

# Lazarus

## Die SynEdit-Bibel

Die Übersetzung entstand aus der original spanischen Fassung und der englischen Übersetzung mittels Deepl und Google Übersetzer. Zusätzlich habe ich so gut wie ich es konnte die Texte nachverbessert. Alle orangen Kommentare wurden von mir eingefügt und kommen im Orginal nicht vor!  
Alle ScreenShots welche von mir eingefügt wurden sind unter Linux Mint entstanden.

**DATENBLATT**

|                          |   |
|--------------------------|---|
| <b>AUTOR</b>             | Tito Hinostroza - Lima Peru   |
| <b>DATUM</b>             | Rev7 abgeschlossen am 11.12.2015  |
| <b>ANWENDBAR FÜR</b>     | Lazarus SynEdit Paket 1.0.12<br>Die Beispiele wurden unter Windows-32 entwickelt.   |
| <b>DOKUMENTENSTUFE</b>   | Halb. Kenntnisse von Free Pascal und Lazarus werden vorausgesetzt.  |
| <b>FRÜHERE DOKUMENTE</b> | Keine   |
| <b>BIBLIOGRAPHIE</b>     | SynEdit-Quellcode - Lazarus<br>SynEdit-Quellcode - SourceForge<br><a href="http://forum.lazarus.freepascal.org/">http://forum.lazarus.freepascal.org/</a><br><a href="http://wiki.freepascal.org/SynEdit/es">http://wiki.freepascal.org/SynEdit/es</a><br><a href="http://wiki.freepascal.org/SynEdit">http://wiki.freepascal.org/SynEdit</a> |

## KONTROLLE ÄNDERN

| VERSION | DATUM      | BESCHREIBUNGSÄNDERUNGEN   |
|---------|------------|---|
| Rev. 1  | 10/12/2013 | Von Tito Hinostroza.<br>Erste vollständig überarbeitete Version der Dokumentation.<br>Dies muss noch dokumentiert werden: <ul style="list-style-type: none"> <li>Die anderen Steuerelemente im SynEdit-Paket.</li> <li>Ausführlicherer Betrieb im Spaltenmodus. (smCurrent)</li> <li>Die Verwendung von Plugins.</li> <li>Die automatische Vervollständigung.</li> </ul>  |
| Rev2    | 10/12/2013 | Von Tito Hinostroza<br>Die Syntax einiger Beispiele wurde korrigiert.<br>Abschnitt 1.4.1 wurde erweitert und die Grafik korrigiert.<br>Informationen zu Abschnitt 1.4.4 hinzugefügt   |
| Rev3    | 10/19/2013 | Von Tito Hinostroza<br>Anhang hinzugefügt und Informationen über den "Hash"-Algorithmus hinzugefügt, der in der Highlighter-Implementierung in Lazarus verwendet wird.<br>Abschnitt 2.3.6 wurde geändert.<br>Abschnitt 1.7.2 hinzugefügt<br>Abschnitt 2.4 wurde geändert  |
| Rev4    | 10/27/2013 | Von Tito Hinostroza<br>Die Eigenschaftstabelle wurde an das Ende übergeben<br>Abschnitt 1.4.9 über die Eigenschaften "Optionen" und "Optionen2" hinzugefügt<br>In Abschnitt 1.4.2 wurden Informationen hinzugefügt und der Abschnitt "Typografie" wurde hinzugefügt.<br>Die Syntaxhervorhebung in Abschnitt 2 wurde neu geordnet und die Einleitung vervollständigt.<br>Informationen über weitere SynEdit-Eigenschaften hinzugefügt. |
| Rev5    | 01/26/2014 | Von Tito Hinostroza<br>Korrigiert einige Wörter mit Fehlern in Abschnitt 1.3<br>Abschnitt 1.4.2 wurde geändert<br>Mehrere Abschnitte wurden geändert und ergänzt.<br>Der Abschnitt "Inhalt ändern" wurde erstellt<br>Informationen zur Erstellung von Attributen in 2.3.4 hinzugefügt<br>Abschnitt 2.4 wird hinzugefügt.<br>Es wurden zusätzliche Informationen zur Codefaltung hinzugefügt.  |
| Rev6    | 04/05/2014 | Typographische Fehler korrigiert.<br>Informationen über die Klassen TSynCustomFoldHighlighter und TSynCustomHighlighter hinzugefügt.<br>Weitere Eigenschaften und Methoden sind in Abschnitt 1.9 enthalten.<br>Abschnitt 1.5.1 wird erstellt  |
| Rev7    | 10/25/2017 | Die Informationen werden in Abschnitt 2.3.4 hinzugefügt.<br>Weitere Informationen über Editor-Koordinaten und einige zusätzliche SynEdit-Eigenschaften wurden hinzugefügt.<br>Es werden Informationen über "Plugins" gespeichert.   |

***"Am Anfang waren es TECO und VI"***

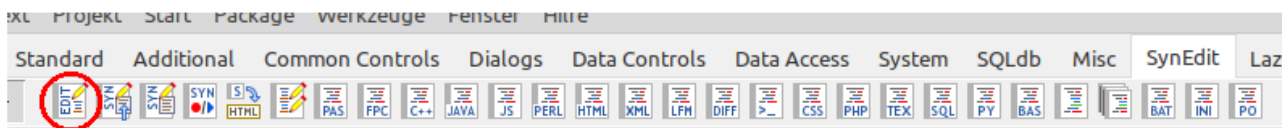
# 1 Syntax-bewusster Editor: SynEdit.

Ein Großteil dieser Arbeit basiert auf Benutzererfahrungen, der begrenzten Dokumentation, die im Internet existiert, Reverse Engineering und der Analyse des Quellcodes der SynEdit-Komponente.

## 1.1 Was ist SynEdit?

Es ist eine Komponente oder ein Steuerelement, das in die Lazarus Umgebung integriert ist. Es ist ein Editier-Steuerelement. Es erlaubt Ihnen, schnell Text-Editoren zu implementieren, mit erweiterten Funktionen wie Syntax-Highlighting.

Um genau zu sein, ist SynEdit ein ganzes Paket, das bereits in Lazarus integriert ist, wenn es installiert wird (und das verschiedene Komponenten enthält), aber im Allgemeinen, wenn wir SynEdit sagen, beziehen wir uns auf die TSynEdit Komponente, die der Editor mit Syntax-Highlighting Fähigkeiten ist.



Es kann über die Registerkarte "SynEdit" der Komponentenpalette aufgerufen werden. Der TSynEdit-Editor befindet sich auf der linken Seite. Auf dieser Registerkarte befinden sich auch verschiedene Steuerelemente, die mit "TSynEdit" zusammenhängen:

- TSynMemo - Version von TSynEdit mit einigen Unterschieden. Sie hat weniger veröffentlichte Methoden und Ereignisse. Abgeleitet von SynEdit. Es kann SynEdit in vielen Fällen ersetzen.
- TSynCompletion - Nicht sichtbares Steuerelement, das die Implementierung der Option "Code Completion" ermöglicht.
- TSynAutoComplete - Nicht sichtbares Steuerelement, das die Implementierung der Option "Auto-Code-Vervollständigung" ermöglicht.
- TSynPasSyn - Syntaxkomponente der Sprache Pascal.
- TSynFreePascalSyn - Syntaxkomponente der Sprache Free Pascal.
- TSynCppSyn - Syntaxkomponente der Sprache C++.
- TSynJavaSyn - Syntaxkomponente der Sprache Java.
- usw.

Das SynEdit Steuerelement, das in Lazarus enthalten ist, ist eine modifizierte Version des eigenständigen SynEdit Projekts. Die für Lazarus angepasste Version wurde aus der Version 1.03 entwickelt, der einige zusätzliche Funktionen hinzugefügt wurden, wie z.B. die Unterstützung für UTF-8 und Code Folding.

Diese Komponente ist gut geprüft und getestet, da sie die gleiche ist, die von der Lazarus IDE für ihren Code Editor verwendet wird.

Leider gibt es nicht genügend technische Dokumentation über das Projekt, aber was bekannt ist, ist, dass es funktionell ist und sehr gut funktioniert.

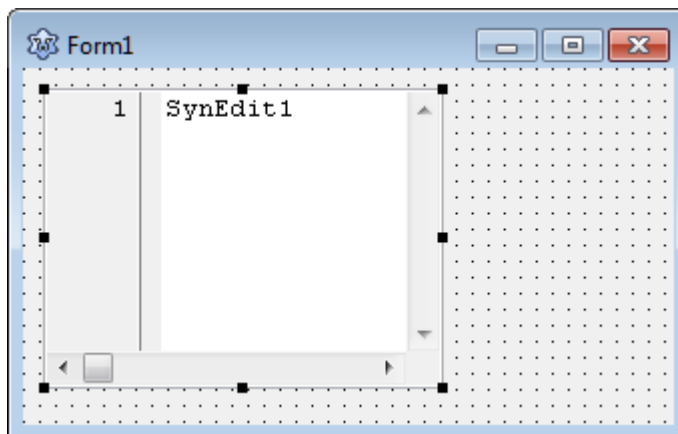
## 1.2 SynEdit-Funktionen

Die Lazarus SynEdit (TSynEdit) Komponente hat die folgenden Eigenschaften:

- Komponente, die von der Lazarus IDE aus zugänglich ist.
- Es erfordert keine zusätzlichen Dateien (wie es bei Scintilla der Fall ist). Sobald es in das Projekt integriert ist, wird es ohne Abhängigkeiten in den Code integriert.
- Sein Code ist vollständig zugänglich und veränderbar.
- Funktioniert vollständig in UTF-8-Kodierung.
- Unterstützt Syntaxfärbung für mehrere vordefinierte Sprachen oder Sie können eine neue Syntax erstellen.
- Unterstützt Optionen zur Code-Vervollständigung und automatischen Vervollständigung.
- Unterstützt die Code-Faltung. Aber es muss durch Code erfolgen.
- Enthält "Rückgängig"- und "Wiederherstellen"-Optionen mit umfassender Speicherung der Änderungen.
- Enthält Methoden für Suchen und Ersetzen.
- Unterstützt die einfache Spaltenauswahl.
- Ermöglicht es Ihnen, die Zeilen zu nummerieren.
- Unterstützt Highlighter und Textbausteine.

## 1.3 Erscheinungsbild

Durch Hinzufügen der TSynEdit-Komponente zum Formular ist dieses nun funktionsfähig. Sie können das Programm ausführen und sehen, dass der Editor wie ein beliebiges Textfeld vom Typ TMemo reagiert.



Der größte visuelle Unterschied besteht in der vertikalen Leiste, die links erscheint. Diese Leiste dient zur Anzeige der Zeilennummer und anderer Optionen. Ein weiterer Unterschied besteht darin, dass die horizontale Schriftgröße einheitlich ist. Das heißt, der Buchstabe "m" hat die gleiche Breite wie der Buchstabe "l". Dies ist die Schriftart, die standardmäßig in einem "SynEdit" geladen wird. SynEdit enthält zunächst keine Optionen zur Syntaxhervorhebung, da es noch keine zugehörige Syntax hat. Was es standardmäßig enthält, ist die Erkennung von "Klammern", d.h. es hebt die Klammern hervor, die sich öffnen und schließen, wenn der Cursor in einer der Klammern steht. Ähnlich verhält es sich mit eckigen Klammern, geschweiften Klammern und Anführungszeichen. Apostrophe werden nicht erkannt. Diese Hervorhebung besteht standardmäßig darin, dass die Anfangs- und Endzeichen fett dargestellt werden.

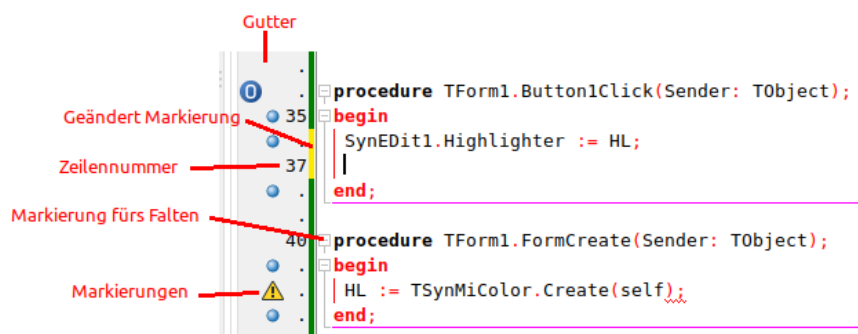
**(Text in Klammern (anderer Text)) mehr Text**

Um diese Funktion zu deaktivieren, muss die Option "eoBracketHighlight" aus der Eigenschaft "Optionen" entfernt werden.

Wenn Sie das Attribut zur Hervorhebung des Begrenzungszeichens ändern möchten, können Sie den folgenden Code verwenden:

```
SynEdit1.BracketMatchColor.Foreground := clRed ; // wechselt zu rot
```

Eine weitere Funktion, die standardmäßig in SynEdit enthalten ist, ist die Möglichkeit, Lesezeichen zu erstellen (**siehe - Textmarkierungen**). Wenn sie nicht verwendet werden soll, muss diese Option deaktiviert werden, da sie zur Laufzeit Fehler verursachen könnte. Die Optionen Ausschneiden, Kopieren und Einfügen sind ebenfalls standardmäßig im SynEdit-Steuerelement aktiviert, ohne dass sie implementiert werden müssen. Im Allgemeinen entsprechen alle Verknüpfungen, die standardmäßig in SynEdit erstellt werden, Aktionen, die vordefiniert sind, ohne dass sie aktiviert werden müssen. Es gibt mehrere Eigenschaften, um das Aussehen des SynEdit-Steuerelements zu ändern. Wir werden einige von ihnen beschreiben.



Die "Gutter" kann per Code ein- oder ausgeblendet werden. Um sie unsichtbar zu machen, sollten Sie dies tun:

```
SynEdit1.Gutter.Visible := False ;
```

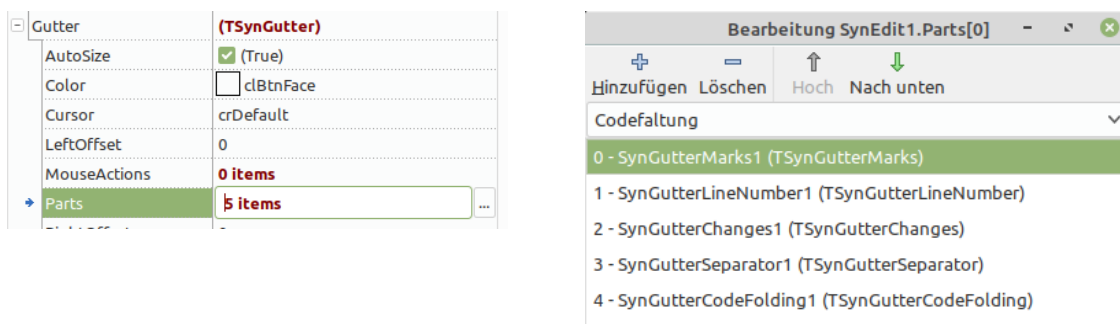
In diesem Fall hat unser Editor den Standardnamen, der zugewiesen wird, wenn er einem Formular hinzugefügt wird: SynEdit1.

Der "Gutter" wird standardmäßig automatisch in seiner Breite angepasst, d.h. er ändert sich entsprechend der Anzahl der Zeilen im Editor. Er kann auf eine bestimmte Breite eingestellt werden, indem man die Eigenschaft "AutoSize" auf "false" setzt:

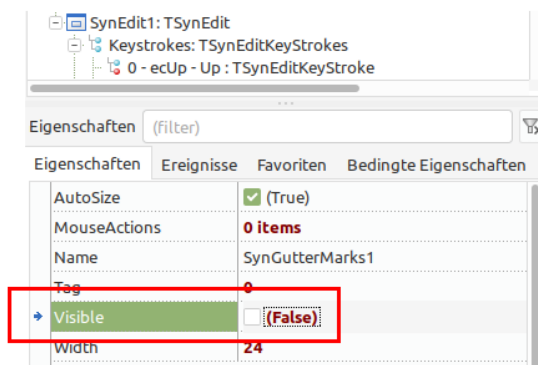
```
SynEdit1.Gutter.AutoSize := false ;  
SynEdit1.Gutter.Width:= 30;
```

Es ist nicht ratsam, die Breite auf diese Weise zu ändern, da auf diese Weise die darin enthaltenen Elemente nicht verschoben werden, so dass ein Teil der Zahlen oder "Faltzeichen" aus dem Blickfeld geraten könnte.

Es ist besser, die automatische Größeneinstellung auf "true" zu belassen und einzelne Elemente der "Gutter" zu deaktivieren, um ihre Größe zu variieren. Dies lässt sich leicht mit dem Objektinspektor bewerkstelligen, indem die Eigenschaft "Parts" der Eigenschaft "Gutter" geändert wird:



Und dann das gewünschte Element ausblenden:



In diesem Beispiel wird der für die Marker vorgesehene Bereich ausgeblendet. Wenn Bereiche ausgeblendet werden, verringert sich die Gesamtgröße des "Gutters".

### 1.3.1 Rechter Rand

Standardmäßig erscheint in SynEdit eine vertikale Linie auf der rechten Seite des Textes, normalerweise in Spalte 80. Diese Linie ist hilfreich, wenn Sie den Inhalt ausdrucken möchten und vermeiden wollen, dass die vom Drucker erlaubte Zeilengröße überschritten wird.



Um die Position zu ändern, müssen Sie die Eigenschaft "RightEdge" ändern:

```
SynEdit1.RightEdge := 100; //fixiert die Position der vertikalen Linie
```

Sie können die Farbe auch mit der Eigenschaft "RightEdgeColor" ändern.

```
SynEdit1.RightEdgeColor:= clBlue;
```

Wenn Sie nicht möchten, dass diese Linie erscheint, können Sie ihre Position auf eine negative Koordinate setzen:

```
SynEdit1.RightEdge := - 1 ; // vertikale Linie ausblenden
```

Sie kann auch mit der Option "eoHideRightMargin" in der Eigenschaft "Options" deaktiviert werden (uses ....., SynEditTypes):

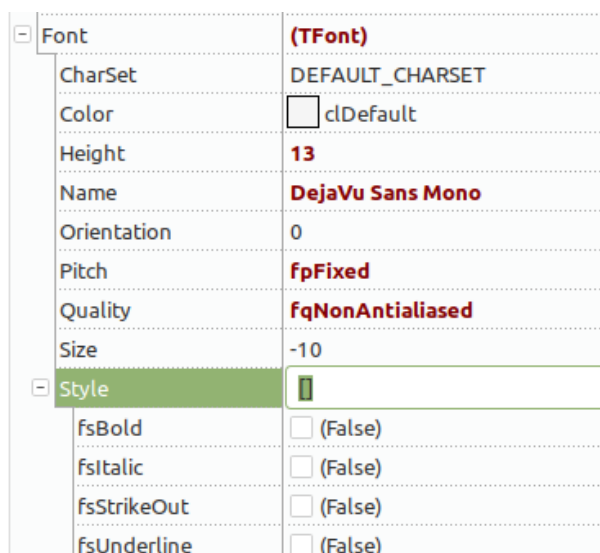
```
SynEdit1.Options := SynEdit1.Options + [eoHideRightMargin] ;
```

Die Bildlaufleisten des Editors können mit der Eigenschaft "ScrollBars" ein- oder ausgeblendet werden.

