtriNumero);
  //atributo de espacios. Sin atributos
  fAtriEspac    := TSynHighlighterAttributes.Create('space');
  AddAttribute(fAtriEspac);
  //atributo de cadenas
  fAtriCadena   := TSynHighlighterAttributes.Create('String');
  fAtriCadena.Foreground := clBlue;   //color de letra azul
  AddAttribute(fAtriCadena);
end;
```

Se han definido atributos de varias categorías de tokens. Aquí es donde se define la apariencia que tendrá el texto de los tokens.

Debemos recordar que todos los elementos de la línea a explorar, debe ser necesariamente un token, inclusive los espacios y símbolos.

El siguiente ejemplo, muestra como se puede dividir una cadena en tokens diversos:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
x	p		:	=		x	p		+		1	;				/	/	c	o	m	e	n	t	a	r	i	o

En este ejemplo el primer token, se define por los caracteres 1 y 2, y se muestra en amarillo. El segundo token es un espacio en blanco y se indica con el color verde. Los caracteres 4 y 5 pueden considerarse como un solo token o como dos tokens distintos. El carácter 12 es un token que seguramente estará en la categoría de números. Los caracteres 14, 15 y 16 se deben agrupar en un solo token espacio de 3 caracteres de ancho (sería ineficiente tratarlo como 3 tokens). A partir del carácter 17, se encuentra un token que abarca hasta el fin de la línea.

Los límites del token, lo define el resaltador. El editor hará caso, sumisamente, a lo indicado por este objeto, coloreándolo de acuerdo a los atributos entregados.

Como se ve en el ejemplo, todos los caracteres de la línea deben pertenecer a un token. El tamaño de un token va desde un carácter hasta el total de caracteres de la línea. El editor procesará más rápidamente la línea, mientras menos tokens hayan en ella.

Para identificar fácilmente a los atributos, es conveniente crear una enumeración para los atributos de tokens:

```
  //ID para categorizar a los tokens
  TtkTokenKind = (tkComment, tkKey, tkNull, tkNumber, tkSpace, tkString,
tkUnknown);
```

El identificador "tkUnknown", indicará que el token actual no se ha identificado. En este caso, se asumirá que no tiene atributos.

Y necesitamos, además, un campo para identificar al token actual:

```
TSynMiColor = class(TSynCustomHighlighter)
   ...
   fTokenID: TtkTokenKind;  //Id del token actual
   ...
end;
```

Ahora cuando queramos asignar un atributo, al token actual, debemos poner en "fTokenID", el identificador del token.

Una vez creados los atributos, debemos agregar funcionalidad al método "Next", para que pueda extraer los tokens adecuadamente, de la línea de trabajo. La implementación debe ser lo más eficiente posible, por ello usaremos el método de tabla de funciones o de métodos.

La idea es leer el carácter de un token, y de acuerdo a su valor ASCCI, llamamos a una función apropiada, para tratar ese carácter. Para que la llamada sea eficiente, creamos una tabla y la llenamos con punteros a las funciones adecuadas.

```
Type
  TProcTableProc = procedure of object; //Tipo procedimiento para procesar el
                                        //token por el carácter inicial.
...

TSynMiColor = class(TSynCustomHighlighter)
  protected
    ...
    fProcTable: array[#0..#255] of TProcTableProc; //tabla de funciones
    ...
end;
```

El tipo “TProcTableProc” es un método simple que define procedimientos sin parámetros (así la llamada se hace más rápida). Este tipo de procedimiento es el que se llamará cuando se identifique el carácter inicial de algún token.

Ahora que se tiene definido el tipo de procedimiento a usar, se debe crear estos procedimientos de tratamientos de tokens y llenar la tabla de métodos con sus direcciones. El siguiente código es un ejemplo sencillo de llenado de la tabla de métodos:

```
...

procedure TSynMiColor.CreaTablaDeMetodos;
{Construye la tabla de las funciones usadas para cada caracter inicial del
 tóken a procesar. Proporciona una forma rápida de procesar un token por el
 caracter inicial}
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
      '_','A'..'Z','a'..'z': fProcTable[I] := @ProcIdent;
      #0  : fProcTable[I] := @ProcNull; //Caracter de marca de fin de cadena
      #1..#9, #11, #12, #14..#32: fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
end;
```

Este método, hace corresponder la dirección de una función a cada una de las 256 de las posiciones de la tabla “fProcTable[]”.

El procedimiento “ProcIdent”, es la que se llama cuando se detecta un carácter alfabético (o guion), porque corresponde al inicio de un identificador. Su implementación es sencilla:

```
procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
begin
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
    Inc(posFin);
  fTokenID := tkKey;
end;
```

La cadena “linAct”, se va explorando hasta encontrar un carácter que no sea un carácter válido para un identificador. Observar que no se considera los caracteres del código ASCII extendido (á,é,í, etc). En este ejemplo sencillo, no se distingue el tipo de identificador, sino que se le asigna a todos el atributo “tkKey”. Si se quisiera elegir solo a algunas palabras para marcarlas como “tkKey”, se debe hacer aquí.

El procedimiento “ProcNull”, se llama al detectar el carácter NUL, es decir el fin de la cadena. Así que su procesamiento solo reduce a marcar “fTokenID” como “tkNull”.

```
procedure TSynMiColor.ProcNull;
```

```
//Procesa la ocurrencia del caracter #0
begin
  fTokenID := tkNull;   //Solo necesita esto para indicar que se llegó al
final de la línea
end;
```

Observar que no se avanza más en la exploración de la cadena. Este procedimiento es importante para detectar el final de la cadena y permite implementar “GetEol”, de forma sencilla:

```
function TSynMiColor.GetEol: Boolean;
//Indica cuando se ha llegado al final de la línea
begin
  Result := fTokenId = tkNull;
end;
```

EL procedimiento “ProcSpace”, permite procesar los bloques de espacios en blanco. Para los fines de sintaxis, se considerará espacios en blanco, los primeros 32 caracteres del código ASCII, exceptuando los caracteres #10 y #13 que corresponden a saltos de línea:

```
procedure TSynMiColor.ProcSpace;
//Procesa caracter que es inicio de espacio
begin
  fTokenID := tkSpace;
  repeat
    Inc(posFin);
  until (linAct[posFin] > #32) or (linAct[posFin] in [#0, #10, #13]);
end;
```

El carácter de tabulación #9 y el espacio #32, están considerados como espacios en blanco. A estos caracteres en blanco, se le asigna el atributo “tkSpace”, que normalmente debe estar sin resaltado.

El otro procedimiento importante, es “ProcUnknown”, que está destinado a procesar a todos aquellos tokens que no están considerados dentro de una categoría especial. En nuestro caso será todos los símbolos y números:

```
procedure TSynMiColor.ProcUnknown;
begin
  inc(posFin);
  while (linAct[posFin] in [#128..#191]) OR // continued utf8 subcode
   ((linAct[posFin]<>#0) and (fProcTable[linAct[posFin]] = @ProcUnknown)) do
inc(posFin);
  fTokenID := tkUnknown;
end;
```

Es importante tener siempre un procedimiento de este tipo para considerar todos aquellos tokens que no son categorizados en grupos predefinidos. Observar que también se consideran los

caracteres UTF-8 del código ASCII extendido. Esto es normal ya que SynEdIt trabaja solamente con UTF-8.

Una vez definidos estos procedimientos básicos, se debe implementar la llamada en el método "Next". El código tendría la siguiente forma:

```
procedure TSynMiColor.Next;
//Es llamado por SynEdit, para acceder al siguiente Token.
begin
  posIni := posFin;            //apunta al siguiente token
  fProcTable[linAct[posFin]]; //Se ejecuta la función que corresponda.
end;
```

Aunque no resulta obvio, se puede apreciar la llamada a la función de procesamiento apropiada para cada carácter. Obviamente se debe haber llenado primeramente "fProcTable".

Este modo de procesamiento resulta bastante rápido si se compara con un conjunto de condicionales o hasta con una sentencia "case .. of".

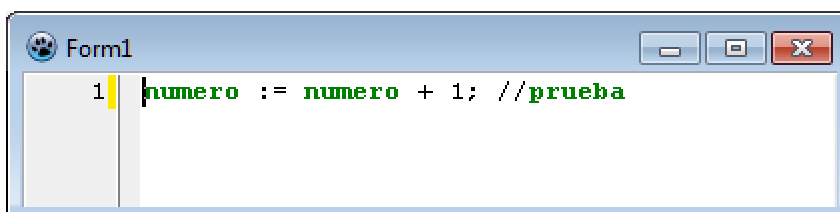
La función de procesamiento asignada, se encargará de actualizar el índice "posFin", que debe quedar siempre apuntando al inicio del siguiente token o al fin de la cadena.

Para que la sintaxis sea reconocida, solo falta modificar "GetTokenAttribute", para indicarle al editor, que atributo debe usar para cada token:

```
function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
//Devuelve información sobre el token actual
begin
  case fTokenID of
    tkComment: Result := fAtriComent;
    tkKey     : Result := fAtriClave;
    tkNumber : Result := fAtriNumero;
    tkSpace   : Result := fAtriEspac;
    tkString : Result := fAtriCadena;
    else Result := nil;  //tkUnknown, tkNull
  end;
end;
```

Tal como hemos definido nuestro resaltador, se reconocerán todas las palabras como palabras claves, y se mostrarán en color verde. Los símbolos y demás caracteres imprimibles, se mostrarán sin atributos, es decir que tomarán el color por defecto del texto.

La siguiente pantalla muestra como quedaría un texto simple, usando este resaltador:



El código completo de la unidad quedaría así:

```
{
 Unidad mínima que demuestra la estructura de una clase sencilla que será usada
 para el resaltado de sintaxis. No es funcional, es solo demostrativa.
                                                Creada por Tito Hinostroza: 04/08/2013
}
unit uSintax; {$mode objfpc}{$H+}
interface
uses
  Classes, SysUtils, Graphics, SynEditHighlighter;
type
  {Clase para la creación de un resaltador}

  //ID para categorizar a los tokens
  TtkTokenKind = (tkComment, tkKey, tkNull, tkNumber, tkSpace, tkString, tkUnknown);

  TProcTableProc = procedure of object; //Tipo procedimiento para procesar el
                                        //token por el carácter inicial.
  { TSynMiColor }
  TSynMiColor = class(TSynCustomHighlighter)
  protected
    posIni, posFin: Integer;
    linAct   : PChar;
    fProcTable: array[#0..#255] of TProcTableProc;   //tabla de procedimientos
    fTokenID : TtkTokenKind;  //Id del token actual
    //define las categorías de los "tokens"
    fAtriComent  : TSynHighlighterAttributes;
    fAtriClave   : TSynHighlighterAttributes;
    fAtriNumero  : TSynHighlighterAttributes;
    fAtriEspac   : TSynHighlighterAttributes;
    fAtriCadena  : TSynHighlighterAttributes;
  public
    procedure SetLine(const NewValue: String; LineNumber: Integer); override;
    procedure Next; override;
    function  GetEol: Boolean; override;
    procedure GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
              override;
    function  GetTokenAttribute: TSynHighlighterAttributes; override;
  public
    function GetToken: String; override;
    function GetTokenPos: Integer; override;
    function GetTokenKind: integer; override;
    constructor Create(AOwner: TComponent); override;
  private
    procedure CreaTablaDeMetodos;
    procedure ProcIdent;
    procedure ProcNull;
    procedure ProcSpace;
    procedure ProcUnknown;
  end;

implementation

constructor TSynMiColor.Create(AOwner: TComponent);
```

```
//Constructor de la clase. Aquí se deben crear los atributos a usar.
begin
  inherited Create(AOwner);
  //atributo de comentarios
  fAtriComent  := TSynHighlighterAttributes.Create('Comment');
  fAtriComent.Style := [fsItalic];     //en cursiva
  fAtriComent.Foreground := clGray;    //color de letra gris
  AddAttribute(fAtriComent);
  //atribuuto de palabras claves
  fAtriClave   := TSynHighlighterAttributes.Create('Key');
  fAtriClave.Style := [fsBold];       //en negrita
  fAtriClave.Foreground:=clGreen;     //color de letra verde
  AddAttribute(fAtriClave);
  //atributo de números
  fAtriNumero  := TSynHighlighterAttributes.Create('Number');
  fAtriNumero.Foreground := clFuchsia;
  AddAttribute(fAtriNumero);
  //atributo de espacios. Sin atributos
  fAtriEspac   := TSynHighlighterAttributes.Create('space');
  AddAttribute(fAtriEspac);
  //atributo de cadenas
  fAtriCadena  := TSynHighlighterAttributes.Create('String');
  fAtriCadena.Foreground := clBlue;   //color de letra azul
  AddAttribute(fAtriCadena);