### 1.3.2 Typografie

Mit SynEdit können Sie verschiedene Eigenschaften der zu verwendenden Schriftart konfigurieren. Standardmäßig wird der Text mit der Schriftart "Courier New" in Größe 10 angezeigt (*bei mir unter Linux ist es DejaVu Sans Mono Größe 10*).

Um die Schriftart zu ändern, die ohne SynEdit verwendet werden soll, muss das Font-Objekt konfiguriert werden. Diese Aufgabe kann durch Code oder mit dem Objektinspektor erledigt werden:

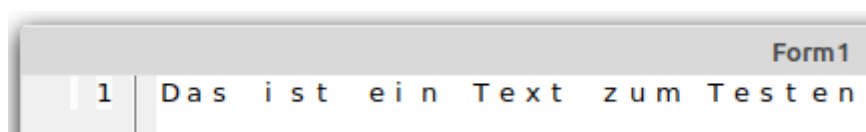




Das Font-Objekt verfügt über verschiedene Eigenschaften und Methoden, von denen viele nur über Code zugänglich sind.

Die vielleicht häufigste Eigenschaft zur Änderung des Erscheinungsbildes von Text auf dem Bildschirm ist die Schriftart. Diese kann mit der Eigenschaft "Name" geändert werden. Beachten Sie, dass die in SynEdit anzuzeigenden Zeichen immer monospaced sind, d.h. alle Zeichen haben die gleiche Breite auf dem Bildschirm. Wenn eine Schriftart verwendet wird wo für jedes Zeichen eine anderen Breite verwendet wird, zeigt SynEdit sie genauso an wie eine monospaced Schriftart, wodurch der Eindruck entsteht, dass die Zeichen nicht gleichmäßig verteilt sind.

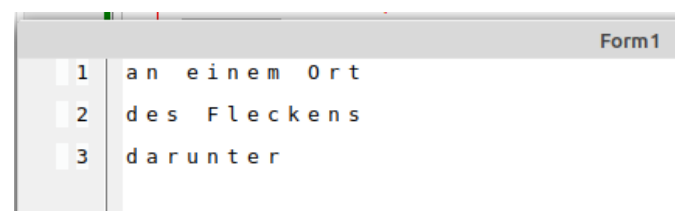
Das folgende Beispiel zeigt einen Editor, in dem die Schriftart "Sans", die nicht monospaced ist, verwendet wurde:



Der Text wurde korrekt geschrieben und getrennt, aber da die Buchstaben in "Sans" unterschiedlich breit sind (das "D" ist breiter als das "i"), scheint der Text nicht richtig getrennt zu sein. Wegen dieses Effekts wird empfohlen, nur Schriftarten mit einheitlicher Breite zu verwenden, z. B. Courier, Fixed oder Lucida.

Die Schriftgröße wird mit der Eigenschaft "Height" und die Farbe mit der Eigenschaft "Color" festgelegt. Mit der Eigenschaft "Style" können Sie die Attribute fett, unterstrichen und kursiv festlegen.

Mit den Eigenschaften "ExtraCharSpacing" und "ExtraLineSpacing" ist es auch möglich, den Abstand zwischen Zeilen und zwischen Zeichen zu ändern:



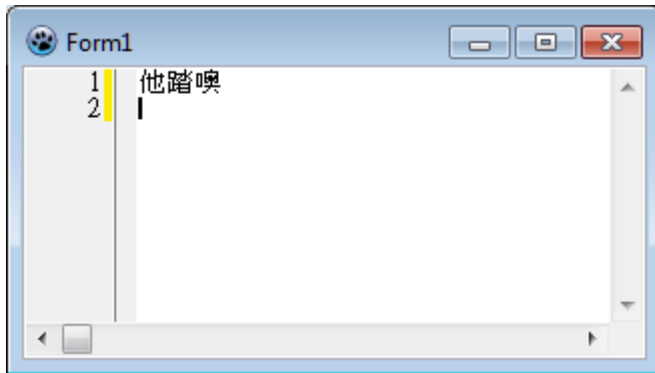
Die vorstehende Abbildung wurde mit Hilfe des folgenden Codes erstellt:

```
SynEdit1.ExtraCharSpacing := 5 ;  
SynEdit1.ExtraLineSpacing := 10 ;
```

Standardmäßig ist der Abstand Null. Wenn Sie eine Verbindung statt einer Trennung wünschen, können Sie für diese Eigenschaften negative Werte verwenden.

Eine weitere Eigenschaft, die wir verwenden können, um SynEdit anzupassen, ist der Zeichensatz (CharSet).

Mit dem Zeichensatz können Sie die Sprache ändern, die im Editor verwendet werden soll. Mit dem Zeichensatz CHINESEBIG5\_CHARSET könnten wir zum Beispiel chinesische Zeichen verwenden:



Einige Zeichensätze, wie der im Beispiel, benötigen doppelt so viel Platz für ein Zeichen im Editor wie traditionelle westliche Zeichen. Kein Problem, SynEdit kann diese Arten von Zeichen verarbeiten und sogar Zeichen mit einfachem und doppeltem Leerzeichen (oder zwei verschiedene Zeichensätze) im selben Dokument kombinieren.

(Hat bei mir weder unter Linux noch Windows funktioniert, eventuell müsste ich da noch das passende Charset installieren)

## 1.4 Funktionsweise

### 1.4.1 Editor Koordinaten

Bei SynEdit ist der Bildschirm ein Gitter aus Zellen, wobei jede Zelle ein Zeichen darstellt<sup>1</sup> :

|       |       |     |       |  |  |  |  |
|-------|-------|-----|-------|--|--|--|--|
| (1,1) | (2,1) | ... |       |  |  |  |  |
| (1,2) | (2,2) |     |       |  |  |  |  |
| ...   |       |     |       |  |  |  |  |
|       |       |     |       |  |  |  |  |
|       |       |     | (x,y) |  |  |  |  |
|       |       |     |       |  |  |  |  |
|       |       |     |       |  |  |  |  |
|       |       |     |       |  |  |  |  |
|       |       |     |       |  |  |  |  |

← Zelle

Jedes Feld wird  
durch seine  
Koordinate  
(Reihe, Spalte)  
dargestellt.

Andererseits werden die auf dem Bildschirm anzuzeigenden Informationen in einer Liste von Zeichenfolgen gespeichert, wobei jede Zeichenfolge eine Zeile darstellt.

Um beide Darstellungen zu handhaben, werden in einem SynEdit zwei Arten von Koordinaten verwendet:

- **Physikalische Koordinaten.** Bezieht sich auf die Position, an der ein Zeichen auf dem Bildschirm erscheint, wobei davon ausgegangen wird, dass der Bildschirm in Zellen mit gleicher Breite unterteilt ist.
- **Logische Koordinaten.** Bezieht sich auf die Position des Bytes (oder der Bytes), die das Zeichen in der Zeichenkette darstellen.

Dieser Unterschied macht sich vor allem dadurch bemerkbar, dass SynEdit mit UTF-8-Kodierung arbeitet, was die Verwaltung der Koordinaten auf dem Bildschirm erschwert.

Zeichenketten werden als Byte-Folgen gespeichert, aber was auf dem Bildschirm angezeigt wird, sind Zeichenfolgen, die sich in Zellen befinden.

Die Entsprechung von Bytes zu Zeichen ist nicht 1 zu 1<sup>2</sup>.

- Ein Byte in der Zeichenkette kann mehr als ein Zeichen auf dem Bildschirm darstellen<sup>3</sup>. Dies ist der Fall bei der Verwendung von Tabulatoren, die zu mehreren Leerzeichen erweitert werden müssen.
- Ein Zeichen auf dem Bildschirm kann durch mehr als ein Byte in der Zeichenkette dargestellt werden. Dies liegt daran, dass SynEdit die universelle UTF-8-Kodierung verwendet, die in einigen Fällen (z. B. bei betonten Vokalen) mehr als ein Byte pro Zeichen zuweist.

<sup>1</sup> Dies ist nicht ganz richtig, denn einige orientalische Zeichen können zwei Editorzellen (in voller Breite) belegen.

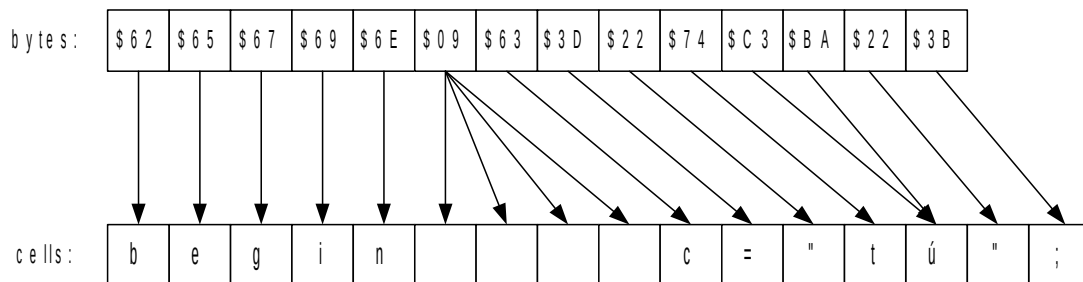
<sup>2</sup> Ursprünglich wurde in den ersten Texteditoren immer eine 1:1-Entsprechung beibehalten (es sei denn, es wurden Tabellierungen unterstützt), wobei ASCII-Kodierung oder etwas Ähnliches verwendet wurde.

<sup>3</sup> Wenn man bedenkt, dass ein Zeichen auf dem Bildschirm zwei Zellen im Editor belegen kann, wird die Sache noch komplizierter: Im Allgemeinen können ein oder mehrere Stringbytes eine oder zwei Zellen auf dem Bildschirm darstellen.

Das folgende Schema zeigt, wie ein typischer Textstring in SynEdit kodiert wird:

|        |    |    |    |    |    |    |  |  |  |    |    |    |    |
|--------|----|----|----|----|----|----|--|--|--|----|----|----|----|
| bytes  | 62 | 65 | 67 | 69 | 6E | 09 |  |  |  | 63 | 3D | 22 | 74 |
| celdas | b  | e  | g  | i  | n  |    |  |  |  | c  | =  | "  | t  |

Der Tabstopp wird als 4 Leerzeichen dargestellt, kann aber je nach den Einstellungen des Editors variieren. Es sollte auch beachtet werden, dass in UTF-8 das Zeichen "ú" durch die Bytes \$C3 und \$BA<sup>4</sup> dargestellt wird. Eine andere Art, diese Entsprechung zu betrachten, wäre wie folgt:



Sie können deutlich sehen, wie sich die logischen Koordinaten von den physischen Koordinaten unterscheiden. Hier sehen wir, dass die logische X-Koordinate des Buchstabens "c" 7 ist, aber seine physische Koordinate ist 10.

In vertikaler Richtung sind die logische und die physische Y-Koordinate immer gleich, so dass keine Transformationen vorgenommen werden müssen.

Der Cursor arbeitet immer in physikalischen Koordinaten. Die Koordinaten des Cursors sind in den Eigenschaften CaretX und CaretY enthalten. CaretX geht von 1 bis zum Ende der Zeile. CaretY geht von 1 bis zur Anzahl der Zeilen.

Um zum Beispiel den Cursor auf das fünfte Zeichen der zweiten Zeile zu positionieren, würden wir dies tun:

```
SynEdit1.CaretX := 5 ;
SynEdit1.CaretY := 2 ;
```

Sie können auch die Eigenschaft CaretXY verwenden, die die beiden Koordinaten X und Y in einer Struktur vom Typ TPoint enthält:

```
var Pos : TPoint ;
...
Pos.x := 5 ;
Pos.y := 2 ;
SynEdit1.CaretXY := Pos ; // Äquivalent zu SynEdit1.CaretXY := Point ( 5,2 ) ;
```

<sup>4</sup> Aufgrund der Ungleichheit der Zeichengröße in UTF-8 gibt es spezielle Funktionen für die Behandlung von Zeichenketten in UTF-8, wie Utf8Length und Utf8Pos.

Sie können auch auf die logischen Koordinaten des Cursors zugreifen, indem Sie die Eigenschaft: `SynEdit1.LogicalCaretXY` verwenden. Wenn wir also logische Koordinaten haben, um den Cursor in SynEdit korrekt zu positionieren, können wir dies tun:

```
SynEdit1.LogicalCaretXY := Point ( 5, 2 );
```

Um Transformationen zwischen logischen und physikalischen Koordinaten durchzuführen, gibt es eine Gruppe von Transformationsfunktionen:

```
SynEdit1.LogicalToPhysicalCol ();  
SynEdit1.LogicalToPhysicalPos ();  
SynEdit1.PhysicalToLogicalCol ();  
SynEdit1.PhysicalToLogicalPos ();
```

Normalerweise werden wir diese Funktionen nicht benötigen, es sei denn, SynEdit enthält UTF-8-Zeichen mit mehr als einem Byte oder Tabulatoren, da die logischen und physischen Koordinaten normalerweise übereinstimmen.

Betrachten wir ein Beispiel, bei dem wir Inhalte mit Umlauten haben (sie sind in UTF-8 mit 2 Bytes kodiert). Wenn wir in unserem Editor den folgenden Text in der ersten Zeile haben:

"Müslī"

Und wir wollen das dritte Zeichen erhalten (das der Buchstabe "s" sein muss). Um die tatsächliche Position des Zeichens in der Zeichenkette zu ermitteln, müssen wir auf die Position zugreifen:

```
SynEdit1.CaretX := 3 ;  
SynEdit1.CaretY := 1 ;  
xReal := SynEdit1.PhysicalToLogicalPos ( SynEdit1.CaretXY ) .x;  
showmessage(inttostr(xReal));
```

In `xReal` erhalten wir die tatsächliche Position des Zeichens innerhalb der Zeichenkette, in unserem Fall 4.

In bestimmten Fällen kann es nützlich sein, die Koordinaten des Cursors in Pixeln zu kennen. In diesem Fall müssen die Eigenschaften verwendet werden:

```
SynEdit1.CaretXPix  
SynEdit1.CaretYPix
```

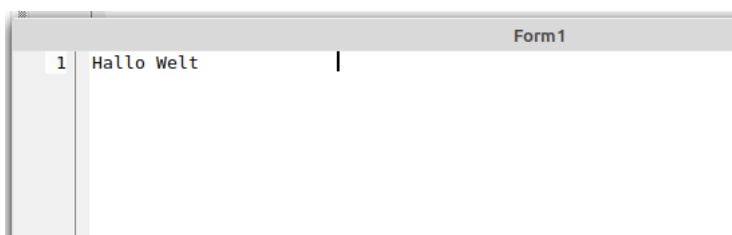
Die `CaretXPix`-Koordinate wird vom linken Rand des Steuerelements aus gemessen (einschließlich der Breite des vertikalen Bereichs oder der Randleiste) und nicht vom bearbeitbaren Bereich des Editors.

## 1.4.2 Handhabung des Cursors

Wie wir bereits gesehen haben, wird die Position des Cursors mit den Eigenschaften CaretX und CaretY festgelegt, aber es gibt bestimmte Grenzen für die Position des Cursors.

In der Praxis kann man sagen, dass es keine praktische Grenze für die Größe einer Zeile in SynEdit gibt, es sei denn, Sie wollen 2 Milliarden überschreiten. Aber der Cursor hat mehr Einschränkungen.

Eine seltsame Funktionsweise von SynEdit besteht darin, dass es den Cursor über die Zeilengrenzen hinaus platzieren kann. Wenn die Zeile z. B. nur 10 Zeichen breit ist, kann der Cursor an Position 20 platziert werden:



Diese seltsame Platzierung des Cursors kann entweder über die Tastatur oder mit der Maus erfolgen.

Ich nenne diesen Effekt "Floating Cursor" und es ist eine ungewöhnliche Funktion in Texteditoren, da der Cursor normalerweise nicht über die Grenzen der Zeile hinaus platziert werden kann.

Dieser Effekt ist nur horizontal gültig, da der Cursor vertikal immer durch die Anzahl der Zeilen im Text begrenzt ist, oder anders gesagt, die Y-Koordinate des Cursors kann nicht größer sein als die Anzahl der Zeilen.

Um die horizontale Position des Cursors innerhalb der physikalischen Grenzen der aktuellen Zeile zu begrenzen, sollte diese Einstellung verwendet werden:

```
SynEdit1.Options := SynEdit1.Options + [eoKeepCaretX] ;
```

Der "Floating Cursor"-Modus kann jedoch durch die Eigenschaft "MaxLeftChar" eingeschränkt werden. Wenn für "MaxLeftChar" ein Maximalwert festgelegt wird, ist eine horizontale Platzierung über diesen voreingestellten Wert hinaus nur dann zulässig, wenn eine Zeile existiert, die diese Größe überschreitet. Das heißt, "MaxLeftChar" begrenzt nicht die Größe der aktuellen Zeile, aber es kann die horizontale Position des Cursors begrenzen.

Um die Positionierung des Cursors ohne Berücksichtigung der Größe der Zielzeile zu erzwingen, kann die Methode MoveCaretIgnoreEOL() verwendet werden. Diese Methode funktioniert auch, wenn "MaxLeftChar" überschritten wird.

Eine weitere, etwas merkwürdige Eigenheit von SynEdit besteht darin, dass es Ihnen erlaubt, den Cursor in der Mitte eines Tabulatorstopps zu positionieren, als ob es sich um einfache Leerzeichen handeln würde, wodurch der Eindruck entsteht, dass es an dieser Stelle keinen Tabulatorstopp gibt.

Um das Vorhandensein eines solchen Tabstopps festzustellen, können Sie versuchen, ein Feld innerhalb des Tabstoppbereichs auszuwählen. Wenn die Auswahl nicht möglich ist, wird angezeigt, dass Sie sich innerhalb eines Tabstopps befinden.

Wenn Sie dieses Verhalten deaktivieren und erzwingen möchten, dass Tabulatoren im Editor als ein einzelnes Zeichen behandelt werden, müssen Sie die Option "eoCaretSkipTab" in der Eigenschaft Options2 aktivieren:

```
SynEdit1.Options2 := SynEdit1.Options2 + [eoCaretSkipTab] ;
```

Die Position des Cursors kann mit Hilfe der Eigenschaften beliebig verändert werden: CaretX, CaretY oder CaretXY, wie bereits erwähnt, aber Sie können den Cursor auch über Befehle positionieren ([siehe Abschnitt 1.5.1 - Befehle Befehle ausführen](#)).

Der folgende Code zeigt, wie man den Cursor am Ende des gesamten SynEdit-Textes positioniert:

```
SynEdit1.ExecuteCommand ( ecEditorBottom, ' ', nil ); //uses ynEditKeyCmds
```

Um den Cursor wie mit den Pfeiltasten zu bewegen, müssen die Befehle "ecLeft", "ecRight", "ecUp" und "ecDown" verwendet werden.

### 1.4.3 Zeilen Begrenzungszeichen

Wie in vielen Bereichen der Informatik gibt es auch hier keinen Konsens darüber, wie ein Zeilenumbruch im Text definiert wird. Derzeit gibt es 3 bekannte Möglichkeiten, eine Zeile abzugrenzen:

- Format ZWEI: Zeichen #13#10. Wird auf DOS/Windows-Systemen verwendet.
- Unix-Format: Zeichen #10. Wird auf Linux/Unix-Systemen verwendet.
- MAC-Format: Zeichen #13. Wird in MAC-Systemen verwendet.