  CreaTablaDeMetodos;  //Construte tabla de métodos
end;

procedure TSynMiColor.CreaTablaDeMetodos;
{Construye la tabla de las funciones usadas para cada caracter inicial del tóken a
procesar.
 Proporciona una forma rápida de procesar un token por el caracter inicial}
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
      '_','A'..'Z','a'..'z': fProcTable[I] := @ProcIdent;
      #0      : fProcTable[I] := @ProcNull;   //Se lee el caracter de marca de fin de
cadena
      #1..#9, #11, #12, #14..#32: fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
end;

procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
begin
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
    Inc(posFin);
  fTokenID := tkKey;
end;

procedure TSynMiColor.ProcNull;
//Procesa la ocurrencia del caracter #0
```

```
begin
  fTokenID := tkNull;   //Solo necesita esto para indicar que se llegó al final de la
línae
end;
procedure TSynMiColor.ProcSpace;
//Procesa caracter que es inicio de espacio
begin
  fTokenID := tkSpace;
  repeat
    Inc(posFin);
  until (linAct[posFin] > #32) or (linAct[posFin] in [#0, #10, #13]);
end;
procedure TSynMiColor.ProcUnknown;
begin
  inc(posFin);
  while (linAct[posFin] in [#128..#191]) OR // continued utf8 subcode
   ((linAct[posFin]<>#0)   and   (fProcTable[linAct[posFin]]   =   @ProcUnknown))   do
inc(posFin);
  fTokenID := tkUnknown;
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
{Es llamado por el editor, cada vez que necesita actualizar la información de
 coloreado sobre una línea. Despues de llamar a esta función, se espera que
 GetTokenEx, devuelva el token actual. Y también después de cada llamada a
 "Next".}
begin
  inherited;
  linAct := PChar(NewValue);  //copia la línea actual
  posFin := 0;                //apunta al primer caracter
  Next;
end;

procedure TSynMiColor.Next;
//Es llamado por SynEdit, para acceder al siguiente Token.
begin
  posIni := posFin;           //apunta al siguiente token
  fProcTable[linAct[posFin]]; //Se ejecuta la función que corresponda.
end;

function TSynMiColor.GetEol: Boolean;
{Indica cuando se ha llegado al final de la línea}
begin
  Result := fTokenId = tkNull;
end;

procedure TSynMiColor.GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
{Devuelve información sobre el token actual}
begin
  TokenLength := posFin - posIni;
  TokenStart := linAct + posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
//Devuelve información sobre el token actual
```

```
begin
  case fTokenID of
    tkComment: Result := fAtriComent;
    tkKey     : Result := fAtriClave;
    tkNumber : Result := fAtriNumero;
    tkSpace   : Result := fAtriEspac;
    tkString : Result := fAtriCadena;
    else Result := nil;  //tkUnknown, tkNull
  end;
end;

{Las siguientes funciones, son usadas por SynEdit para el manejo de las
 llaves, corchetes, parentesis y comillas. No son cruciales para el coloreado
 de tokens, pero deben responder bien.}
function TSynMiColor.GetToken: String;
begin
  Result := '';
end;

function TSynMiColor.GetTokenPos: Integer;
begin
  Result := posIni - 1;
end;

function TSynMiColor.GetTokenKind: integer;
begin
  Result := 0;
end;

end.
```

1.3.4 Propiedad GetDefaultAttribute

Puede que alguien se haya preguntado ¿Cómo acceder, desde fuera de la clase, a los atributos de, por ejemplo, las palabras claves?. Recordemos que los atributos de los tokens se deben declarar en el resaltador y no en la clase padre “TSynCustomHighlighter”.

Una respuesta sencilla, sería “ponemos las propiedades de atributo como públicos, y luego podremos referenciarlo como cualquier, propiedad de nuestro resaltador.

Y es cierto, eso funcionaría, pero si la pregunta fuera: ¿Cómo acceder desde el editor a los atributos de los tokens? Entonces ahí si se complica un poco la situación, porque, a pesar de que el editor (de la clase TSyenEdit) tiene la propiedad “HighLighter”, esta solo hace referencia a la clase “TSynCustomHighlighter” y no a la clase derivada (resaltador) que siempre usamos para implementar el coloreado.

Para solventar, en parte, esta dificultad, existe un método adicional que es recomendable implementar. Este método es “GetDefaultAttribute” y permitirá a nuestro resaltador, responder a las peticiones de acceso a los atributos que genere “TSynCustomHighlighter”.

A pesar de que la clase “TSynCustomHighlighter” no incluye propiedades de tipo atributo (se deja al programador la libertad de crear los que desee), si incluye una forma de acceder a los atributos principales de los tokens. En la clase se han definido las propiedades fijas:

```
property CommentAttribute: TSynHighlighterAttributes;
property IdentifierAttribute: TSynHighlighterAttributes;
property KeywordAttribute: TSynHighlighterAttributes;
property StringAttribute: TSynHighlighterAttributes;
property SymbolAttribute: TSynHighlighterAttributes;
property WhitespaceAttribute: TSynHighlighterAttributes;
```

Que permiten leer o modificar los atributos indicados. Sin embargo, para que estas propiedades funcionen, nosotros debemos sobre-escribir (override) en nuestro resaltador, el siguiente método:

```
function TSynMiColor.GetDefaultAttribute(Index: integer): TSynHighlighterAttributes;
{Este método es llamado por la clase "TSynCustomHighlighter", cuando se accede a alguna de
 sus propiedades:  CommentAttribute, IdentifierAttribute, KeywordAttribute, StringAttribute,
 SymbolAttribute o WhitespaceAttribute.
}
begin
  case Index of
    SYN_ATTR_COMMENT   : Result := fCommentAttri;
    SYN_ATTR_IDENTIFIER: Result := fIdentifierAttri;
    SYN_ATTR_KEYWORD   : Result := fKeyAttri;
    SYN_ATTR_WHITESPACE: Result := fSpaceAttri;
    SYN_ATTR_STRING    : Result := fStringAttri;
    else Result := nil;
  end;
end;
```

Como se ve, la idea es darle acceso a nuestros atributos, de acuerdo al tipo de atributo, solicitado. Desde luego, si no hemos definido un atributo específico, podríamos devolver NIL. De la misma forma, es posible que hayamos definido atributos adicionales que podrían no ser accesibles desde fuera de la clase, porque no se encuentran en la categoría solicitada.

Cuando alguien accede a la propiedad “CommentAttribute”, de “TSynCustomHighlighter”, esta llama a “GetDefaultAttribute”, pasando el parámetro “SYN_ATTR_COMMENT”. Es decisión del programador, devolver el atributo que considere necesario. Lo común sería devolver el atributo que represente a los comentarios, pero la clase no hará ninguna validación posterior. En teoría podríamos devolver el atributo que deseemos.

Si no va a haber accesos a las propiedades mencionadas de “TSynCustomHighlighter”, no es necesario implementar “GetDefaultAttribute”, sin embargo, es recomendable implementarlo siempre.

1.3.5 Reconociendo palabras claves.

En el ejemplo anterior marcamos a todos los identificadores como palabras claves asignándole el atributo “tkKey”. Esto lo hacíamos en el método “ProcIdent”:

```
procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
begin
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
    Inc(posFin);
  fTokenID := tkKey;
end;
```

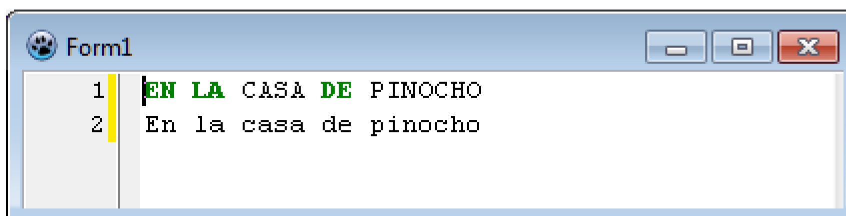
Pero en un caso norma, solo se marcarán algunos identificadores como palabras claves. Para ello, el camino más sencillo podría ser, comparar el token actual, con un grupo de palabras claves, y solo en caso de que coincidan, marcarlas como palabras claves.

El código a usar, podría ser como este:

```
procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
var tam: integer;
begin
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
    Inc(posFin);
  tam := posFin - posIni;
  if strlcomp(linAct + posIni,'EN',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'DE',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LA',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LOS',tam) = 0 then fTokenID := tkKey else
    fTokenID := tkUnknown;  //identificador común
end;
```

La comparación de cadenas se hace usando la función “strlcomp”, porque estamos manejando una variable “PChar”.

En este código se reconocen solo las palabras “EN”, “LA” Y “DE” como palabras reservadas. Al aplicar esta modificación podríamos tener una pantalla como esta.



Observar que las rutinas solo reconocen las mayúsculas, porque la comparación de cadenas se hace de esta forma.

Para ir agregando más palabras, se puede ir aumentando la lista y elegir los atributos dadas a cada categoría de palabras. Sin embargo, este método se vuelve pesado, conforme crece la cantidad de palabras y condicionales a agregar y por lo tanto no es el camino ideal a seguir en la implementación de una sintaxis adecuada.

Más adelante veremos como podemos optimizar la detección de identificadores.

1.3.6 Optimizando la sintaxis.

Como hemos estado insistiendo a lo largo de esta descripción, es importante mantener un código rápido en aras del buen desempeño del editor. Para ello debemos identificar los puntos donde podamos reducir ciclos de ejecución.

Analizando el código anterior se puede ver que el procedimiento "ProcIdent", es el más pesado en cuanto a procesamiento. Por su implementación requiere hacer múltiples comparaciones y verificaciones para detectar los identificadores a colorear.

La primera optimización que haremos tiene que ver con la condición:

```
  while linAct[posFin] in ['_','A'..'Z','a'..'z'] do
```

A pesar, de que el uso de conjuntos resulta eficiente, este código puede optimizarse considerablemente si se usa una tabla de selección.

Poder identificar rápidamente si un carácter se encuentra en una lista, es fácil si enfocamos el problema desde otra perspectiva. Imaginemos que cada carácter está asociado a una tabla que contiene como valor una bandera simple que indica si la variable es o no es parte de la lista. Una estructura así sería del tipo:

```
  Identifiers: array[#0..#255] of ByteBool;
```

Ahora creamos un procedimiento de llenado que marque solo las casillas de caracteres válidos, como "true":

```
procedure CreaTablaIdentif;
var
  i: Char;
begin
  for I := #0 to #255 do
  begin
    Case i of
      '_', '0'..'9', 'a'..'z', 'A'..'Z': Identifiers[i] := True;
    else
      Identifiers[i] := False;
    end;
  end;
end;
```

Una vez llenada esta tabla, y apodemos usarla para detectar rápidamente, que caracteres se consideran como parte de un identificador:

```
procedure TSynMiColor.ProcIdent;
//Procesa un identificador o palabra clave
var tam: integer;
begin
  while Identifiers[linAct[posFin]] do Inc(posFin);
  tam := posFin - posIni;
  if strlcomp(linAct + posIni,'EN',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'DE',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LA',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LOS',tam) = 0 then fTokenID := tkKey else
    fTokenID := tkUnknown;  //identificador común
end;
```

El siguiente punto a optimizar Está en las comparaciones múltiples. Lógicamente en este ejemplo, solo hay 4 comparaciones, pero normalmente podemos estar trabajando con más de 100. En estas condiciones, aunque no lo parezca, se puede estar perdiendo un tiempo valioso en detección de cadenas, haciendo muchas veces, verificaciones redundantes.

El problema se reduce en optimizar la comparación de una cadena en una lista de varias. Existen diversos métodos para llevar a cabo la optimización de esta tarea.

La mayoría de los componentes de sintaxis de Lazarus, usan el método de las funciones Hash (Hash-functions) en el cual no entraré en detalle, pero que básicamente se trata en asignarle a cada palabra clave, a detectar, un valor numérico, más o menos único[1], que permita categorizarlo en un número pequeño de grupos. El procesamiento de cada grupo está implementado en un procedimiento llamado: "FuncXX", donde XX es el valor suma de sus caracteres.

A cada nuevo identificador que se desea comparar, se le aplica la misma función par obtener su valor numérico y de acuerdo al valor obtenido, se le direcciona a la función adecuada (usando una tabla de funciones). Esta función hace una comparación rápida para validar si el identificador corresponde a la palabra(s) clave(s) buscadas.

Aunque este método es rápido, no es legible y confunde fácilmente. Además las modificaciones sencillas, como agregar una nueva palabra clave, requiere de un cálculo cuidadoso antes de modificar el código. Para complicar las cosas, las palabras claves están empotradas (hardcoded) en el código.

Aquí propondré un método que, comparable en eficiencia, resulta ser mucho más legible y fácil de modificar.

---

[1] Este valor único se obtiene creando una tabla que asigna un valor a cada letra del alfabeto ingles (usualmente en la tabla mHashTable[], y aprovechando para llenarla en la función "MakeIdentTable"). Con esta tabla se calcula el valor que le corresponde a cada palabra clave, sumando el valor de cada letra en el identificador. El valor obtenido suele depender de las letras del identificador y de su tamaño, pero usualmente no supera a 200 en una sintaxis normal. Este valor no es único para cada palabra, ya que varias palabras distintas pueden compartir el mismo valor, pero permite categorizar de manera efectiva a los identificadores para restringir la búsqueda a un grupo mucho menor.

El método consiste en crear una primera categorización de las palabras usando la misma tabla de métodos creada en “CreaTablaDeMetodos”, creando una función para cada letra inicial del identificador. Así el código de “CreaTablaDeMetodos”, tendría la siguiente forma:

```
procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of

      ...

      'A','a': fProcTable[I] := @ProcA;
      'B','b': fProcTable[I] := @ProcB;
      'C','c': fProcTable[I] := @ProcC;
      'D','d': fProcTable[I] := @ProcD;
      'E','e': fProcTable[I] := @ProcE;
      'F','f': fProcTable[I] := @ProcF;
      'G','g': fProcTable[I] := @ProcG;
      'H','h': fProcTable[I] := @ProcH;

      ...

end;
```

Luego en los procedimientos ProcA, ProcB, ... etc, se realiza el procesamiento de un grupo reducido de identificadores, reduciendo sensiblemente la cantidad de comparaciones.

Por ejemplo, el procedimiento encargado de identificar las palabras claves, empezadas en “L”, sería:

```
procedure TSynMiColor.ProcL;
var tam: integer;
begin
  while Identifiers[linAct[posFin]] do Inc(posFin);
  tam := posFin - posIni;
  if strlcomp(linAct + posIni,'LA',tam) = 0 then fTokenID := tkKey else
  if strlcomp(linAct + posIni,'LOS',tam) = 0 then fTokenID := tkKey else
    fTokenID := tkUnknown;  //sin atributos
end;
```

Se nota, que se reduce considerablemente la cantidad de comparaciones a realizar. De hecho, si las palabras clave a comparar, estuvieran distribuidas uniformemente en el alfabeto, la cantidad de comparaciones se reduciría en 26 veces.

Tal como está este procedimiento, solo detectará las palabras reservadas en mayúscula[2]. Para hacerlo insensible a la caja, se debería agregar un procesamiento adicional.

Aprovechamos esta funcionalidad faltante para optimizar las comparaciones, usando una función de comparación rápida que, además ignore la caja (mayúscula o minúscula). Para ello usaremos nuevamente la invaluable ayuda de las tablas. EL método consistirá en crear una tabla que asigne un ordinal a cada carácter alfabético, independientemente de su caja. A esta tabla la llamaremos "mHashTable", y aprovecharemos para llenarla en "CreaTablaIdentif":

```
procedure CreaTablaIdentif;
var
  i, j: Char;
begin
  for i := #0 to #255 do
  begin
    Case i of
      '_', '0'..'9', 'a'..'z', 'A'..'Z': Identifiers[i] := True;
    else Identifiers[i] := False;
    end;
    j := UpCase(i);
    Case i in ['_', 'A'..'Z', 'a'..'z'] of
      True: mHashTable[i] := Ord(j) - 64
    else
      mHashTable[i] := 0;
    end;
  end;
end;
```

Ahora con esta función creada, ya podemos crear una función, para comparaciones rápidas. Usaremos la que se usan en las librerías de Lazarus:

```
function TSynMiColor.KeyComp(const aKey: String): Boolean;
//Compara rápidamente una cadena con el token actual, apuntado por "fToIden".
//El tamaño del token debe estar en "fStringLen"
var
  I: Integer;
  Temp: PChar;
begin
  Temp := fToIdent;
  if Length(aKey) = fStringLen then
  begin
    Result := True;
    for i := 1 to fStringLen do
    begin
      if mHashTable[Temp^] <> mHashTable[aKey[i]] then
      begin
```

[2] También se puede ver que el reconocimiento de palabras no es efectivo porque reconocerá las palabras aunque solo coincidan en los primeros caracteres.

```
        Result := False;
        break;
      end;
      inc(Temp);
    end;
  end else Result := False;
end;
```

Esta función de comparación usa el puntero “fToIdent” y la variable “fStringLen”, para evaluar la comparación. El único parámetro que requiere es la cadena a comparar.

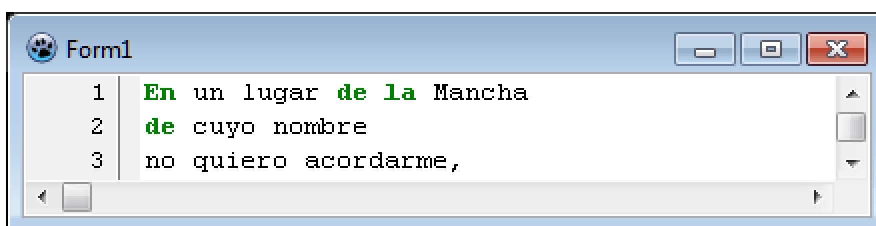
Con la ayuda de la tabla “mHashTable”, la comparación se hará ignorando la caja.

Ahora podemos replantear el procedimiento “ProcL”:

```
procedure TSynMiColor.ProcL;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1;  //calcula tamaño - 1
  fToIdent := linAct + posIni + 1;  //puntero al identificador + 1
  if KeyComp('A')    then fTokenID := tkKey else
  if KeyComp('OS')    then fTokenID := tkKey else
    fTokenID := tkUnknown;  //sin atributos
end;
```

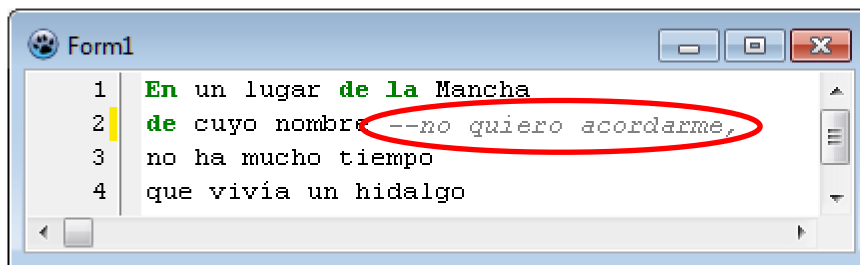
Este procedimiento verifica si se detectan las palabras “LA” o “LOS”. Como una optimización adicional, se omite la comparación del primer carácter, ya que este se ha detectado antes de llamar a esta función.

Ahora ya tenemos lista nuestro resaltador básico. Una prueba del programa con unas palabras claves más, nos dará el siguiente resultado:



1.3.7 Coloreado de comentarios de una sola línea

Se debe hacer en el resaltador. El procedimiento es sencillo. Se debe detectar primero la secuencia de caracteres, que corresponden al inicio del comentario, y luego ubicar el fin de la línea.



En nuestro ejemplo agregamos a nuestra función “MakeMethodTables”, la detección de comentarios, identificando el carácter guion “-”, ya que el token para comentarios de una línea es el doble guion “--”.

```
procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
...
      '-'     : fProcTable[I] := @ProcMinus;
...
    end;
end;
```

Solo podemos detectar un carácter en “`CreaTablaDeMetodos`”, así que es necesario detectar el siguiente carácter en la función “ProcMinus”.

Esta función debe tener la siguiente forma:

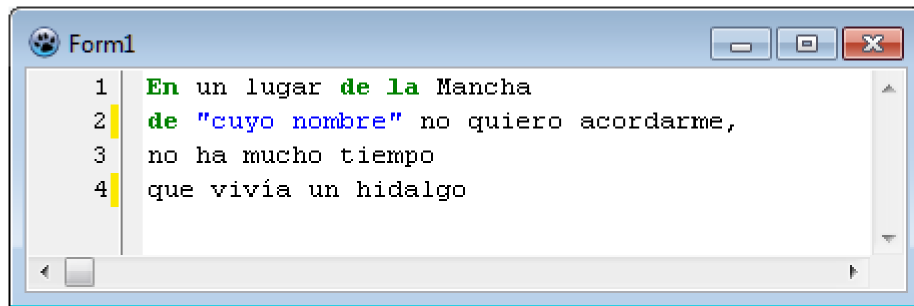
```
procedure TSynMiColor.ProcMinus;
//Procesa el símbolo '-'
begin
  case LinAct[PosFin + 1] of   //ve siguiente caracter
    '-':                       //es comentario de una sola línea
      begin
        fTokenID := tkComment;
        inc(PosFin, 2);        //salta a siguiente token
        while not (linAct[PosFin] in [#0, #10, #13]) do Inc(PosFin);
      end;
  else  //debe ser el operador "menos".
    begin
      inc(PosFin);
      fTokenID := tkUnknown;
    end;
  end
end;
```

Es necesario ver el siguiente carácter, para determinar si se trata del token de comentario. De ser así, se busca el final de línea, para considerar todo ese bloque como un solo token con atributo “tkComent”.

De no ser el token de comentario, simplemente pasamos al siguiente token, marcando el símbolo "-" como de tipo "tkUnknown". Si deseamos considerar al signo "-" en una categoría especial, este es el punto donde debe hacerse.

1.3.8 Coloreado de cadenas

El siguiente caso de coloreado, corresponde al coloreado de cadenas. Este es un caso simple, porque las cadenas ocuparán a lo más una línea. Una cadena tiene un delimitador, que se usa tanto para el inicio como para el fin de la cadena.



En nuestro caso, usaremos las cadenas delimitadas por comillas.

Primero debemos incluir su detección en el procedimiento "CreaTablaDeMetodos":

```
procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
...
      '"'     : fProcTable[I] := @ProcString;
...
    end;
end;
```

Al detectarse el carácter comilla, se pasará el control a "ProcString", quien se encargará de buscar el delimitador final de la cadena, y marcar toda la cadena como un solo token:

```
procedure TSynMiColor.ProcString;
//Procesa el caracter comilla.
begin
  fTokenID := tkString;   //marca como cadena
  Inc(PosFin);
  while (not (linAct[PosFin] in [#0, #10, #13])) do begin
    if linAct[PosFin] = '"' then begin //busca fin de cadena
      Inc(PosFin);
      if (linAct[PosFin] <> '"') then break;  //si no es doble comilla
    end;
    Inc(PosFin);
  end;
```