Beim Lesen von Text aus einer Datei mit SynEdit1.Lines.LoadFromFile() wird jedes dieser Begrenzungszeichen erkannt und die Zeilen werden korrekt geladen.

Wenn Sie jedoch den Inhalt mit SynEdit1.Lines.SaveToFile() speichern wollen, kann die ursprüngliche Kodierung verloren gehen, abhängig vom verwendeten Betriebssystem oder der Systemkonfiguration.

Um den mit SaveToFile() verwendeten Zeilenvorschub (DOS, UNIX, MAC) zu ändern, müssen Sie die Unit SynEditLines verwenden und die Methode FileWriteLineEndType der Klasse TSynEditLines ausführen:

```
uses..., SynEditLines;  
...  
if Typ = 'TWO' then TSynEditLines ( SynEdi1.Lines ) .FileWriteLineEndType := sfleCrLf ;  
if Typ = 'UNIX' then TSynEditLines ( SynEdi1.Lines ) .FileWriteLineEndType := sfleLf ;  
if Typ = 'MAC' then TSynEditLines ( SynEdi1.Lines ) .FileWriteLineEndType := sfleCr ;
```

Eine andere Möglichkeit ist, die Eigenschaft "TextLineBreakStyle" der Liste "SynEdit1.Lines" zu ändern. Dieses Trennzeichen wird jedoch nur wirksam, wenn die Eigenschaft "Text" der Liste verwendet wird.

Zum Beispiel:

```
SynEdit1.Lines.TextLineBreakStyle:=tlbsLF;  
SynEdit1.Lines.Text:='Ich bin ein Text .....';
```



## 1.5 Ändern Sie den Inhalt

Der Inhalt des SynEdit-Editors wird in der Regel vom Benutzer über die Tastatur geändert, aber oft müssen wir die Kontrolle über den Editor aus dem Programm heraus übernehmen. Hier beschreiben wir die verschiedenen Möglichkeiten, auf den Inhalt des Editors zuzugreifen.

Um den gesamten Inhalt des Editors zu löschen, müssen Sie die Methode "ClearAll" verwenden:

```
SynEdit1.ClearAll ;
```

Um in SynEdit zu schreiben, ist es am einfachsten, der Eigenschaft Text einen Text zuzuweisen:

```
SynEdit1.Text := 'Hallo Welt' ;
```

Wir können auch Zeilenumbrüche zuweisen:

```
SynEdit1.Text := 'Hallo' + #13#10 + 'Welt' ;
```

Wenn wir auf diese Weise schreiben, gehen alle vorherigen Informationen, die das Steuerelement möglicherweise enthält, verloren (ohne Rückgängig-Option), da wir einen neuen Wert zuweisen. Wenn wir nur Informationen hinzufügen wollten, könnten wir das tun:

```
SynEdit1.Text := SynEdit1.Text + 'Hallo Welt' ;
```

Die Eigenschaft Text ist eine einfache Zeichenkette und ermöglicht es uns, den Inhalt des Editors zu lesen oder zu schreiben. Da es sich um einen Kettentyp handelt, können mit ihm dieselben Operationen durchgeführt werden, die auch mit Strings durchgeführt werden (Suche, Verkettung, usw.).

Ein Detail, das Sie beachten sollten, ist, dass Sie beim Lesen des Inhalts von SynEdit über die Eigenschaft "Text" einen zusätzlichen Zeilenumbruch am Ende des Textes erhalten.

Eine weitere Möglichkeit, Text in den Editor einzufügen, ist die Methode InsertTextAtCaret():

```
SynEdit1.InsertTextAtCaret ('eingefügter Text' );
```

Mit InsertTextAtCaret() wird der Text direkt an der Stelle eingefügt, an der sich der Cursor befindet. Besteht eine aktuelle Auswahl, wird die Auswahl aufgehoben (der markierte Text wird nicht entfernt) und der angegebene Text an der Cursorposition eingefügt.

Der eingefügte Text kann aus einer oder mehreren Zeilen bestehen.

Eine weitere nützliche Funktion zum Ändern des Inhalts von SynEdit ist TextBetweenPoints(). Das folgende Beispiel zeigt einen schnellen Weg, um ausgewählten Text durch einen neuen zu ersetzen:

```
SynEdit1.TextBetweenPoints[SynEdit1.BlockBegin, SynEdit1.BlockEnd] := 'Neuer Text' ;
```

Informationen zu BlockBegin und BlockEnd finden Sie im Abschnitt - [Verwaltung der Auswahl](#). Eine andere Möglichkeit wäre die Verwendung der Methode TextBetweenPointsEx(), die mehr Möglichkeiten zur Steuerung des Cursors nach der Ersetzung bietet.

### 1.5.1 Befehle ausführen

Der Editor kann auch durch die Verwendung von Befehlen mit der Methode "ExecuteCommand" gesteuert werden.

Praktisch alles, was Sie im Editor mit der Tastatur machen können, können Sie auch mit Befehlen machen. Um zum Beispiel das Zeichen "x" an der aktuellen Cursorposition einzufügen, verwenden Sie:

```
SynEdit1.ExecuteCommand ( ecChar, 'x' , nil );
```

Es gibt eine große Gruppe von Befehlen, die in SynEdit eingegeben werden können. Sie alle werden in der Unit "SynEditKeyCmds" deklariert. Einige von ihnen werden hier gezeigt:

```
ecLeft  = 1 ; // Cursor um ein Zeichen nach links bewegen
ecRight = 2 ; // Cursor um ein Zeichen nach rechts bewegen
ecUp    = 3 ; // Cursor eine Zeile nach oben bewegen
ecDown  = 4 ; // Cursor eine Zeile nach unten bewegen

ecDeleteLastChar = 501 ; // Letztes Zeichen löschen (d.h. Rückschritttaste)
ecDeleteChar     = 502 ; // Zeichen am Cursor löschen (d.h. Löschtaste)
ecDeleteWord     = 503 ; // Löschen vom Cursor bis zum Ende des Wortes
ecDeleteLastWord = 504 ; // Löschen vom Cursor bis zum Wortanfang
ecDeleteBOL      = 505 ; // Löschen vom Cursor bis zum Anfang der Zeile
ecDeleteEOL      = 506 ; // Löschen vom Cursor bis zum Ende der Zeile
ecDeleteLine     = 507 ; // Aktuelle Zeile löschen
ecClearAll       = 508 ; // Alles löschen
ecLineBreak      = 509 ; // Zeilenumbruch an der aktuellen Position, Cursor auf
                        // neue Zeile setzen
ecInsertLine     = 510 ; // Zeilenumbruch an der aktuellen Position, Caret stehen
                        // lassen
ecChar           = 511 ; // Einfügen eines Zeichens an der aktuellen Position
ecSmartUnindent  = 512 ; // NICHT als Befehl registriert, wird für Gruppen -
                        // Rückgängig machen verwendet, wird von beautifier gesetzt

ecImeStr         = 550 ; // Zeichen (s) aus IME einfügen

ecundo           = 601 ; // Rückgängig machen, falls vorhanden
ecRedo           = 602 ; // Redo durchführen, falls vorhanden
ecCut            = 603 ; // Auswahl in Zwischenablage ausschneiden
ecPaste          = 604 ; // Einfügen der Zwischenablage an der aktuellen Position

ecBlockIndent    = 610 ; // Auswahl einrücken
ecBlockUnindent  = 611 ; // Einrückung aufheben
ecTab            = 612 ; // Tabulator-Taste
ecShiftTab       = 613 ; // Umschalttaste + Tabulator-Taste
...

```

Wie Sie sehen können, lassen sich mit den Befehlen alle möglichen Aktionen durchführen, z. B. Text einfügen, ein Zeichen löschen oder Änderungen rückgängig machen.

Mit den ExecuteCommand()-Befehlen können Sie alle Arten von Änderungen in SynEdit vornehmen, aber es ist eine langsame Art der Änderung, da sie Zeichen für Zeichen durchgeführt wird.

Zu den durchführbaren Aktionen gehören auch die Verwaltung von Lesezeichen und das Falten von Textblöcken. Die folgenden Befehle dienen dieser Funktion:

```
ecGotoMarker0 = 301 ; // Gehe zur Markierung
ecGotoMarker1 = 302 ; // Springender Marker
ecGotoMarker2 = 303 ; // Gehe zur Markierung
ecGotoMarker3 = 304 ; // Sprungmarke (goto marker)
ecGotoMarker4 = 305 ; // Sprungmarke (goto marker)
ecGotoMarker5 = 306 ; // goto-Marker
ecGotoMarker6 = 307 ; // Sprungmarke (goto marker)
ecGotoMarker7 = 308 ; // Sprungmarke (goto marker)
ecGotoMarker8 = 309 ; // Sprungmarke (goto marker)
ecGotoMarker9 = 310 ; // Sprungmarke (goto marker)
ecSetMarker0 = 351 ; // Markierung setzen, Data = PPoint - X, Y Pos
ecSetMarker1 = 352 ; // Markierung setzen, Data = PPoint - X, Y Pos
ecSetMarker2 = 353 ; // Markierung setzen, Daten = PPoint - X, Y Pos
ecSetMarker3 = 354 ; // Markierung setzen, Data = PPoint - X, Y Pos
ecSetMarker4 = 355 ; // Markierung setzen, Data = PPoint - X, Y Pos
ecSetMarker5 = 356 ; // Markierung setzen, Data = PPoint - X, Y Pos
ecSetMarker6 = 357 ; // Markierung setzen, Data = PPoint - X, Y Pos
ecSetMarker7 = 358 ; // Markierung setzen, Daten = PPoint - X, Y Pos
ecSetMarker8 = 359 ; // Markierung setzen, Data = PPoint - X, Y Pos
ecSetMarker9 = 360 ; // Markierung setzen, Data = PPoint - X, Y Pos
ecToggleMarker0 = 361 ; // Wenn Marker in der Lücke ist, Marker entfernen, Marker
                        // setzen lesen, Data = PPoint - X, Y Pos
ecToggleMarker1 = 362 ;
ecToggleMarker2 = 363 ;
ecToggleMarker3 = 364 ;
ecToggleMarker4 = 365 ;
ecToggleMarker5 = 366 ;
ecToggleMarker6 = 367 ;
ecToggleMarker7 = 368 ;
ecToggleMarker8 = 369 ;
ecToggleMarker9 = 370 ;

EcFoldLevel1 = 371 ; // Falten aller Falten, größer/gleich Schachtelungsebene 1
EcFoldLevel2 = EcFoldLevel1 + 1 ;
EcFoldLevel3 = EcFoldLevel2 + 1 ;
EcFoldLevel4 = EcFoldLevel3 + 1 ;
EcFoldLevel5 = EcFoldLevel4 + 1 ;
EcFoldLevel6 = EcFoldLevel5 + 1 ;
EcFoldLevel7 = EcFoldLevel6 + 1 ;
EcFoldLevel8 = EcFoldLevel7 + 1 ;
EcFoldLevel9 = EcFoldLevel8 + 1 ;
EcFoldLevel10 = EcFoldLevel9 + 1 ;
EcFoldCurrent = 381 ;
EcUnFoldCurrent = 382 ;
EcToggleMarkupWord = 383 ;
```

### 1.5.2 Zugriff auf Lines[]

Der Inhalt des Editors (die Textzeilen) wird in der Eigenschaft "Lines" gespeichert, bei der es sich um eine Liste von Strings handelt, die einem "TStringList"-Objekt ähnelt, so dass auf sie wie auf eine gewöhnliche Liste von Strings zugegriffen werden kann. Um z.B. den Inhalt der ersten Zeile anzuzeigen, würden wir folgendes tun:

```
showmessage ( SynEdit1.Lines[0] );
```

Der Zugriff auf Lines[] ist ein schneller Weg, um auf den Inhalt von SynEdit zuzugreifen. Da es sich um eine Liste von Strings handelt, können wir alle String-Funktionen für unsere Zwecke nutzen. Allerdings können die vorgenommenen Änderungen nicht mit der "Undo"-Methode des Editors rückgängig gemacht werden.

Es ist daher nicht ratsam, Lines[] zu ändern, wenn Sie beabsichtigen, die Änderungen später rückgängig zu machen.

Da Lines[] den gesamten Text von SynEdit enthält, müssen wir, wenn wir auf die erste Zeile von SynEdit zugreifen wollen, auf Lines[0] zugreifen. Um also einen Text in die erste Zeile von SynEdit zu schreiben, müssen wir Folgendes tun:

```
SynEdit1.Lines[0] := 'Hallo Welt' ;
```

Diese Anweisung wird immer funktionieren, da SynEdit mindestens eine Textzeile enthält, aber wenn wir versuchen, auf Lines[1] zuzugreifen, ohne dass eine zweite Zeile im Editor vorhanden ist, wird zur Laufzeit ein Fehler erzeugt.

Um die Anzahl der Zeilen im Editor zu ermitteln, können wir die Methode Count verwenden:

```
Zeilennummern := SynEdit1.Lines.Count;
```

Da es sich bei Lines[] um eine Liste handelt, verfügt sie über viele der uns bekannten Listenmethoden. Um zum Beispiel eine weitere Zeile zum Editor hinzuzufügen, können wir dies tun:

```
SynEdit1.Lines.Add ( 'Neue Textzeile' );
```

Einige der Eigenschaften von Lines[] sind in der folgenden Tabelle aufgeführt:

| EIGENSCHAFT  | BESCHREIBUNG  |
|--------------|---|
| Add          | Hinzufügen einer Textzeile  |
| AddStrings   | Fügt alle Inhalte einer anderen Liste hinzu.  |
| Capacity     | Bestimmt die Anzahl der Zeilen, die erstellt werden, wenn die Liste erweitert werden muss. Dies spart Zeit, da mehrere Elemente auf einmal erstellt werden. |
| Clear        | Löschen Sie den Inhalt der Liste.   |
| Count        | Anzahl der Line[]-Elemente (Editorzeilen)   |
| Delete       | Ein Element aus der Liste löschen.  |
| Exchange     | Tauschen Sie die Positionen zweier Zeilen.  |
| Insert       | Fügt eine Linie an einer bestimmten Position ein.   |
| LoadFromFile | Ermöglicht es Ihnen, den Inhalt einer Datei zu lesen  |
| SaveToFile   | Ermöglicht es Ihnen, den Inhalt in einer Datei zu speichern   |
| Text         | Gibt den vollständigen Inhalt aller Zeilen im Editor zurück   |

Um über den gesamten Inhalt von Lines[] zu iterieren, kann die folgende Konstruktion verwendet werden:

```
for i := 0 to SynEdit1.Lines.Count - 1 do  
  ShowMessage ( SynEdit1.Lines[i] );
```

Um Änderungen vorzunehmen, ist es besser, sie mit Editorbefehlen vorzunehmen, als direkt auf "Lines[]" zuzugreifen, damit die "Undo"-Optionen aktiv bleiben. Komplexe Befehlsänderungen können jedoch sehr viel langsamer sein als Änderungen in "Lines[]".

### 1.5.3 Die Zwischenablage

Mit diesen Methoden werden typische Funktionen der Zwischenablage aktiviert:

```
SynEdit1.CopyToClipboard ;  
SynEdit1.CutToClipboard ;  
SynEdit1.PasteFromClipboard ;
```

Es ist nicht notwendig, zu erklären, was sie tun, da ihre Namen recht gut bekannt sind und wir bereits wissen, dass sie Daten zwischen dem markierten Text und der Zwischenablage verschieben. Das Verhalten ist dasselbe, als ob wir die Tastenkombinationen Strg+C, Strg+V und Strg+X ausführen würden. Tatsächlich ist dieselbe Tastenkombination standardmäßig aktiv, wenn ein SynEdit verwendet wird.

Es ist zu beachten, dass diese Optionen auch im Spaltenauswahlmodus funktionieren, so dass rechteckige Textblöcke kopiert und eingefügt werden können.

Auf die Optionen der Zwischenablage kann auch mit Hilfe von Befehlen zugegriffen werden:

```
SynEdit1 . CommandProcessor ( ecCopy , ' ' , nil ) ;  
SynEdit1 . CommandProcessor ( ecPaste , ' ' , nil ) ;  
SynEdit1 . CommandProcessor ( ecCut , ' ' , nil ) ;
```

Da die Zwischenablage mit der Auswahl arbeitet, ist es üblich, mit den Eigenschaften "BlockBegin" und "BlockEnd" zu arbeiten.

Es ist auch möglich, Text direkt in die Zwischenablage zu legen, ohne eine "Kopie" machen zu müssen:

```
SynEdit1.DoCopyToClipboard ('abzuschneidender Text' , '' ); // in die  
Zwischenablage stellen
```

### 1.5.4 Wiederherstellen und Rückgängig machen.

SynEdit bietet eine sehr gute Kontrolle über die am Text vorgenommenen Änderungen. In fast allen Fällen können Sie Änderungen rückgängig machen und wiederherstellen. Diese Änderungen können sogar im Spaltenmodus geändert werden.

Jedes Mal, wenn eine Änderung vorgenommen wird, speichert SynEdit die Änderung im internen Speicher.

Zur Kontrolle von Änderungen werden die Methoden verwendet:

| METHODE/<br>EIGENSCHAFT                                     | BESCHREIBUNG  |
|---|---|
| SynEdit1.Undo()   | Macht eine auf dem Bildschirm vorgenommene Änderung rückgängig  |
| SynEdit1.Redo()   | Eine bereits rückgängig gemachte Änderung wiederherstellen.   |
| SynEdit1.ClearUndo()  | Löscht die Liste der Änderungen (Rückgängig machen) und erlaubt kein Rückgängigmachen ab diesem Punkt.    |
| SynEdit1.MaxUndo  | Maximale Anzahl von Aktionen (Rückgängig), die aufgezeichnet werden und rückgängig gemacht werden können. |
| SynEdit1.BeginUndoBlock()<br>SynEdit1.EndUndoBlock()<br>( ) | Sie erzeugen einzigartige Änderungsblöcke.  |
| SynEdit1.CanUndo  | Zeigt an, ob es Aktionen gibt, die rückgängig gemacht werden können                                       |
| SynEdit1.CanRedo  | Zeigt an, ob es Aktionen gibt, die erneut durchgeführt werden müssen                                      |

Fast alle Änderungen, die manuell über die Tastatur in SynEdit vorgenommen werden, können "rückgängig" gemacht werden. Wenn Sie jedoch Inhalte über den Code ändern, sollten Sie bedenken, dass einige Aktionen nicht rückgängig gemacht werden können.

Die folgende Tabelle zeigt die Änderungsmethoden, die in einem SynEdit vorgenommen werden und ob sie "Rückgängig" unterstützen.

| <b>AKTION</b>  | <b>ERLAUBT<br/>RÜCKGÄNGIGMA-<br/>CHEN</b> |
|--|---|
| Methoden:<br>SynEdit1.ClearAll;<br>SynEdit1.ExecuteCommand()<br>SynEdit1.InsertTextAtCaret()<br>SynEdit1.TextBetweenPoints()<br>SynEdit1.SearchReplace()<br>SynEdit1.SearchReplaceEx() | Ja  |
| Typänderungen:<br>SynEdit1.Text:='Hallo';<br>SynEdit1.LineText:='Hallo';   | Nein                                      |
| Direkte Änderungen an SynEdit1.Lines[]   | Nein                                      |
| Änderungen über die Zwischenablage:<br>SynEdit1.CopyToClipboard;<br>SynEdit1.CutToClipboard;<br>SynEdit1.PasteFromClipboard;   | Ja  |

Wir müssen uns überlegen, welche Methoden wir verwenden müssen, um Änderungen in einem SynEdit vorzunehmen, wenn wir die "Rückgängig"-Möglichkeiten beibehalten wollen. Normalerweise können die Änderungen, die sie erzeugen, jede Anweisung, die den Inhalt von SynEdit ändert und die "Undo" unterstützt, mit einem einfachen Aufruf von "Undo" verworfen werden.