```
end;
```

Observar que antes de determinar si se ha encontrado el final de la cadena, se verifica primero que no se trate de un caso de doble comilla. Normalmente una doble comilla “entre comillas” representa al carácter comillas.

Se puede deducir también que el token de cadena termina necesariamente en la misma línea donde empezó. Es posible generar coloreado de cadenas de múltiples líneas, como si fueran comentarios multi-líneas, que es el caso que veremos a continuación.

1.3.9 Manejo de rangos

Antes de ver como se implementa el coloreado de rangos, es conveniente conocer el manejo de rangos, en el editor SynEdit.

El uso de rangos permite colorear elementos que pudieran extenderse hasta más allá de una línea. Tal es el caso de los comentarios de múltiples líneas que implementan muchos lenguajes.

Para implementar esta funcionalidad, el editor maneja tres métodos que están definidos en la clase “TSynCustomHighlighter” de la unidad “SynEditHighlighter”:

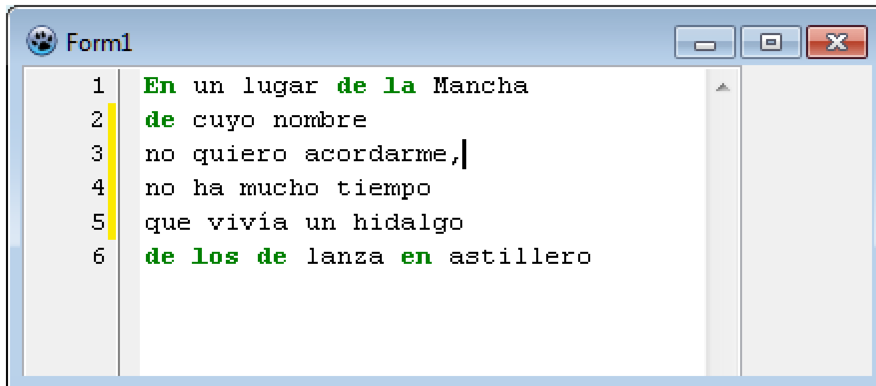
```
  TSynCustomHighlighter = class(TComponent)
   ...
    function GetRange: Pointer; virtual;
    procedure SetRange(Value: Pointer); virtual;
    procedure ResetRange; virtual;
   ...
   end;

function TSynCustomHighlighter.GetRange: pointer;
begin
  Result := nil;
end;

procedure TSynCustomHighlighter.SetRange(Value: Pointer);
begin
end;

procedure TSynCustomHighlighter.ResetRange;
begin
end;
```

Estos métodos se van llamando siempre en la siguiente secuencia:

- ResetRange.- Se ejecuta antes de explorar la primera línea del texto, ya que no hay líneas anteriores que afecten el nivel del rango.
- GetRange.- Es llamado después de terminar la exploración de una línea, para obtener el nivel al final dela línea. Este nivel es almacenado internamente.

- SetRange.- Es llamado antes de la exploración de una línea. Cambia el nivel usando el nivel de la línea anterior.

El siguiente ejemplo muestra un editor, y las llamadas a los métodos de rangos y "SetLine", cuando se modifica la línea 3:




```
1.   SetRange
2. SetLine: no quiero acordarme,
3.   GetRange
4. SetLine: no ha mucho tiempo
5.   GetRange
6. SetLine: que vivía un hidalgo
7.   GetRange
8.   SetRange
9. SetLine: de cuyo nombre
10.  SetRange
11.SetLine: no quiero acordarme,
12.  SetRange
13.SetLine: no ha mucho tiempo
14.  SetRange
15.SetLine: que vivía un hidalgo
```

El editor suele explorar el texto desde una línea antes de la línea modificada, hasta encontrar que una línea devuelve el mismo nivel que tenía anteriormente.

Si no se va a realizar tratamiento de sintaxis, no es necesario sobrescribir estos métodos. Solo deben modificarse, cuando se va a implementar coloreado por rangos, o "folding".

El valor que envía "SetRange", en el parámetro "Value", es un puntero, así como el valor que espera recibir "GetRange", porque han sido diseñados para trabajar con objetos. Pero no es necesario trabajar con punteros. En la práctica se suele usar un tipo enumerado para identificar los niveles de los rangos, teniendo cuidado de hacer las conversiones necesarias.

```
Type
   ...
   TRangeState = (rsUnknown, rsComment);
   ...

TSynMiColor = class(TSynCustomHighlighter)
   ...
   fRange: TRangeState;
```

```
   ...
end;
   ...
function TSynMiColor.GetRange: Pointer;
begin
  Result := Pointer(PtrInt(fRange));
end;

procedure TSynMiColor.SetRange(Value: Pointer);
begin
  fRange := TRangeState(PtrUInt(Value));
end;
```

Las funciones PtrInt y PtrUInt, convierten un puntero a un entero del mismo tamaño que el puntero.

El valor de los punteros no es importante en sí[3], porque no se hace acceso a los objetos apuntados por ellos, en condiciones normales. Lo importante es que tomen valores diferentes cuando se encuentra un rango particular (comentarios, bloques, etc.), de cuando no hay rangos activos.

En el siguiente ejemplo, se ha implementado el coloreado de comentarios (/* … */), y se muestran los valores de los parámetros pasados y leídos, cuando se inserta un comentario a partir de la línea 4:

Form1

```
1  En un lugar de la Mancha
2  de cuyo 'nombre'
3  /* no quiero acordarme,
4  no quiero acordarme,
5  no ha mucho tiempo*/
6  que vivía un hidalgo
7  de los de lanza en astillero
```

```
1.   SetRange: 0
2. SetLine: /*no quiero acordarme,
3.   GetRange: 1
4. SetLine: no quiero acordarme,
5.   GetRange: 1
6. SetLine: no ha mucho tiempo*/
7.   GetRange: 0
```

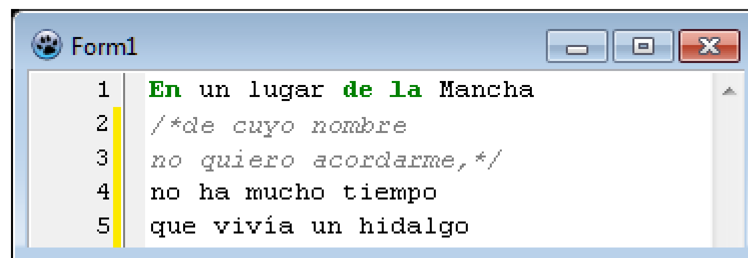
[3] En el diseño de “TSynCustomHighlighter”, se ha definido el uso de punteros, con el fin de poder usar referencias a objetos reales, que puedan asociarse a un rango específico. Para la mayoría de casos, nos bastará con manejar un simple entero o enumerado. Sin embargo, en ciertos desarrollos, como la clase “TSynCustomFoldHighlighter”, si se hace uso de objetos para su funcionalidad. El uso de punteros para manejar rangos diversos, resulta confuso al principio, pues queda la sensación de que hubiera bastado con usar un simple entero que se incrementara y disminuyera, y tal vez así sea, pero el diseño actual permite mayor libertad.