Wenn Sie mehrere rückgängig zu machende Aktionen mit einem einzigen "Undo" zusammenfassen wollen, müssen Sie die Methoden BeginUndoBlock und EndUndoBlock verwenden:

```
SynEdit1.BeginUndoBlock ;    // Hier kann es mehrere Änderungen geben, die das  
Rückgängigmachen unterstützen.  
...  
SynEdit1.EndUndoBlock ;
```

Mit dieser Konstruktion werden wir in der Lage sein, alle Änderungen mit einem einzigen Aufruf von "Undo" rückgängig zu machen.



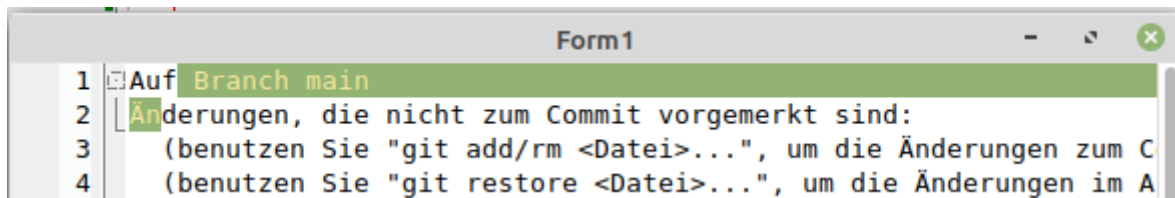
Es kann sinnvoll sein, die Konstrukte BeginUpdate und EndUpdate zu verwenden, um zu verhindern, dass das Steuerelement aktualisiert wird, bis alle Änderungen an SynEdit vorgenommen wurden:

```
ActiveEditor.SynEditor.BeginUpdate ; // SynEdit-Aktualisierung ist deaktiviert
try
    SynEdit1.BeginUndoBlock ;
    // Möglicherweise gibt es hier mehrere Änderungen, die das Rückgängigmachen
    // unterstützen.
    ...
    SynEdit1.EndUndoBlock ;
finally
    ActiveEditor.SynEditor.EndUpdate ; // SynEdit-Aktualisierung ist wieder
                                       // aktiviert
end ;
```

Auf diese Weise können wir die Geschwindigkeit von Änderungen verbessern, da das Steuerelement nicht jedes Mal aktualisiert werden muss, wenn etwas geändert wird. Wenn die SynEdit-Aktualisierung deaktiviert ist, werden auch Anfragen vom Typ Application.ProcessMessages() ignoriert.

## 1.6 Verwaltung der Auswahl

Wie die meisten heutigen Editoren kennt auch SynEdit nur einen Auswahlblock<sup>5</sup>. Dieser Block wird für das Ausschneiden und Kopieren von Text verwendet, kann aber auch zum Ändern von Text durch Überschreiben verwendet werden:



Die Auswahl wird im Code durch die Eigenschaften "BlockBegin" und "BlockEnd" definiert, die vom Typ "TPoint" sind. Im vorherigen Beispiel sind die Werte, die "BlockBegin" und "BlockEnd" zugewiesen wurden, folgende (beachte Umlaut!):

```
SynEdit1.BlockBegin := Point(4,1);  
SynEdit1.BlockEnd   := Point(4,2);
```

Das lässt sich nicht zuweisen, nur auslesen!

```
SynEdit1.BlockBegin.x := 4;  
SynEdit1.BlockBegin.y := 1;  
SynEdit1.BlockEnd.x   := 4;  
SynEdit1.BlockEnd.y   := 2;  
zum Beispiel:  
i := SynEdit1.BlockBegin.x
```

Wenn es keine aktive Auswahl gibt, zeigen "BlockBegin" und "BlockEnd" die gleiche Position an.

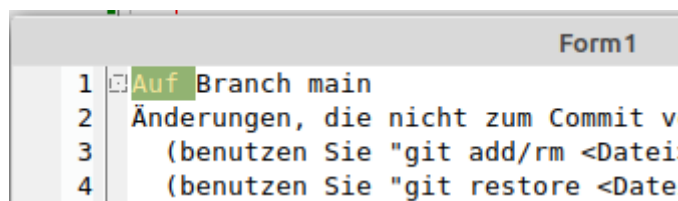
---

<sup>5</sup> Es gibt Editoren, die mehr als einen Auswahlblock verarbeiten können, aber die meisten arbeiten nur mit einem einzigen Auswahlblock.

Eine andere Möglichkeit, den Auswahlblock zu definieren, ist die Verwendung der Eigenschaften "SelStart" und "SelEnd". Mit diesen Eigenschaften können Sie auf die gleiche Weise arbeiten wie mit "BlockBegin" und "BlockEnd", aber sie sind nicht vom Typ "TPoint", sondern einfache Ganzzahlen, die den Text so abbilden, als ob es sich um eine ununterbrochene Reihe von Zeichen handelt.

Zeilenumbrüche gelten in Windows und Linux als zwei Zeichen und können ebenfalls Teil der Auswahl sein.

Das folgende Beispiel zeigt eine Auswahl, die mit SelStart = 1 und SelEnd = 5 definiert wurde:



```
SynEdit1.SelStart:= 1;
SynEdit1.SelEnd  := 5;
```

Der Auswahlblock wird von der Position des Zeichens, auf das "SelStart" verweist, bis zu dem Zeichen vor dem Zeichen, auf das "SelEnd" verweist, definiert. Daher ist die Anzahl der ausgewählten Zeichen gleich (SelEnd-SelStart).

Wenn Sie einen Bereich relativ zur aktuellen Cursorposition auswählen möchten, können Sie diese Methoden verwenden:

|                           |   |
|---------------------------|---|
| SynEdit1.SelectWord;      | Wählt das aktuelle Wort an der Stelle aus, an der sich der Cursor befindet.                 |
| SynEdit1.SelectLine;      | Wählen Sie die aktuelle Zeile aus, in der sich der Cursor befindet.                         |
| SynEdit1.SelectParagraph; | Wählen Sie den aktuellen Absatz an der Stelle aus, an der sich der Cursor befindet.         |
| SynEdit1. SelectToBrace;  | Wählt den durch Klammern, geschweifte Klammern oder eckige Klammern abgegrenzten Block aus. |

Alle diese Auswahlmethoden basieren auf der aktuellen Position des Cursors und funktionieren so, als ob die Auswahl manuell erfolgen würde, da SelectWord ein Wort nur anhand der alphabetischen Zeichen identifiziert, einschließlich akzentuierter Zeichen, **Umlauten** und dem Buchstaben ñ

"SelectLine", hat die folgende Erklärung:

```
procedure SelectLine ( WithLeadSpaces : Boolean = True );
```

Mit dem optionalen Parameter können Sie angeben, ob Sie führende und abschließende Leerzeichen in die Auswahl einbeziehen möchten. Wenn er auf FALSE gesetzt ist, ist die Auswahl der aktuellen Zeile möglicherweise nicht vollständig, wenn sich in der Zeile führende oder nachgestellte Leerzeichen befinden.

Mit der Methode SelectToBrace können Sie Textblöcke auswählen, die durch Klammern, geschweifte Klammern oder eckige Klammern abgegrenzt sind. Sie funktioniert nur, wenn die folgenden Bedingungen erfüllt sind:

- Das aktuelle Zeichen, an dem sich der Cursor befindet, sei '(', '{' oder '[' oder das Zeichen vor dem Cursor sei ')', '}' oder ']'.  
• dass das entsprechende Begrenzungszeichen in derselben oder einer anderen Zeile vorhanden ist.

Außerdem ist zu beachten, dass die Auswahl mit SelectToBrace die Verschachtelung von Blöcken desselben Typs erlaubt.

Um festzustellen, ob eine aktive Auswahl (markierter Text) vorhanden ist, muss die Eigenschaft "SelAvail" verwendet werden:

```
if editor.SelAvail then ...
```

Eine andere Möglichkeit wäre, die Koordinaten von "BlockBegin" und "BlockEnd" zu vergleichen.

Der ausgewählte Text kann über die Eigenschaft "SelText" abgerufen werden. Es wird nur ein Auswahlblock unterstützt. Normalerweise befindet sich der Cursor an einer der Grenzen des Auswahlblocks, aber im Modus "persistenter Block" kann der Cursor unabhängig von der Position des Auswahlblocks gemacht werden.

Die Eigenschaft "SelText" ist auch beschreibbar, so dass wir den ausgewählten Text ändern können. Mit dem folgenden Code wird der ausgewählte Text entfernt:

```
SynEdit1.SelText := '' ;
```

Um die Auswahl zu löschen, gibt es jedoch die Methode "ClearSelection", die eine verkürzte Form darstellt.

Um den gesamten Text zu markieren, müssen wir verwenden:

```
SynEdit1.SelectAll ;
```

Um den ausgewählten Text zu entfernen, können Sie die Methode "ClearSelection" verwenden:

```
SynEdit1.ClearSelection ;
```

Sie können auch den Befehl " ecDeleteLastChar " mit ExecuteCommand() senden:

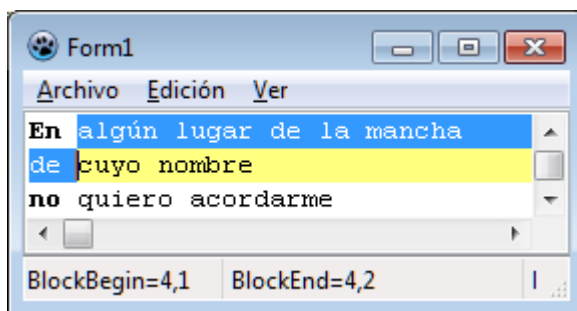
```
SynEdit1.ExecuteCommand ( ecDeleteLastChar, ' ', nil );
```

Eine andere Möglichkeit wäre, den ausgewählten Text durch TextBetweenPoints() zu ersetzen:

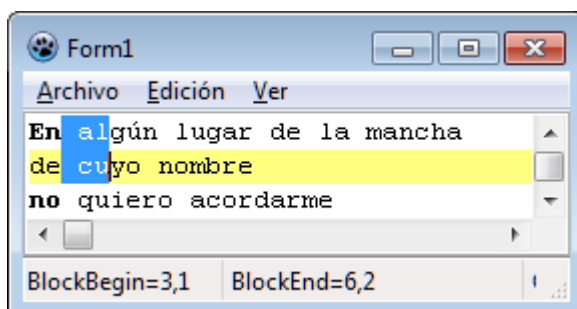
```
SynEdit1.TextBetweenPoints[editor.BlockBegin,editor.BlockEnd] := ' ' ;
```

### 1.6.1 Auswahl im Spaltenmodus

Die Auswahl von Text in SynEdit kann auf verschiedene Weise erfolgen. Normalerweise werden alle Zeilen zwischen dem Anfang und dem Ende des Auswahlblocks ausgewählt:



Sie können aber auch den Spaltenauswahlmodus verwenden:



In diesem Modus bildet der Auswahlbereich ein Rechteck und schließt die Linien in seinem Verlauf nur teilweise ein.

Um im Spaltenmodus in SynEdit auszuwählen, müssen Sie die Tastenkombination <Alt>+<Umschalt>+Richtungstaste verwenden.

Sobald Sie den Text ausgewählt haben, können Sie die Aktionen Ausschneiden, Kopieren oder Einfügen ausführen. Sie können die Auswahl auch durch Drücken einer beliebigen Taste aufheben.

Es ist zu beachten, dass jedes Drücken einer anderen Taste als der Kombination <Alt>+<Umschalt>+Richtung dazu führt, dass der Auswahlmodus für den Spaltenmodus beendet wird.

Um pro Programm in den Spaltenmodus zu wechseln, kann der folgende Code verwendet werden:

```
uses ... , SynEditTypes ;

var pos : TPoint ;
...
pos.x := 3 ;
pos.y := 2 ;
SynEdit1.BlockBegin := pos ; // Anfangsauswahlpunkt festlegen
pos.x := 8 ;
pos.y := 3 ;
SynEdit1.BlockEnd := pos ; // Endpunkt der Auswahl festlegen
SynEdit1.CaretXY := pos ;
SynEdit1.SelectionMode := smColumn ; // Umschalten auf Spaltenmodus
...
```

Ebenso führt in diesem Fall jedes Drücken einer anderen Taste als der Kombination <Alt>+<Umschalt>+Richtung dazu, dass der Auswahlmodus für den Spaltenmodus beendet wird.

Es gibt noch andere Formen der Auswahl, die in der Unit "SynEditTypes" definiert sind:

- smNormal,
- smColumn
- smLine,
- **smCurrent**

Der Modus smNormal ist der standardmäßig aktive Modus und ist der normale Auswahlmodus.

Der Modus smColumn ist der Modus der Spaltenauswahl.

Der smLine-Modus ist ein Auswahlmodus, der bewirkt, dass alle Zeilen zwischen BlockBegin und BlockEnd als Teil der Auswahl markiert werden.

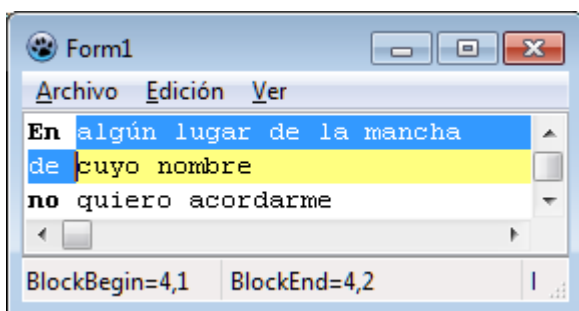
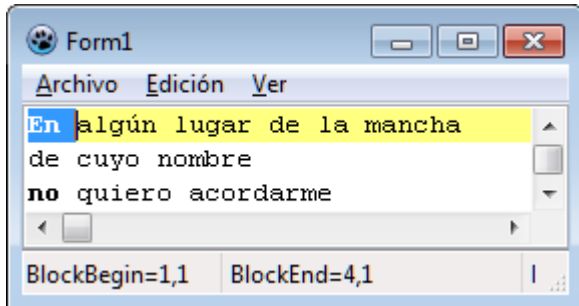
Sie können auch mit Hilfe von Tastaturbefehlen nach Spalten auswählen:

```
SynEdit1.ExecuteCommand ( ecColSelUp, #0, nil )
```

Die Konstanten ecColSelUp, ecColSelDown, ecColSelLeft, ecColSelRight und andere ermöglichen es Ihnen, den Cursor zu bewegen, während Sie den Spaltenmodus beibehalten, was dem Gedrückthalten der Tasten <Alt>+<Shift> entspricht.

## 1.6.2 BlockBegin und BlockEnd

Sie bestimmen die Koordinaten des Auswahlblocks.



Auswahlblöcke können mit BlockBegin und BlockEnd definiert werden. Zum Beispiel können Sie eine Selektion mit diesem Code erzeugen:

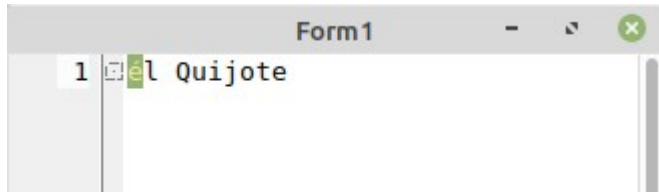
```
var p : TPoint ;  
...  
SynEdit1.Text := 'Don Quijote' ;  
// Blockanfang setzen  
p := SynEdit1.BlockBegin ;  
p.x := 1 ;  
SynEdit1.BlockBegin := p ;  
// Ende des Blocks setzen  
p := SynEdit1.BlockEnd ;  
p.x := 3 ;  
SynEdit1.BlockEnd := p ;
```

Der Code erzeugt die folgende Ausgabe auf dem Bildschirm:



Wenn SynEdit jedoch in UTF-8 arbeitet (wie es normalerweise der Fall ist), kann ein Sonderzeichen zwei Zeichen breit sein. Daher muss auf die logischen Koordinaten (die von BlockBegin und BlockEnd behandelt werden) und die physischen Koordinaten geachtet werden.

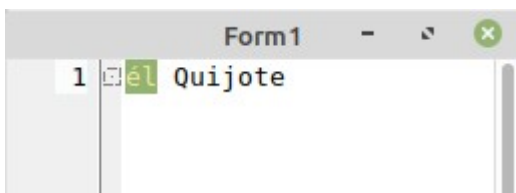
Wäre die Zeichenfolge im Editor "él Quijote" gewesen, hätte die Auswahl ein anderes Ergebnis:



Um dieses Verhalten zu umgehen, müssen Sie die Funktion PhysicalToLogicalCol() verwenden:

```
var p : TPoint ;  
...  
SynEdit1.Text := 'él Quijote' ;  
// Blockanfang setzen  
p := SynEdit1.BlockBegin ;  
p.x := 1 ;  
SynEdit1.BlockBegin := p ;  
// Ende des Blocks setzen  
p := SynEdit1.BlockEnd ;  
p.x := SynEdit1.PhysicalToLogicalCol(SynEdit1.Lines[0], 0, 3) ;  
SynEdit1.BlockEnd := p ;
```

Jetzt ist das Verhalten wie erwartet:



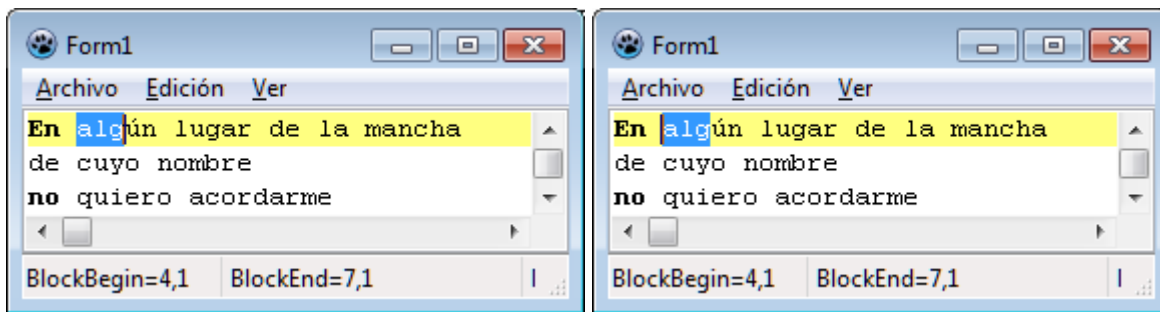


### 1.6.3 BlockBegin und BlockEnd in der normalen Auswahl

(smNormal)

Wenn die Auswahl nur eine Zeile hat, zeigt der BlockBegin-Punkt immer in die Spalte links von der Auswahl, unabhängig davon, aus welcher Richtung die Auswahl erfolgt ist.

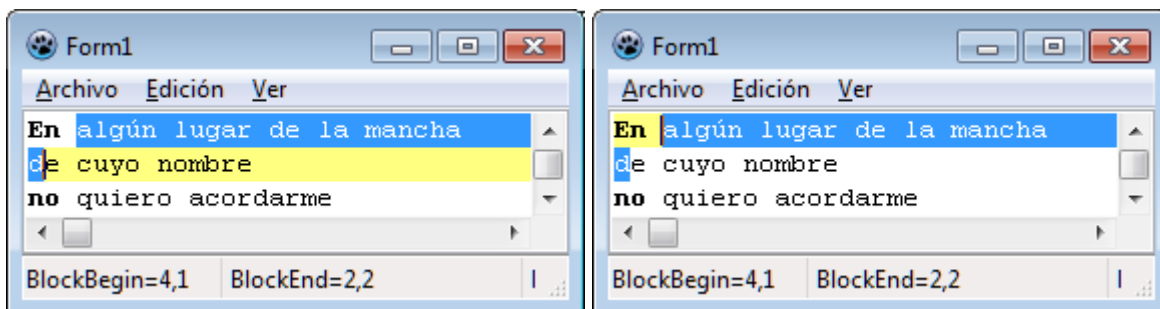
In den folgenden Beispielen wurde eine Auswahl von links nach rechts bzw. von rechts nach links getroffen.



Wie es aussieht, gibt es keinen Unterschied im Sinne der Auswahl.

Wenn die Auswahl mehrere Zeilen umfasst, erscheint der BlockBegin-Punkt immer in der obersten Zeile, unabhängig davon, aus welcher Richtung die Auswahl erfolgt ist.

In den folgenden Beispielen wurde eine Auswahl von links nach rechts bzw. von rechts nach links getroffen.

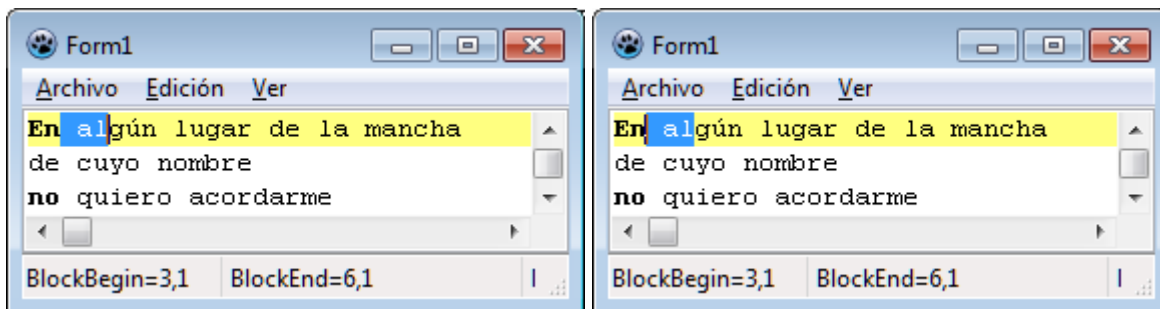


## 1.6.4 BlockBegin und BlockEnd in der Auswahl im Spaltenmodus

(smColumn)

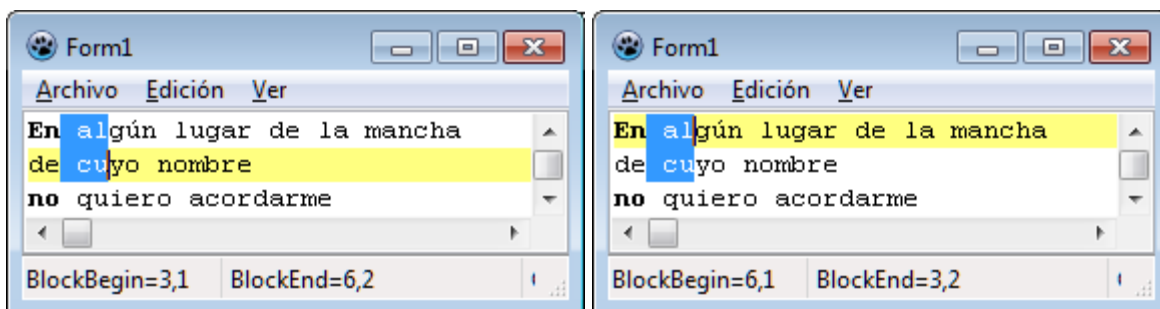
Wenn die Auswahl nur eine Zeile hat, zeigt der BlockBegin-Punkt immer in die Spalte links von der Auswahl, unabhängig davon, aus welcher Richtung die Auswahl erfolgt ist.

In den folgenden Beispielen wurde eine Auswahl von links nach rechts bzw. von rechts nach links getroffen.



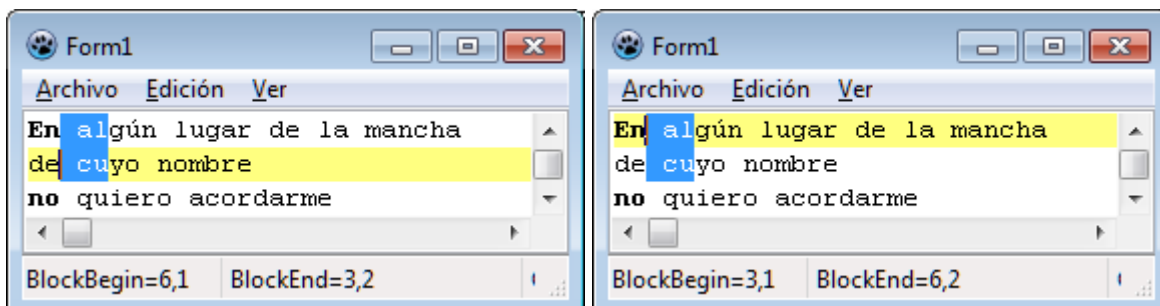
Wenn die Auswahl mehrere Zeilen umfasst, erscheint der BlockBegin-Punkt immer in der obersten Zeile, unabhängig davon, aus welcher Richtung die Auswahl erfolgt ist.

In den folgenden Beispielen wurde eine Auswahl von links nach rechts, unten und oben getroffen:



Beachten Sie, dass im letzten Fall BlockBegin die Position des Cursors übernimmt.

In den folgenden Beispielen wurde eine Auswahl von rechts nach links, unten und oben getroffen:



Die Positionen von BlockBegin und BlockEnd sind umgekehrt wie im vorherigen Fall.

In allen betrachteten Fällen nimmt der Cursor immer den Wert von BlockBegin oder BlockEnd an.

## 1.7 Suchen und Ersetzen

Fast jeder Texteditor verfügt über Optionen zum Suchen und Ersetzen. Es ist also nützlich zu wissen, wie man die Such- und Ersetzungsoptionen in SynEdit implementiert.

Nachdem wir einige seiner Eigenschaften und Methoden kennen, können wir die Funktionen Suchen und Ersetzen selbst implementieren. Für die Suche können wir den Inhalt von SynEdit zeilenweise durchsuchen (mit Zugriff auf die Liste Lines[]) und den gesuchten Text mit den Eigenschaften BlockBegin und BlockEnd auswählen.

SynEdit verfügt jedoch bereits über zwei Funktionen zum Suchen und Ersetzen:

```
function SearchReplace ( const ASearch, AReplace : string ;  
                        AOptions : TSynSearchOptions ) : integer ;  
  
function SearchReplaceEx ( const ASearch, AReplace : string ;  
                          AOptions : TSynSearchOptions ; AStart : TPoint ) :  
                          integer ;
```

SearchReplaceEx() ist ähnlich wie SearchReplace(), mit dem Unterschied, dass SearchReplaceEx() ab der angegebenen Position (AStart) sucht.

Diese Funktionen ermöglichen die Suche nach oben oder unten, nach Groß-/Kleinschreibung, nach ganzen Wörtern oder mit regulären Ausdrücken.

Der von diesen Funktionen zurückgegebene Wert ist eine ganze Zahl:

| MODUS    | WERT | BEDEUTUNG                                 |
|----------|------|---|
| Suche    | 0    | Artikel nicht gefunden                    |
| Suche    | 1    | Wenigstens ein Gegenstand wurde gefunden. |
| Ersetzen | 0    | Artikel nicht gefunden                    |
| Ersetzen | n    | "n" Elemente wurden ersetzt               |

Im Suchmodus halten diese Funktionen an, wenn sie den ersten Treffer gefunden haben, markieren ihn und machen den markierten Text im Editor sichtbar.

Der Parameter AOptions ist ein Array, das die folgenden Elemente enthalten kann:

```
TSynSearchOption =  
( ssoMatchCase,  
  ssoWholeWord,  
  ssoBackwards,  
  ssoEntireScope,  
  ssoSelectedOnly,  
  ssoReplace,  
  ssoReplaceAll,  
  ssoPrompt,  
  ssoSearchInReplacement, //Suche fortsetzen - Ersetzen durch Ersetzen  
                           // (mit ssoReplaceAll ) rekursiv ersetzen  
  
  ssoRegExpr,  
  ssoRegExprMultiLine,  
  ssoFindContinue //Angenommen, die aktuelle Auswahl ist die letzte  
                  //Übereinstimmung,  
                  //und Suche hinter der Auswahl starten ( vor  
                  //wenn ssoBackward )  
                  //Standardmäßig wird am Caret begonnen ( Only  
                  //SearchReplace /  
                  //SearchReplaceEx hat Start-/End-Parameter )  
);
```

Alle diese Konstanten sind in der Unit "SynEditTypes" definiert.

Wenn die Elemente ssoReplace oder ssoReplaceAll enthalten sind, wird eine Ersetzung vorgenommen, andernfalls wird nur eine Suche durchgeführt.

### 1.7.1 Suchen

Für die Suche nach einer einfachen Zeichenkette kann der folgende Code verwendet werden:

```
var
  encon      : integer ;
  searched   : string ;
begin
  searched := 'text to search' ;
  encon := SynEdit1.SearchReplace(searched, ' ' ,[] );
  if encon = 0 then
    ShowMessage ( 'Not found : ' + searched );
  ...
```

Wenn keine Suchoptionen angegeben sind, werden die folgenden Optionen angenommen;

- Die Suche in SynEdit beginnt immer an der Cursorposition.
- Die Suchrichtung ist vorwärts vom Cursor bis zum Ende der Datei.
- Groß- und Kleinschreibung wird nicht unterschieden, Groß- und Kleinbuchstaben werden gleichermaßen erkannt.

Der Aufruf von SearchReplace() erzeugt den folgenden Ablauf:

- 1 Starten Sie die Suche mit dem gewünschten Text und den angegebenen Optionen.
- 2 Wenn die Suche erfolglos war, wird Null zurückgegeben und die Funktion beendet.
- 3 Wenn die Suche erfolgreich war, wird der erste Treffer ausgewählt und der ausgewählte Text im Editor sichtbar gemacht.

Die Auswahl des gefundenen Textes bewirkt, dass der Cursor an das Ende des ausgewählten Textes wandert. Beim nächsten Aufruf von SearchReplace() wird also ab dieser Position gesucht (find next).

Um eine Rückwärtssuche durchzuführen:

```
encon := SynEdit1.SearchReplace (searched, ' ' , [ssoBackwards] );
```

Um eine Suche unter Berücksichtigung des Feldes (Groß/Klein) durchzuführen:

```
encon := SynEdit1.SearchReplace (searched, ' ' , [ssoMatchCase] );
```

Die Suchoptionen können kombiniert werden.

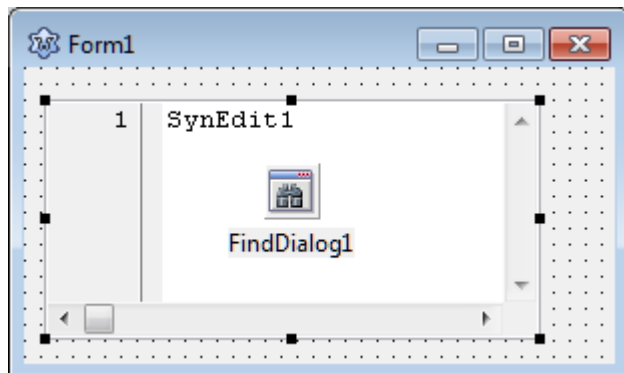
Wenn die Option "ssoEntireScope" angegeben ist, beginnt die Suche immer am Anfang des Textes (unabhängig davon, wo sich der Cursor befindet) und endet, wenn das erste Element gefunden wurde.

Wenn die Option "ssoSelectedOnly" angegeben ist, wird die Suche innerhalb des markierten Textes durchgeführt, und die Suche wird beendet, wenn das erste Element gefunden wird. Wenn Sie versuchen, eine weitere Suche durchzuführen, erhalten Sie immer das gleiche Ergebnis, da der ausgewählte Text nach dem Aufruf von SearchReplace() immer den gesuchten Text enthält.

## 1.7.2 Suche mit TFindDialog

Es gibt unter den Lazarus Steuerelementen einen Dialog, der speziell für Suchoperationen erstellt wurde. Diese Komponente ist TFindDialog, und sie befindet sich in der Komponentenpalette, in der Registerkarte "Dialoge".

Um sie zu verwenden, müssen wir diese Komponente in unserem Formular platzieren:



Und dann müssen wir eine Prozedur erstellen, die das Ereignis "OnFind" behandelt. Der Ablauf eines Ereignisses kann folgendermaßen aussehen:

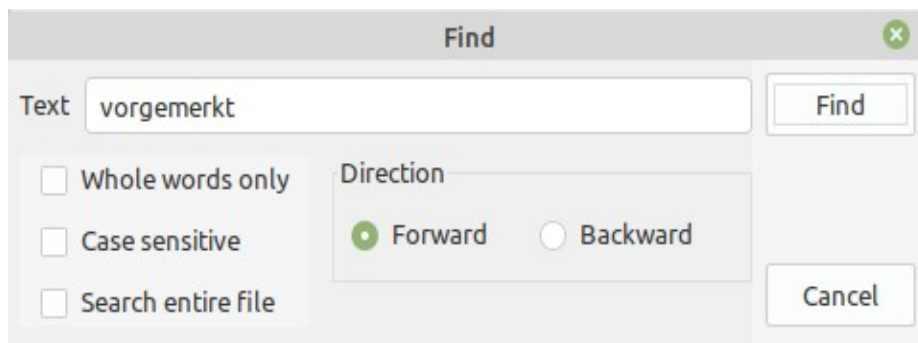
```
procedure TForm1.FindDialog1Find(Sender: TObject);
var
  encon      : integer;
  searched   : string;
  options    : TSynSearchOptions;
begin
  searched := FindDialog1.FindText;
  options := [];
  if not(frDown in FindDialog1.Options) then options += [ssoBackwards];
  if frMatchCase in FindDialog1.Options then options += [ssoMatchCase];
  if frWholeWord in FindDialog1.Options then options += [ssoWholeWord];
  if frEntireScope in FindDialog1.Options then options += [ssoEntireScope];
  encon := Synedit1.SearchReplace(searched, '', options);
end;
```

Das Steuerelement "FindDialog1" gibt die ausgewählten Optionen über seine Eigenschaft "Optionen" preis.

Die Idee ist, die im Dialog ausgewählten Optionen an die Variable "options" zu übergeben, bevor SearchReplace() aufgerufen wird.

Nun müssen wir in einem strategischen Teil unseres Programms (z. B. in der Antwort auf das Menü, **oder einen Button klick**) den Code zum Öffnen des Dialogs und zum Starten der Suche einfügen:

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    FindDialog1.Execute ; // öffnet den Suchdialog  
end;
```



Wenn Sie auf "Suchen" klicken, wird das Ereignis "OnFind" aufgerufen, das mit der von uns erstellten Methode "FindDialog1Find" verbunden sein muss. Der Dialog bleibt jedoch sichtbar, bis Sie auf "Abbrechen" klicken oder die <escape>-Taste drücken.

Die Methode "FindDialog1Find" hat noch eine weitere Verwendung. Sie wird verwendet, um die Funktion "Weiter suchen" zu implementieren, da diese Methode auch dann aufgerufen werden kann, wenn der Dialog "FindDialog1" ausgeblendet ist.

Wenn dieses Fenster nicht den gewünschten Suchanforderungen entspricht, können Sie jederzeit ein spezielles Formular für unsere personalisierte Suche erstellen.

### 1.7.3 Ersetzen

Der Ersetzungsprozess ist ähnlich wie der Suchprozess. Um nach einer einfachen Zeichenkette zu suchen, kann der folgende Code verwendet werden:

```
var
  encon    : integer;
  searched : string;
begin
  searched := FindDialog1.FindText;
  encon := Synedit1.SearchReplace(searched, 'neuer String', [ssoReplace]);
  if encon = 0 then
    ShowMessage('Nicht gefunden : ' + searched);
end;
```

Der Unterschied besteht darin, dass der SearchReplace()-Methode zusätzlich zur Angabe des zu verwendenden Ersetzungstextes die Option "ssoReplace" mitgeteilt werden muss.

Die Funktionsweise von SearchReplace() im Ersetzungsmodus ist ähnlich wie im Suchmodus:

- 1 Starten Sie die Suche mit dem gewünschten Text und den angegebenen Optionen.
- 2 Wenn die Suche erfolglos war, wird Null zurückgegeben und die Funktion beendet.
- 3 Wenn die Suche erfolgreich war, wird der erste Treffer ersetzt und der ersetzte Text wird im Editor sichtbar gemacht.
- 4 Der Cursor bleibt am Ende des ersetzten Textes stehen, bereit für eine weitere Suchsequenz.

Diese Arbeitsweise ist im Suchmodus nützlich, aber im Ersetzungsmodus kann sie seltsam sein, da keine Bestätigung zum Ersetzen des Textes verlangt wird und die Ersetzung sofort und ohne vorherige Auswahl erfolgt.



Um dieses Verhalten zu verbessern, kann vor der Ersetzung ein Bestätigungsfenster eingefügt werden, in dem auch der zu ersetzende Text zu sehen ist.

Es gibt keinen vordefinierten Dialog zur Erstellung eines Bestätigungsfensters. Wenn wir eines verwenden wollen, müssen wir es selbst erstellen.

Ein einfacher Dialog, der uns helfen könnte, wäre eine `MessageBox()` mit den Schaltflächen Yes-No-Cancel. Unser Ersatzverfahren könnte also wie folgt geschrieben werden:

```
var
  encon, r      : integer;
  searched      : string;
  options       : TSynSearchOptions;