```
8.   SetRange: 0
9. SetLine: de cuyo 'nombre'
10.  SetRange: 0
11.SetLine: /*no quiero acordarme,
12.  SetRange: 1
13.SetLine: no quiero acordarme,
14.  SetRange: 1
15.SetLine: no ha mucho tiempo*/
```

Al lado de “SetRange” o “GetRange”, se está mostrando el ordinal de “fRange”, como una ayuda visual para ver como va cambiando. Aclaramos que el cambio en “fRange” no tiene por qué ser consecutivo, basta con que “fRange” tome valores diferentes, para que la funcionalidad de coloreado, trabaje. La siguiente sección ayudará en la comprensión de rangos, al ver la implementación real.

1.3.10 Coloreado de rango o contexto

El coloreado de contexto, involucra considerar un intervalo del texto, que puede estar en una o varias líneas, como si fuera un único token. Lógicamente, por la jerarquía usada, un token, no puede ser mayor a una línea, por ello, si el rango se extiende a más de una línea, se identificará como varios tokens (uno por cada línea) de la misma categoría.



Dada nuestra clase, este coloreado se puede hacer usando una clase derivada del resaltador usado, o se puede incluir la funcionalidad en el mismo código de la sintaxis original.

Si se desea crear una clase derivada, debe tener una estructura parecida a esta:

```
TSynDemoHlContextFoldBase = class(TSynMiColor)
protected
  FCurRange: Integer;
public
  procedure Next; override;
  function GetTokenAttribute: TSynHighlighterAttributes; override;
public
```

En los métodos “Next” y “GetTokenAttribute”, se debe agregar el comportamiento adicional que se necesita para colorear los rangos de texto.

Para mayor información, se recomienda ver el ejemplo que viene con Lazarus: en \examples\SynEdit\NewHighlighterTutorial\

Sin embargo, la forma más eficiente, sería incluir esta funcionalidad en el mismo resaltador.

Ahora vamos a considerar el caso de colorear un rango de una o varias líneas. Para este ejemplo consideremos el coloreado de comentarios de múltiples líneas.

Primero elegimos los caracteres delimitadores de comentarios. Para nuestro ejemplo usaremos los típicos caracteres de C: "/*" y "*/". Todo el texto que se encuentre entre estos caracteres será considerado como un comentario y tendrá el atributo "fStringAttri".

El delimitador de inicio debemos detectarlo por el carácter "/", pero haciendo la validación, ya que podría tratarse del operador de división.

Nuevamente hacemos la interceptación en la función "`CreaTablaDeMetodos`":

```
procedure TSynMiColor.CreaTablaDeMetodos;
var
  I: Char;
begin
  for I := #0 to #255 do
    case I of
...
      '/'     : fProcTable[I] := @ProcSlash;
...
    end;
end;
```

Y la identificación de comentarios la hacemos en "ProcSlash":

```
procedure TSynMiColor.ProcSlash;
//Procesa el símbolo '/'
begin
  case linAct[PosFin + 1] of
    '*':             //comentario multi-línea
      begin
        fRange := rsComment;      //marca rango
        inc(PosFin, 2);
        CommentProc;    //Procesa en modo comentario
      end;
  else       //debe ser el operador "entre".
    begin
      inc(PosFin);
      fTokenID := tkUnknown;
    end;
  end
end;
```

Observamos que estamos trabajando con un procedimiento "CommentProc" y con una nueva bandera, llamada "fRange". Que se debe declarar como se muestra:

```
Type
   ...
   TRangeState = (rsUnknown, rsComment);
   ...

TSynMiColor = class(TSynCustomHighlighter)
   ...
   fRange: TRangeState;
   ...
```

Esta declaración es importante para el manejo de rangos. La detección de coloreado en rangos, requiere este tipo de manejo.

Se pueden crear diversos tipos de rangos en "TRangeState", de acuerdo a las necesidades de la sintaxis. El orden los enumerados no es importante.

Adicionalmente para implementar la funcionalidad de rangos, se debe sobre-escribir los tres métodos de rangos:

```
TSynMiColor = class(TSynCustomHighlighter)
   ...
   function GetRange: Pointer; override;
   procedure SetRange(Value: Pointer); override;
   procedure ResetRange; override;
   ...
end;

   ...

{Implementación de las funcionalidades de rango}
procedure TSynMiColor.ReSetRange;
begin
  fRange := rsUnknown;
end;

function TSynMiColor.GetRange: Pointer;
begin
  Result := Pointer(PtrInt(fRange));
end;

procedure TSynMiColor.SetRange(Value: Pointer);
begin
  fRange := TRangeState(PtrUInt(Value));
end;
```

Una vez definido el comportamiento de estos métodos. El editor se encargará de la gestión de sus llamadas, ahorrándonos el trabajo de controlar el estado de las líneas.

Pero hace falta aún, procesar las líneas que estén en el rango de comentarios. Para ello, implementamos el método “CommentProc”:

```
procedure TSynMiColor.CommentProc;
begin
  fTokenID := tkComment;
  case linAct[PosFin] of
    #0:
      begin
        ProcNull;
        exit;
      end;
  end;
  while linAct[PosFin] <> #0 do
    case linAct[PosFin] of
      '*':
        if linAct[PosFin + 1] = '/' then
        begin
          inc(PosFin, 2);
          fRange := rsUnknown;
          break;
        end
        else inc(PosFin);
      #10: break;
      #13: break;
    else inc(PosFin);
    end;
end;
```

Este método, explora las líneas en busca del delimitador final del comentario. Si no lo encuentra considera todo lo explorado (inclusive la línea completa) como un solo token de tipo “tkComment”. Hay que notar que no se está detectando el delimitador de fin de comentario “*/” en ninguna otra parte de la clase.

Que debe ser llamado cuando se detecte que estamos en medio de un comentario, como se hace en “ProcSlash”, pero debemos también incluirlo en “Next”:

```
procedure TSynMiColor.Next;
begin
  posIni := PosFin;   //apunta al primer elemento
  if fRange = rsComment then
     CommentProc
  else
    begin
      fRange := rsUnknown;
      fProcTable[linAct[PosFin]]; //Se ejecuta la función que corresponda.
    end;
end;
```

De esta forma, pasamos el control a “CommentProc”, cuando nos encontremos en medio de un comentario de contexto.