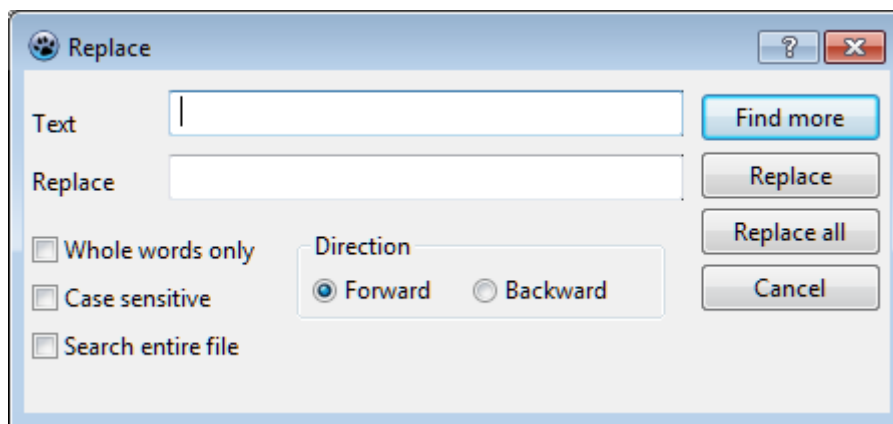
begin
  ...
  searched := 'gewünschte Zeichenfolge';
  options := []; //Suchoptionen
  encon := SynEdit1.SearchReplace(searched, '', options); //Suche
  while encon <> 0 do
    begin
      r:=Application.MessageBox('Dieses_Vorkommnis_ersetzen?', 'Ersetzen', MB_YESNOCANCEL);
      if r = IDCANCEL then exit;
      if r = IDYES then
        begin
          SynEdit1.TextBetweenPoints[SynEdit1.BlockBegin, SynEdit1.BlockEnd] := 'neuer String';
        end;
      encon := SynEdit1.SearchReplace(searched, '', options); //nächste Suche
    end;
  ShowMessage('Nicht gefunden : ' + searched);
end;
```

Die Idee dabei ist, vor jeder Ersetzung zu fragen und Ihnen die Möglichkeit zu geben, ein Ereignis zu überspringen oder den gesamten Vorgang abubrechen.

In dieser Arbeitsweise verwenden wir nicht die Option "ssoReplace", sondern wir verwenden `SearchReplace()`, nur im Suchmodus. Die Ersetzung im Editor erfolgt mit der Methode `"TextBetweenPoints()"`.

### 1.7.4 Ersetzen mit TReplaceDialog

So wie es den TFindDialog für die Suche gibt, ist es auch möglich, den "TReplaceDialog" aus der Komponentenpalette zu verwenden.



Dieses Dialogfeld erleichtert uns die Eingabe von Daten zum Starten eines Such-/Ersetzen-Vorgangs, aber es bietet uns keinen Bestätigungsdiallog vor dem Ersetzen.

Die Arbeit mit diesem Dialog ist einfach. Die Ereignisse "OnFind" und "OnReplace" müssen zugewiesen werden. Das erste Ereignis wird ausgeführt, wenn die Schaltfläche "Find more" gedrückt wird, und das zweite wird mit der Schaltfläche "Replace (Ersetzen)" oder der Schaltfläche "Replace all (Alle ersetzen)" ausgeführt.

Der folgende Code ist ein Beispiel für die Implementierung der Ersetzungsoption unter Verwendung des Dialogs "TReplaceDialog":

```
procedure TForm1.ReplaceDialog1Replace(Sender: TObject);
var
  encon, r      : integer;
  searched      : string;
  options       : TSynSearchOptions;
begin
  searched := ReplaceDialog1.FindText;
  options := [ssoFindContinue];
  if not(frDown in ReplaceDialog1.Options) then options += [ssoBackwards];
  if frMatchCase in ReplaceDialog1.Options then options += [ssoMatchCase];
  if frWholeWord in ReplaceDialog1.Options then options += [ssoWholeWord];
  if frEntireScope in ReplaceDialog1.Options then options += [ssoEntireScope];
  if frReplaceAll in ReplaceDialog1.Options then
  begin
    //alles ist zum Austausch bereit
    encon := Synedit1.SearchReplace(searched, ReplaceDialog1.ReplaceText,
      options+[ssoReplaceAll]); //Ersetzt
    ShowMessage('Es wurden ersetzt ' + IntToStr(encon) + ' Vorkommnisse.');
```

```
    exit;
  end;
  //Ersetzen mit Bestätigung
  ReplaceDialog1.CloseDialog;
  encon := Synedit1.SearchReplace(searched, '', options); //Suche
  while encon <> 0 do
  begin
    r:=Application.MessageBox('Dieses_Vorkommnis_ersetzen?', 'Ersetzen', MB_YESNOCANCEL);
    if r = IDCANCEL then exit;
    if r = IDYES then
    begin
      Synedit1.TextBetweenPoints[Synedit1 .BlockBegin, Synedit1 .BlockEnd]:=
        ReplaceDialog1.ReplaceText;
    end;
    encon := Synedit1.SearchReplace(searched, '', options); //nächste Suche
  end;
  ShowMessage('Nicht gefunden : ' + searched);
end;
```

In diesem Beispielen wird davon ausgegangen, dass der Dialog "ReplaceDialog1" in das Formular aufgenommen wurde.

Dieses Dialogfeld verfügt über verschiedene Optionen, die über die Eigenschaft "Optionen" angepasst werden können. Mit diesen Optionen können Sie bestimmte Schaltflächen ein- oder ausblenden.

Die Option "frPromptOnReplace" ermöglicht es Ihnen, die Kontrolle an ein Bestätigungsformular zu übergeben, genau wie wir es mit MessageBox() gemacht haben, aber diese Bestätigung wird durch das "OnReplaceText"-Ereignis des Texteditors (nicht des Dialogs) angefordert, der folgende Deklaration besitzt:

```
TReplaceTextEvent = procedure(Sender: TObject; const ASearch, AReplace: string;
  Line, Column: integer; var ReplaceAction:
  TSynReplaceAction) of object;
```

## 1.8 Hervorhebungsoptionen

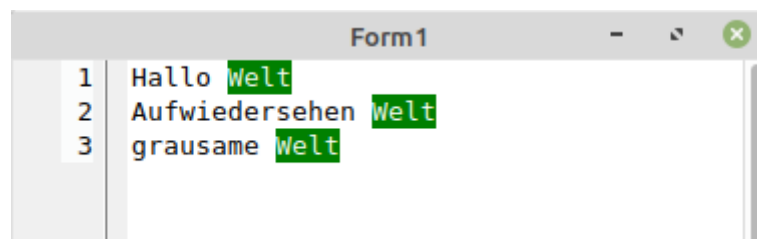
SynEdit ist eine ziemlich vollständige Komponente. Zu seinen verschiedenen Optionen gehört die Hervorhebung von Inhalten. Diese Optionen sind im Code des Editors selbst enthalten und sind unabhängig von der Verwendung von Syntax-Highlightern, die in Abschnitt 2 beschrieben werden.

### 1.8.1 Hervorheben eines Textes.

Um einen beliebigen Text hervorzuheben, können Sie die Methode SetHighlightSearch() wie folgt verwenden:

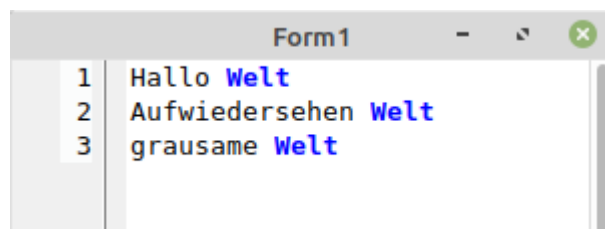
```
uses ..., SynEditTypes ;  
...  
SynEdit1.HighlightAllColor.Background := clGreen ;  
SynEdit1.SetHighlightSearch ( 'Welt' , [ssoSelectedOnly] ) ;
```

Wendet man diesen Code auf einen Text an, der die Zeichenfolge Welt enthält, erhält man ein Ergebnis wie das abgebildete:



```
uses ..., SynEditTypes ;  
...  
SynEdit1.HighlightAllColor.Foreground := clBlue;  
SynEdit1.HighlightAllColor.Background:= SynEdit1.Color;  
SynEdit1.HighlightAllColor.Style:= [fsBold];  
SynEdit1.SetHighlightSearch ( 'Welt' , [ssoSelectedOnly] ) ;
```

Wendet man diesen Code auf einen Text an, der die Zeichenfolge Welt enthält, erhält man ein Ergebnis wie das abgebildete:



Es ist zu beachten, dass der hervorzuhebende Text eine beliebige Kombination von Zeichen sein kann, nicht unbedingt Buchstaben.

Um festzulegen, dass nur ganze Wörter markiert werden, kann die Option "ssoWholeWord" verwendet werden:

```
SynEdit1.HighlightAllColor.Background := clGreen ;  
SynEdit1.SetHighlightSearch ( 'welt' , [ssoSelectedOnly, ssoWholeWord] );
```

Die Suchoptionen sind vom Typ TSynSearchOption.  
Diese sind:

```
TSynSearchOption =  
  (ssoMatchCase, ssoWholeWord,  
   ssoBackwards,  
   ssoEntireScope, ssoSelectedOnly,  
   ssoReplace, ssoReplaceAll,  
   ssoPrompt,  
   ssoSearchInReplacement,  
   ssoRegExpr, ssoRegExprMultiLine,  
   ssoFindContinue);
```

Wie Sie sehen, handelt es sich um dieselben Optionen, die auch für Suchen/Ersetzen verwendet werden. Nur einige dieser Optionen funktionieren mit SetHighlightSearch().

Mit den Optionen "ssoRegExpr" und "ssoRegExprMultiLine" können Sie Suchen mit regulären Ausdrücken wie "ab" durchführen. Allerdings müssen Sie bei der Auswahl der Ausdrücke vorsichtig sein, denn ein Ausdruck vom Typ "a\*" passt auf jedes beliebige Zeichen (auch auf ein leeres Zeichen), wodurch die Suche in eine Endlosschleife gerät.

Bei dieser Art der Hervorhebung werden die gleichen Optionen wie bei der Textsuche verwendet (siehe Abschnitt - Suchen und Ersetz), daher gelten hier die gleichen Überlegungen. Der Text kann geändert werden, aber jedes Mal, wenn die gesuchte Sequenz gefunden wird, wird der Text erneut markiert. Es ist, als hätte man eine permanente Textsuchmaschine implementiert. Diese Methode zur Texthervorhebung funktioniert wie ein einfacher Syntax-Highlighter, ist jedoch eingeschränkt, da sie nur für einen einzigen Text gilt und nur Bezeichner und Symbole erkennt.

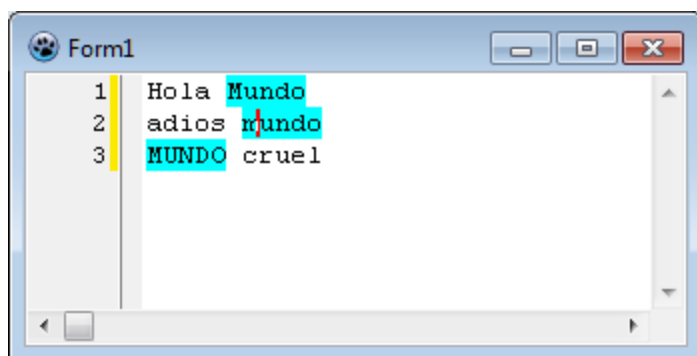
## 1.8.2 Hervorhebung des aktuellen Wortes

Es gibt eine weitere Form der Texthervorhebung, die nur für Wörter und die aktuelle Cursorposition gilt. Ein Wort kann nur alphanumerische Zeichen, das Dollarzeichen und den Unterstrich enthalten. Dazu gehören auch akzentuierte Vokale (und Umlaute) und der Buchstabe ñ.

Um diese Funktion zu nutzen, müssen wir die Unit "SynEditMarkupHighAll" einbinden und ein Objekt der Klasse "TSynEditMarkupHighlightAllCaret" erstellen.

```
uses ... , SynEditMarkupHighAll;
...
var
  SynMarkup: TSynEditMarkupHighlightAllCaret;
begin
  // Hervorhebung der gleichen Wörter beginnen
  SynMarkup := TSynEditMarkupHighlightAllCaret(
    SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret]);
  SynMarkup.MarkupInfo.Background := clAqua;
  SynMarkup.WaitTime := 250; // Zeit in Millisekunden
  SynMarkup.Trim := True;
  SynMarkup.FullWord := True; // passt nur auf das ganze Wort
```

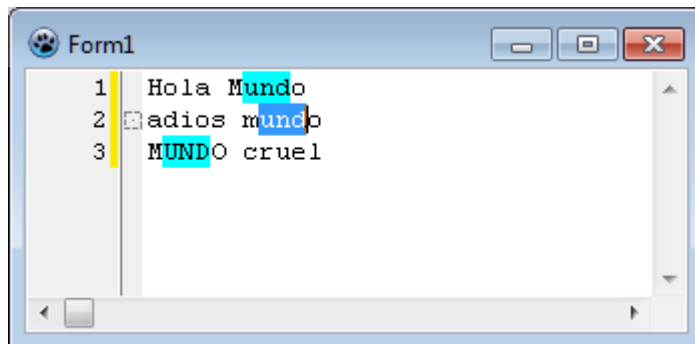
Dieser Code hebt Wörter hervor, die dem Wort entsprechen, auf dem sich der Cursor befindet, wie in der folgenden Abbildung dargestellt:



Der Editor erkennt automatisch das Wort unter dem Cursor und verwendet es, um den gesamten Text zu markieren. Standardmäßig wird die Art des Feldes (Groß- oder Kleinschreibung) ignoriert. Die Hervorhebung kann verschiedene Attribute wie Hintergrundfarbe, Textfarbe oder Rahmen enthalten. Der Zugriff auf diese Attribute erfolgt über die Eigenschaft "MarkupInfo".

Der auf "WaitTime" eingestellte Wert gibt an, wie viele Millisekunden gewartet wird, bis das Wort, an dem sich der Cursor befindet, erkannt wird. Wird kein Wert angegeben, so wird 2 Sekunden gewartet. Es wird nicht empfohlen, einen sehr niedrigen Wert für "WaitTime" zu verwenden, da dies den Editor dazu zwingen könnte, sehr häufig zu suchen.

Diese Markierungsoption funktioniert auch bei markiertem Text. Es wird also nur der Text durchsucht, der sich innerhalb der Auswahl befindet:



Es ist zu beachten, dass die Hervorhebung nicht den gesamten Inhalt des gesamten Editors liest. Standardmäßig werden nur bis zu 100 Zeilen vor und hinter dem aktuellen Bildschirm gelesen.

Um die Hervorhebung des aktuellen Wortes zu deaktivieren, können Sie die SynEdit zugewiesene Markierung deaktivieren:

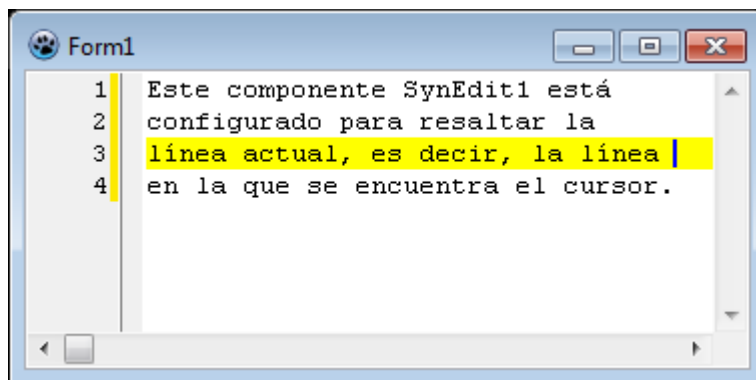
```
SynMarkup.Enabled := False ; // Hervorhebung deaktivieren
```

### 1.8.3 Hervorhebung der aktuellen Zeile

Oftmals ist es praktisch, die aktuelle Zeile zu markieren, um leicht zu erkennen, wo sich der Cursor befindet. Mit SynEdit können Sie die Hintergrundfarbe der Zeile, in der sich der Cursor befindet, einfach ändern:

```
SynEdit1.LineHighlightColor.Background := clYellow ;
```

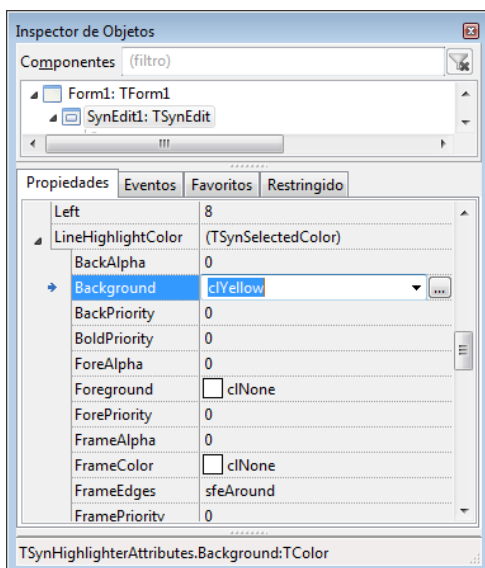
Die vorherige Anweisung färbt den Hintergrund der aktuellen Zeile gelb. Das Ergebnis sieht ähnlich aus wie in der folgenden Abbildung dargestellt:



Um die Hervorhebung der aktuellen Zeile zu deaktivieren, können Sie dieselbe Hintergrundfarbe des Editors oder die Farbe clNone verwenden:

```
SynEdit1.LineHighlightColor.Background := clNone ;
```

Die Hervorhebung der aktuellen Zeile kann auch über den Objektinspektor aktiviert werden, indem die Eigenschaft LineHighlightColor gesetzt wird:



Hier können Sie feststellen, dass es mehrere zusätzliche Eigenschaften zu "Hintergrund" gibt, um das Aussehen der hervorgehobenen Zeile zu ändern. Alle diese Eigenschaften können per Code oder über den Objektinspektor konfiguriert werden.



### 1.8.4 Hervorheben einer beliebigen Zeile

Sie können SynEdit anweisen, eine oder mehrere Zeilen des Inhalts zu markieren. Dazu müssen wir zunächst eine Methode in unserem Formular erstellen, die die zu markierende Zeile identifiziert und die Attribute zuweist:

```
procedure TForm1.SynEdit1SpecialLineMarkup ( Sender : TObject ;  
                                             Zeile : Ganzzahl ;var Special : boolean ;  
                                             Markup : TSynSelectedColor );  
begin  
  if Line = 2 then  
  begin  
    Special := True ; // als Spezialzeile markieren  
    Markup.Background := clGreen ; // Hintergrundfarbe  
  end;  
end;
```

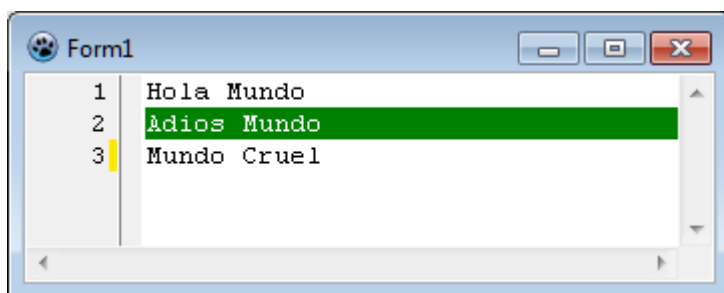
Der Typ "TSynSelectedColor" ist in der "SynEditMiscClasses"-Unit definiert, so dass es notwendig ist, ihn zuerst einzubinden, damit er funktioniert.

Dann müssen wir diese Methode dem Ereignis "OnSpecialLineMarkup" des Editors zuweisen:

```
SynEdit1.OnSpecialLineMarkup := @SynEdit1SpecialLineMarkup ;
```

Jedes Mal, wenn der Editor eine Zeile durchsucht, ruft er das Ereignis "OnSpecialLineMarkup" auf, um festzustellen, ob sie besondere Attribute hat oder eine gewöhnliche Zeile ist. Die Implementierung dieses Ereignisses muss schnell reagieren, um den Betrieb von SynEdit nicht zu stören.

Das Ergebnis dieses Codes würde in etwa so aussehen:



In diesem Fall haben wir die Markierung der zweiten Zeile aktiviert, und sie bleibt markiert, bis wir das Ereignis abbrechen. Die markierte Zeile wird immer die zweite sein, auch wenn Zeilen eingefügt oder gelöscht werden. Um eine Zeile zu markieren, die auf den enthaltenen Text folgt, muss eine weitere Verarbeitung hinzugefügt werden.

Wenn die hervorgehobene Zeile durch Code geändert wird, muss der Editor aktualisiert werden, um die neue hervorgehobene Zeile zu aktualisieren. Dies kann durch den Aufruf der Methode "SynEdit1.Refresh" geschehen, aber es wird empfohlen, die Methode "SynEdit1.Invalidate" zu verwenden, die effizienter ist.

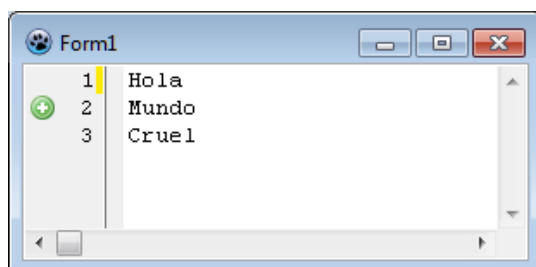
Es gibt auch das Ereignis `OnSpecialLineColors()`, mit dem Sie eine ähnliche Aufgabe durchführen können, aber nur die Hintergrundfarbe und die Textfarbe ändern können. Dieses Ereignis ist definiert als:

```
TSpecialLineColorsEvent = procedure(Sender:TObject;Line:integer;var Special:
                                boolean;var FG,BG : TColor) of object ;
```

## 1.8.5 Textmarkierungen

Highlighter sind spezielle Stellen, die in SynEdit-Inhalten gesetzt werden. Man könnte sagen, dass sie das digitale Äquivalent des Bandes (Trennlinie) sind, das zur Markierung einer Seite in einem Buch verwendet wird.

SynEdit-Lesezeichen speichern jedoch Positionen, die Zeilen und Spalten umfassen, und es können beliebig viele sein. Textmarkierungen können grafisch als Symbol in der vertikalen Leiste angezeigt werden:



Anders als die bisherigen Highlighter heben sie nicht den Inhalt des Editors hervor, sondern speichern eine Position und haben die Möglichkeit, ein Symbol im vertikalen Bereich des Editors anzuzeigen. Diese Lesezeichen sind nützlich, um die Positionen bestimmter Textzeilen zu speichern. Wenn Zeilen eingefügt oder gelöscht werden, versucht die Markierung, ihre Position beizubehalten und der Zeile zu folgen.

Um eine Markierung zu erstellen, müssen Sie die Unit `SynEditMarks` einbinden und eine Liste von Symbolen haben, die in einer `TImageList` gespeichert sind. Der folgende Code, der für die obige Abbildung verwendet wird, erstellt eine Markierung und macht sie in Zeile 2 sichtbar:

```
uses ... , SynEditMarks;
var Marke: TSynEditMark;
...
Marke := TSynEditMark.Create(SynEdit1); // Lesezeichen erstellen
Marke.ImageList:=ImageList1; // Liste der Icons zuweisen
Marke.ImageIndex:=0; // Wählen Sie Ihr Symbol
Marke.Line:=1; // legt die Zeile fest, in der es sich befindet
Marke.Visible:=true; // macht sie sichtbar
SynEdit1.Marks.Add(Marke); // Hinzufügen der Markierung zum Editor
```

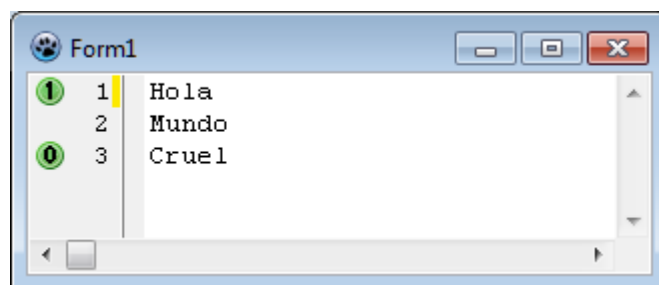
Auf diese Weise können verschiedene Marker an unterschiedlichen Positionen auf dem Bildschirm mit demselben oder einem anderen Symbol hinzugefügt werden. Es ist wichtig, dass die Zeile, in der sich die Markierung befindet, gültig ist, sonst wird sie nicht angezeigt. Die angezeigten Markierungen werden einfach als besondere Positionen im Text behandelt, in keiner bestimmten Reihenfolge.

Es gibt jedoch eine andere Art von Markierungen, die "Lesezeichen", die ebenfalls eine Nummer zur Identifizierung bieten.

Lesezeichen sogenannte "BookMarks" werden in der Regel anders erstellt. Eine ImageList ist ebenfalls erforderlich:

```
SynEdit1.BookMarkOptions.BookmarkImages := ImageList1 ;  
SynEdit1.SetBookmark ( 0, 1, 1 ); // Lesezeichen 0 in Zeile 1  
SynEdit1.SetBookmark ( 1, 1, 3 ); // Lesezeichen 1 in Zeile 3
```

Der erste Parameter von SetBookmark() ist die Nummer des erstellten "Lesezeichens" **und auch der Index der verwendeten ImageList**. Diese Nummer identifiziert Sie von nun an. Die beiden anderen Parameter sind die Zeile und die Spalte, die sie markieren. Obwohl das Symbol eine Zeile zu markieren scheint, speichern Lesezeichen tatsächlich die XY-Koordinaten. Obwohl die für das "Lesezeichen" verwendeten Symbole beliebiger Art sein können, ist es üblich, Symbole zu verwenden, die mit der Lesezeichennummer verbunden sind:



Um ein "Lesezeichen" zu löschen, müssen Sie ihre Nummer kennen:

```
SynEdit1.ClearBookmark ( 5 );
```

Mit diesem Code wird das "Lesezeichen"-Nummer 5 entfernt. Wenn die "Lesezeichen"-Nummer nicht vorhanden ist, wird der Befehl ignoriert.

Um auf eine Markierung zuzugreifen (es kann ein Lesezeichen sein oder auch nicht), wird es wie ein Element eines Arrays behandelt:

```
mark := SynEdit1.Marks[0] ;  
if mark.IsBookmark then showmessage('bookmark');
```

Mit der Eigenschaft "IsBookmark" können Sie feststellen, ob eine Markierung ein Lesezeichen ist.

Sie können auch ein "Lesezeichen" wie jedes andere Markierung erstellen:

```
uses ... , SynEditMarks;
var Markierung: TSynEditMark;
...
SynEdit1.BookMarkOptions.BookmarkImages := ImageList1;
Markierung := TSynEditMark.Create(SynEdit1) ;
Markierung.Line := 1;
Markierung.Column := 5;
Markierung.ImageIndex := 0;
Markierung.BookmarkNumber := 1;
Markierung.Visible := true;
Markierung.InternalImage := false;
SynEdit1.Marks.Add(Markierung);
```

Die "InternalImage" Eigenschaft ist in der aktuellen Version von Lazarus nicht funktional. Sie erinnert an eine Funktion, die es erlaubte, interne Icons für Highlighter zu verwenden.

Eine weitere Möglichkeit, ein "Lesezeichen" zu erstellen, besteht darin, es als Befehl hinzuzufügen:

```
uses ... , SynEditKeyCmds ;

SynEdit1.BookMarkOptions.BookmarkImages := ImageList1 ;
SynEdit1.ExecuteCommand(ecSetMarker4,#0,nil);//erzeugt das 4 Lesezeichen mit ImageIndex 4
```

Es sind 10 Konstanten für "Lesezeichen" definiert: ecSetMarker0 .. ecSetMarker9.

Es gibt auch 10 vordefinierte Konstanten, um den Zustand der Lesezeichen zu ändern:

ecToggleMarker0 .. ecToggleMarker9 .

Standardmäßig werden in SynEdit Tastenkombinationen zum Hinzufügen von Lesezeichen (BookMarks) erstellt, wenn Sie das Steuerelement hinzufügen. Diese Shortcuts sind vom Typ <Umschalt>+<Strg>+<Markierungsnummer>.

Der Versuch, eine Markierung hinzuzufügen, ohne dass eine Symbolliste zugewiesen wurde, führt zu einem Laufzeitfehler.

Um Lesezeichen zu deaktivieren, können Sie den Teil, der den Lesezeichen entspricht, in der vertikalen Leiste (Gutter) ausblenden, oder Sie können auch Tastenkombinationen eliminieren (Eigenschaft "Tastenkombinationen").

```
SynEdit1.Marks[0].Visible:= false;
```

## 1.8.6 Zugriff auf Lesezeichen

Die grundlegendsten Marker in SynEdit sind Objekte der Klasse "TSynEditMarkup". Alle Marker sind direkt oder indirekt von "TSynEditMarkup" abgeleitet.

Die Klasse "TSynEditMarkup" ist eine mehr oder weniger umfangreiche Klasse, die in der Unit "SynEditMarkup" definiert ist. Ihre öffentlichen Felder sind:

```
TSynEditMarkup = class(TObject)
private
...
protected
...
public
  constructor Create(ASynEdit : TSynEditBase);
  destructor Destroy; override;
  procedure PrepareMarkupForRow(aRow : Integer); virtual;
  procedure FinishMarkupForRow(aRow : Integer); virtual;
  procedure EndMarkup; virtual;
  function GetMarkupAttributeAtRowCol(const aRow: Integer;const aStartCol:
      TLazSynDisplayTokenBound;const AnRtlInfo:TLazSynDisplayRtlInfo)
      :TSynSelectedColor;abstract;
  procedure GetNextMarkupColAfterRowCol(const aRow: Integer;const aStartCol:
      TLazSynDisplayTokenBound;const AnRtlInfo: TLazSynDisplayRtlInfo;out
      ANextPhys, ANextLog: Integer); virtual;abstract;
  procedure MergeMarkupAttributeAtRowCol(const aRow: Integer;const aStartCol,
      AEndCol :TLazSynDisplayTokenBound;const AnRtlInfo:
      TLazSynDisplayRtlInfo;AMarkup: TSynSelectedColorMergeResult); virtual;
  procedure TextChanged(aFirstCodeLine, aLastCodeLine, ACountDiff: Integer);
      virtual;
  procedure TempDisable;
  procedure TempEnable;
  procedure IncPaintLock; virtual;
  procedure DecPaintLock; virtual;
  function RealEnabled: Boolean; virtual;

  property MarkupInfo : TSynSelectedColor read fMarkupInfo;
  property FGColor : TColor read GetFGColor;
  property BGColor : TColor read GetBGColor;
  property FrameColor: TColor read GetFrameColor;
  property FrameStyle: TSynLineStyle read GetFrameStyle;
  property Style : TFontStyles read GetStyle;
  property Enabled: Boolean read GetEnabled write SetEnabled;
  property Lines : TSynEditStrings read fLines write SetLines;
  property Caret : TSynEditCaret read fCaret write SetCaret;
  property TopLine : Integer read fTopLine write SetTopLine;
  property LinesInWindow : Integer read fLinesInWindow write SetLinesInWindow;
  property InvalidateLinesMethod : TInvalidateLines write
      SetInvalidateLinesMethod;
```

Um eine neue Markierung zu erstellen, müssen wir ein Objekt des Typs "TSynEditMarkup" oder einen seiner Abkömmlinge erstellen.

Wir haben bereits gesehen, wie man eine Markierung erstellt, im Abschnitt - Hervorhebung des aktuellen Wortes gesehen, als wir ein Objekt der Klasse "TSynEditMarkupHighlightAllCaret" erstellt haben, da diese Klasse von "TSynEditMarkup" abgeleitet ist. Auch die Klasse "TSynEditMarkupHighlightAll", mit der ein beliebiges Wort markiert werden kann, ist ein Abkömmling von "TSynEditMarkup".

Die SynEdit-Komponente wird mit mehreren standardmäßig definierten Markern geliefert. Sie können durch Iteration über die Tabelle Markup[] erkundet werden:

```
// Iterieren durch Markierungen
for i := 0 to SynEdit1.MarkupCount - 1 do begin
    tmp := SynEdit1.Markup[i].MarkupInfo.StoredName ; // Name lesen
    ShowMessage(tmp);
end;
```

Bei den meisten vordefinierten Lesezeichen in SynEdit ist das Feld "StoredName" leer, so dass die vorige Schleife Null-Strings anzeigt, aber wenn wir ein benanntes Lesezeichen erstellt hätten, könnten wir es auf diese Weise platzieren.

Das folgende Beispiel zeigt, wie Sie ein Lesezeichen mit dem Namen "CurrentWordRes" deaktivieren können:

```
var
    mark : TSynEditMarkup ;
    ...

mark := nil ;
// Lesezeichen nach Name suchen
for i := 0 to SynEdit1.MarkupCount - 1 do
    begin
        tmp := SynEdit1.Markup[i].MarkupInfo.StoredName ;
        if SynEdit1.Markup[i].MarkupInfo.StoredName = 'CurrentWordRes' then
            mark := SynEdit1.Markup[i] ;
        end;
    if mark <> nil then
        begin // es gibt eine Markierung
            mark.Enabled := false ; // deaktivieren
        end;
```

Eine weitere Möglichkeit, eine Markierung in SynEdit zu finden, ist die Verwendung der Methode MarkupByClass[], mit der wir den Namen der Klasse der gesuchten Markierung angeben können:

```
mark := SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret];
```

Gibt es zwei Marker der gleichen Klasse, wird der erste zurückgegeben.

Der folgende Code zeigt, wie die Rahmenfarbe des hervorgehobenen Textes durch die Markierung `TSynEditMarkupHighlightAllCaret` festgelegt wird:

```
var
  mark : TSynEditMarkup ;
  ...
  mark := SynEdit1.MarkupByClass[TSynEditMarkupHighlightAllCaret] ;
  TSynEditMarkupHighlightAllCaret(mark).MarkupInfo.FrameColor := clGreen;
```

Um auf die visuellen Eigenschaften des hervorgehobenen Textes zuzugreifen, wird die Eigenschaft "MarkupInfo" des Highlighters verwendet.

Die Eigenschaft "MarkupInfo" ist ein Abkömmling der Klasse "TSynHighlighterAttributes" (siehe Abschnitt - **Wichtige Konzepte**), sie fungiert also als Container für die Erscheinungsmerkmale eines Textes.

Wenn Sie eine Markierung erstellen, ist es praktisch, ihr einen Namen zu geben und ihn in der Eigenschaft "StoredName" von "MarkupInfo" zu speichern, um sie später im Editor wiederzufinden.

### 1.8.7 Mehr über Lesezeichen

Die in den vorherigen Abschnitten beschriebenen Markierungen, die das Aussehen des angezeigten Textes hervorheben, sind relativ einfach (Hervorhebung eines Wortes oder einer Zeile). Die Lesezeichen von SynEdit sind jedoch recht umfangreich und ermöglichen es Ihnen, komplexe Textmarkierungen vorzunehmen.

Ein Beispiel für umfassendere Markierungen sind die vom Lazarus-Editor verwendeten, die es erlauben, den Anfang und das Ende von Codeblöcken wie BEGIN-END oder REPEAT-UNTIL hervorzuheben.

Das folgende Beispiel geht von einem Formular aus, das ein SynEdit-Steuerelement enthält, und zeigt, wie ein einfaches Lesezeichen erstellt wird, das einen Textblock in der ersten Zeile hervorhebt:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, Forms, Controls, Graphics, SynEdit,
  SynEditTypes, SynEditMarkupHighAll;

type
  TMarkup = class(TSynEditMarkupHighlightMatches);

  { TForm1 }
  TForm1 = class(TForm)
    SynEdit1: TSynEdit;
    procedure FormCreate(Sender: TObject);
  private
    Markup: TMarkup; //Markierung 1
  end;

var
  Form1: TForm1;

implementation

{$R *.lfm}
{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);
begin
  Markup := TMarkup.Create(SynEdit1);
  Markup.MarkupInfo.FrameColor := clRed;
  Markup.MarkupInfo.FrameEdges := sfeBottom;
  SynEdit1.MarkupManager.AddMarkup(Markup);
  //Markierung hinzufügen
  Markup.Matches.StartPoint[0] := Point(2,1);
  Markup.Matches.EndPoint[0] := Point(4,1);
  Markup.InvalidateSynLines(1,1);
end;

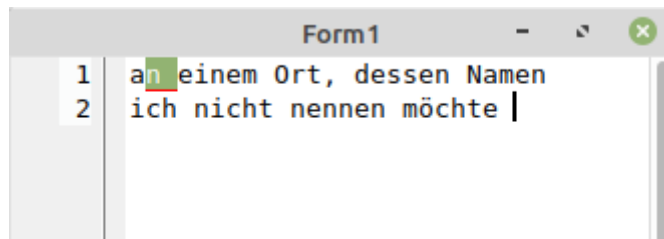
end.
```

In diesem Fall wird die Klasse `TSynEditMarkupHighlightMatches` als Grundlage für die Markierung verwendet, und da der Zugriff auf die geschützte Eigenschaft "Matches" von `TSynEditMarkupHighlightMatches` erforderlich ist, wird die Klasse `TMarkup` definiert. Andernfalls wäre `TSynEditMarkupHighlightMatches` ausreichend gewesen.



Der hervorzuhebende Block wird mit Koordinaten vom Typ TPoint, in StartPoint[] und EndPoint[], definiert. Beachten Sie, dass das Lesezeichen standardmäßig aktiviert ist.

Die Auswirkungen sind im Text wie in der Abbildung dargestellt:



Um umfangreichere Lesezeichen zu erstellen, müssen Sie auf die geschützten Felder von SynEdit zugreifen, also müssen Sie eine Klasse erstellen, die von SynEdit abgeleitet ist, und unseren Editor aus dieser Klasse erstellen. Sobald unser Editor definiert ist, haben wir die Möglichkeit, beliebige Markierungen zu erstellen, deren Logik durch Code definiert werden muss.

Der folgende Code erstellt einen Editor aus SynEdit, indem er eine abhängige Klasse von SynEdit (Unterklasse) erstellt und zwei Textblöcke definiert, um sie hervorzuheben:

```
unit Unit1; {$mode objfpc}{$H+}
interface
uses
Classes, SysUtils, Forms, Controls, Graphics,
SynEdit, SynEditMarkupSelection,
SynEditPointClasses, SynEditMarkup;

type
{ TMiEditor }
TMiEditor = class(TSynEdit) //definiert die Klasse MiEditor
private
Bloque1: TSynEditSelection;
// definiert Abschnitt 1
Markup1: TSynEditMarkupSelection; //Markierung 1
Bloque2: TSynEditSelection;
// definiert Abschnitt 2
Markup2: TSynEditMarkupSelection; //Markierung 2
public
constructor Create(AOwner: TComponent); override;
destructor Destroy; override;
end;

{ TForm1 }
TForm1 = class(TForm)
procedure FormCreate(Sender: TObject);
procedure FormDestroy(Sender: TObject);
private
MiEditor: TMiEditor;
end;

var
Form1: TForm1;

implementation
```

```

{ TMiEditor }
constructor TMiEditor.Create(AOwner: TComponent);
var aMarkupManager: TSynEditMarkupManager;
begin
  inherited Create(AOwner);
  aMarkupManager := TSynEditMarkupManager(MarkupMgr);
  //erstellt einen Block für Bemerkungen
  Bloque1 := TSynEditSelection.Create(ViewedTextBuffer, false);
  Bloque1.InvalidateLinesMethod := @InvalidateLines;
  Markup1 := TSynEditMarkupSelection.Create(self, Bloque1);
  aMarkupManager.AddMarkup(Markup1); //Markierung hinzufügen
  //erstellt einen Block für Bemerkungen
  Bloque2 := TSynEditSelection.Create(ViewedTextBuffer, false);
  Bloque2.InvalidateLinesMethod := @InvalidateLines;
  Markup2 := TSynEditMarkupSelection.Create(self, Bloque2);
  aMarkupManager.AddMarkup(Markup2); //Markierung hinzufügen
end;

destructor TMiEditor.Destroy;
begin
  Bloque1.Free;
  //Markup1, zerstört mit SynEdit
  Bloque2.Free;
  //Markup2, zerstört mit SynEdit
  inherited Destroy;
end;

{$R *.lfm}
{ TForm1 }

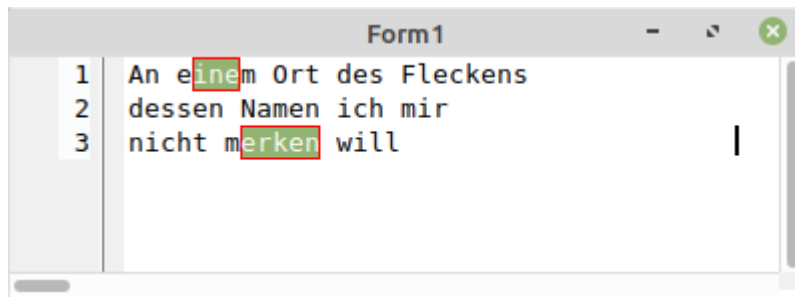
procedure TForm1.FormCreate(Sender: TObject);
begin
  MiEditor := TMiEditor.Create(self); //Komponente erstellen MeinEditor
  MiEditor.Parent := self; //setzt es in das Formual
  MiEditor.Align := alClient; //richtet sie aus
  //einen Text schreiben
  MiEditor.Lines.Add('An einem Ort des Fleckens');
  MiEditor.Lines.Add('dessen Namen ich mir');
  MiEditor.Lines.Add('nicht merken will');
  //definiert Abschnitt 1 wie folgt
  MiEditor.Bloque1.StartLineBytePos := Point(5,1);
  MiEditor.Bloque1.EndLineBytePos := Point(8,1);
  MiEditor.Markup1.Enabled := True;
  MiEditor.Markup1.MarkupInfo.FrameColor := clRed;
  //definiert Abschnitt 2 wie folgt
  MiEditor.Bloque2.StartLineBytePos := Point(8,3);
  MiEditor.Bloque2.EndLineBytePos := Point(13,3);
  MiEditor.Markup2.Enabled := True;
  MiEditor.Markup2.MarkupInfo.FrameColor := clRed;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  MiEditor.Destroy;
end;

end.

```

Die Auswirkungen auf den Text sind ähnlich wie in der folgenden Abbildung dargestellt:



Für diese Hervorhebung wurde die Klasse "TSynEditSelection" verwendet, die in der Unit "SynEditPointClasses" definiert ist und die es ermöglicht, Auswahlbereiche innerhalb des Editors festzulegen. Standardmäßig hat sie die gleiche Hintergrundfarbe wie die Auswahl, aber sie kann mit der Eigenschaft "MarkupInfo.Background" geändert werden.

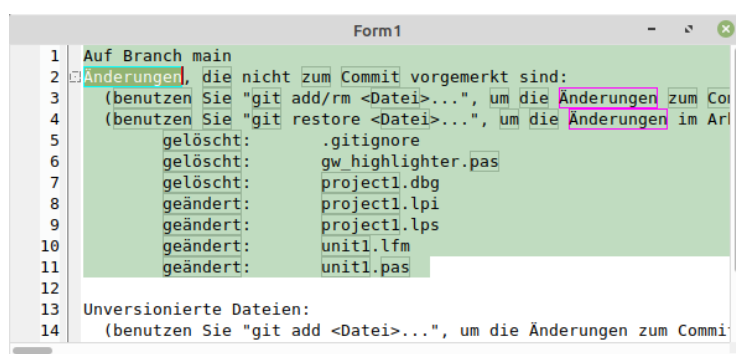
Eine Beobachtung bei dieser Hervorhebung ist, dass die hervorgehobenen Bereiche feste Koordinaten haben, auch wenn der Text geändert wird. Wenn Sie möchten, dass die Hervorhebung dem Text folgt, muss die erforderliche Logik durch Code implementiert werden.

## 1.9 Verwendung von Plugins

Einige Funktionen von SynEdit werden in Form von "Plugins" angeboten, die nichts anderes als Units sind, die die SynEdit zusätzliche Funktionen hinzufügen.

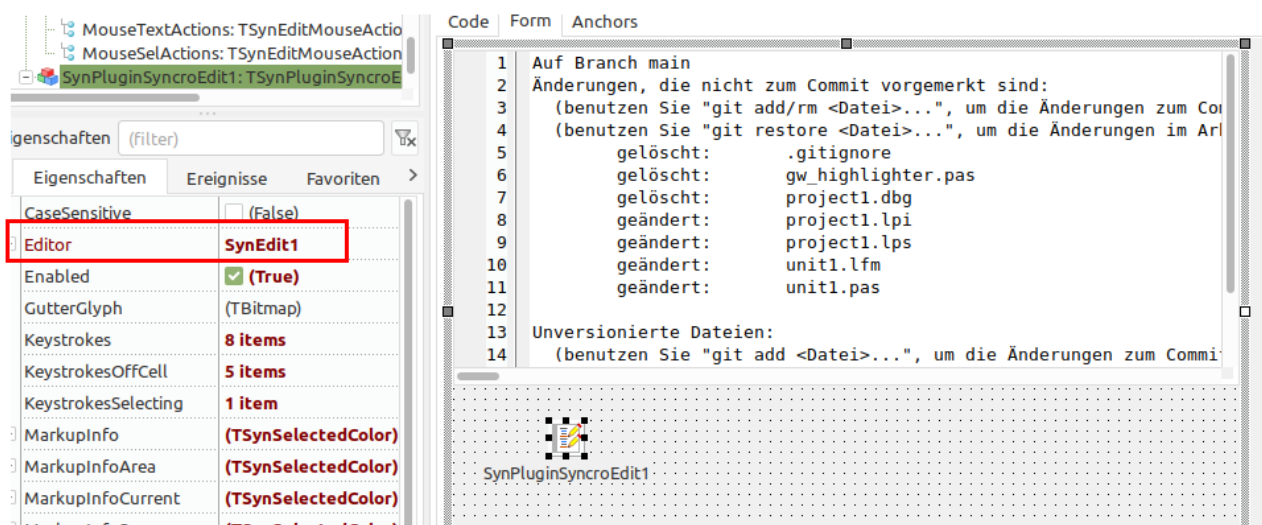
### 1.9.1 Synchrone Bearbeitung

Die synchrone Bearbeitung ermöglicht es Ihnen, denselben Bezeichner an verschiedenen Stellen eines Textes zu bearbeiten.



Um sie zu starten, müssen Sie einen Textblock markieren und die Tastenkombination Strg-J drücken, die standardmäßig die synchrone Bearbeitung startet. In diesem Moment ändert sich die Farbe der Auswahl und die Bezeichner, die mit demjenigen übereinstimmen, an dem sich der Cursor befindet, werden markiert. Wenn in diesem Zustand der aktuelle Bezeichner bearbeitet wird, ändern sich alle anderen ähnlichen Bezeichner in der Auswahl.

Der einfachste Weg, die synchrone Bearbeitung zu implementieren, wäre, ein SynPluginSyncoEdit-Element aus der Komponentenpalette hinzuzufügen und es dann mit einem SynEdit-Element zu verknüpfen, indem man seine Eigenschaft "Editor" setzt.



Eine andere Möglichkeit wäre, dies per Code zu tun, indem Sie die `SynPluginSyncroEdit`-Unit einschließen und ein `TSynPluginSyncroEdit`-Objekt erstellen. Auch in diesem Fall müssen Sie seine "Editor"-Eigenschaft auf den `SynEdit` setzen, dem Sie die synchrone Bearbeitungsfunktionalität hinzufügen möchten.

### 1.9.2 Mehrere Cursor

SynEdit kann die Bearbeitung mit mehreren Cursors durchführen, als ob es nur einen gäbe.

Um diese Funktionalität zu aktivieren, müssen Sie die Unit `SynPluginMultiCaret` einbinden, ein Objekt der Klasse `TSynPluginMultiCaret` erstellen und dessen Feld "Editor" für das `SynEdit` konfigurieren, dem wir die Mehrfachcursor-Funktionalität hinzufügen möchten.

Der folgende Code konfiguriert ein `SynEdit` für die Bearbeitung mit mehreren Cursors und fügt die Kombination `Strg+Umschalt+Klick` hinzu, um einen neuen Cursor zu positionieren:

```
fMultiCaret := TSynPluginMultiCaret.Create(self);
with fMultiCaret do
begin
  Editor := SynEdit1;
  with KeyStrokes do
  begin
    Add.Command := ecPluginMultiCaretSetCaret ;
    Add.Key := VK_INSERT ;
    Add.Shift := [ssShift, ssCtrl] ;
    Add.ShiftMask := [ssShift,ssCtrl,ssAlt] ;
  end;
end;
```

Wenn Sie dann die Tasten `Strg+Umschalt` gedrückt halten und mit der Maus irgendwo in den Text klicken, wird an dieser Stelle ein neuer Cursor für die Bearbeitung aktiviert.

Wenn Sie mehrere Cursor haben, wird die von Ihnen durchgeführte Aktion auf alle erstellten Cursor angewendet, einschließlich des Einfügens und Löschens von Zeichen.

Um einen neuen Cursor zu erstellen, können Sie die Methode Code verwenden:

```
SynEdit1.CommandProcessor(ecPluginMultiCaretSetCaret, '', nil);
```

## 1.10 Zusammenfassung der Eigenschaften und Methoden

| EIGENSCHAFT                    | BESCHREIBUNG   |
|--------------------------------|--|
| BeginUpdate, EndUpdate         | Deaktiviert bzw. Aktiviert die Bildschirmaktualisierung von SynEdit. Siehe - <a href="#">und Rückgängig machen</a> .   |
| BeginUndoBlock<br>EndUndoBlock | Sie ermöglichen es, mehrere Änderungsaktionen zu gruppieren, um sie mit einem einzigen Aufruf von Undo rückgängig zu machen. Siehe - <a href="#">und Rückgängig machen</a> .   |
| BlockBegin<br>BlockEnd         | Sie geben die Koordinaten des ausgewählten Textes an. Siehe Abschnitt - <a href="#">Verwaltung der Auswahl</a>   |
| BlockIndent                    | Gibt die Anzahl der Leerzeichen an, die verwendet werden, um einen Textblock einzurücken, wenn Blockeinrückungsaktionen ausgeführt werden (in der Lazarus IDE werden sie als Ctrl+U und Ctrl+I abgebildet).                                |
| BookMarkOptions                | Es handelt sich um eine Reihe von Optionen, mit denen Sie Textmarkierungen konfigurieren können.   |
| BracketHighlightStyle          | Konfiguriert das Hervorhebungsverhalten von Begrenzungszeichen in Klammern, geschweiften Klammern und eckigen Klammern. Gibt an, wie die Hervorhebung bestimmt wird. Sie können sein:<br>sbhsBoth<br>sbhsLeftOfCursor<br>sbhsRightOfCursor |
| BracketMatchColor              | Attribut zur Hervorhebung von Begrenzungszeichen in Klammern, geschweiften Klammern und eckigen Klammern.  |
| CanUndo                        | Zeigt an, ob es Aktionen gibt, die im Editor rückgängig gemacht werden können  |
| CanRedo                        | Zeigt an, ob es Aktionen gibt, die im Editor erneut durchgeführt werden müssen   |
| CanPaste                       | Zeigt an, ob Inhalte zum Einfügen in den Editor vorhanden sind   |
| CaretX                         | Mit ihnen können Sie die horizontale Koordinate des Editor-Cursors ablesen oder einstellen. Das erste Zeichen hat die Koordinate 1.  |
| CaretY                         | Mit ihnen können Sie die vertikale Koordinate des Editor-Cursors ablesen oder einstellen. Die erste Zeile hat die Koordinate 1.  |
| CaretXY                        | Mit ihnen können Sie die X- und Y-Koordinaten des Editor-Cursors ablesen oder einstellen. Siehe Abschnitt - <a href="#">Editor</a>   |
| ClearAll                       | Löschen Sie alle Inhalte aus dem Editor.   |
| ClearSelection                 | Löscht den markierten Text.  |
| ClearUndo                      | Löschen und Zurücksetzen des "Rückgängig"-Speichers. Einmal durchgeführte Änderungen können nicht mehr rückgängig gemacht werden.  |
| Color                          | Hintergrundfarbe des Editors.  |
| CopyToClipboard                | Kopiert den markierten Text in die Zwischenablage. Siehe Abschnitt - <a href="#">Die Zwischenablage</a> .  |
| CutToClipboard                 | Schneidet den markierten Text in die Zwischenablage aus. Siehe Abschnitt - <a href="#">Die Zwischenablage</a> .  |

|                                |  |
|--------------------------------|--|
| ExecuteCommand                 | Senden Sie einen Befehl an den Editor. Siehe Abschnitt - <a href="#">Befehle ausführen</a>   |
| ExtraCharSpacing               | Gibt den Abstand zwischen den Buchstaben an. Standardmäßig ist er gleich Null. Siehe Abschnitt - <a href="#">Typografie</a> .  |
| ExtraLineSpacing               | Gibt den Abstand zwischen den Zeilen an. Standardmäßig ist er gleich Null. Siehe Abschnitt - <a href="#">Typografie</a> .  |
| FoldAll                        | Ermöglicht es Ihnen, alle in einem Editor vorhandenen Faltblöcke zu schließen. Sie können auch Ebenen einklappen.  |
| Font                           | Objekt, das die Eigenschaften der im Editor zu verwendenden Schriftart definiert. Es hat verschiedene Eigenschaften wie Schriftartname, Größe, Zeichensatz usw. Siehe Abschnitt - <a href="#">Typografie</a> |
| Gutter                         | Verweis auf das Objekt, das das Seitenfeld des Editors definiert, in dem normalerweise die Zeilennummer erscheint.   |
| GetHighlighterAttriAtRowCol    | Liest das Token und das Attribut an einer bestimmten Stelle im Text. Es ist nur gültig, wenn Sie einen Highlighter mit dem Editor verbunden haben.   |
| Highlighter                    | Verweis auf den Highlighter, der zur Implementierung der Syntaxhervorhebung verwendet werden soll (siehe Abschnitt - <a href="#">Syntaxfärbung mit Code</a> ).   |
| InsertMode                     | Ermöglicht es Ihnen, vom normalen Modus in den INSERT-Modus zu wechseln, in dem die eingegebenen Zeichen überschrieben werden. Wenn Sie FALSE wählen, gelangen Sie in den Einfügemodus.                      |
| InsertTextAtCaret              | Fügt Text an der aktuellen Cursorposition ein. Siehe Abschnitt - <a href="#">Ändern Sie den Inhalt</a>   |
| Keystrokes                     | Speichert Tastaturkürzel und die Befehle, denen diese Tastaturkürzel zugeordnet sind.  |
| LogicalCaretXY                 | Gibt die Position des Cursors in logischen Koordinaten zurück. Gemessen in Bytes (nicht in Zeichen).   |
| LineHighlightColor             | Es ist die Hintergrundfarbe der Zeile, auf der sich der Cursor gerade befindet. Siehe Abschnitt - <a href="#">Hervorhebung der aktuellen Zeile</a>   |
| LinesInWindow<br>CharsInWindow | Gibt die Anzahl der Zeilen und Spalten an, die auf dem Bildschirm sichtbar sind. Sie hängt nur von der Größe des Bildschirms und den Abständen zwischen Zeichen und Zeilen ab.                               |
| Lines                          | Liste aller Verlagsinhalte. Es handelt sich um eine Liste von Strings (ähnlich wie TStringList), wobei jedes Element eine Zeile darstellt. Beginnt bei Element 0.  |
| LineText                       | Speichert immer den Inhalt der aktuellen Zeile.  |
| MaxLeftChar                    | Begrenzt die horizontale Position des Cursors im Modus "Fließender Cursor" - Siehe Abschnitt - <a href="#">Handhabung des Cursors</a>  |
| MaxUndo                        | Maximale Anzahl der rückgängig zu machenden Vorgänge.  |
| Modified                       | Zeigt an, wenn der Inhalt des Editors geändert wurde. Er kann auch geschrieben werden.   |
| MoveCaretIgnoreEOL             | Positioniert den Cursor ohne Berücksichtigung der Grenzen der Zielzeile.   |

|                     |  |
|---------------------|--|
| Options             | Verschiedene zusätzliche Optionen zur Konfiguration des Editors, wie z. B. automatisches Einrücken, das Verhalten des Cursors außerhalb der Zeilengrenzen oder die Umwandlung von Tabulatoren in Leerzeichen. Siehe Abschnitt - <a href="#">Optionen</a> |
| PasteFromClipboard  | Fügt den Text aus der Zwischenablage an der Cursorposition ein. Siehe Abschnitt - <a href="#">Die Zwischenablage</a> .   |
| Redo                | Führen Sie eine Rückgängig-Aktion erneut mit "Rückgängig" aus. Siehe Abschnitt - <a href="#">und Rückgängig machen</a> .   |
| RightEdge           | Gibt die Position der vertikalen Linie (rechter Rand) an, mit der die Begrenzung des Druckbereichs markiert wird. Siehe Abschnitt - <a href="#">Erscheinungsbild</a>   |
| RightEdgeColor      | Farbe des rechten Randes.  |
| SelectAll           | Alle Inhalte aus dem Editor auswählen  |
| SelText             | Eigenschaft, mit der Sie den markierten Text lesen oder ändern können. Siehe Abschnitt - <a href="#">Verwaltung der Auswahl</a>  |
| SelectionMode       | Cursor-Auswahlmodus. Ermöglicht es Ihnen, den normalen Textauswahlmodus zu ändern.   |
| SearchReplace       | Ermöglicht das Suchen und Ersetzen von Text innerhalb des Editors. Siehe Abschnitt - <a href="#">Suchen und Ersetz</a>   |
| SelectWord          | Wählt das aktuelle Wort an der Stelle aus, an der sich der Cursor befindet. Siehe Abschnitt - <a href="#">Verwaltung der Auswahl</a> .   |
| SelectLine          | Wählen Sie die aktuelle Zeile aus, in der sich der Cursor befindet. Siehe Abschnitt - <a href="#">Verwaltung der Auswahl</a> .   |
| SelectParagraph     | Wählen Sie den aktuellen Absatz aus, in dem sich der Cursor befindet. Siehe Abschnitt - <a href="#">Verwaltung der Auswahl</a> .   |
| SelectToBrace       | Wählt den Block aus, der durch Klammern, geschweifte oder eckige Klammern begrenzt ist. Siehe Abschnitt - <a href="#">Verwaltung der Auswahl</a> .   |
| ScrollBars          | Steuert die Sichtbarkeit der horizontalen und vertikalen Bildlaufleisten.  |
| TabWidth            | Dies ist die Anzahl der Leerzeichen, die zur Darstellung eines Tabulatorstopps verwendet werden. Diese Option kann im Smart-Tab-Modus ignoriert werden (Options.eoSmartTabs = TRUE)  |
| TextBetweenPoints   | Ermöglicht es Ihnen, den Inhalt eines Textblocks zu lesen oder zu ändern.  |
| TextBetweenPointsEx | Ermöglicht es Ihnen, den Inhalt eines Textblocks zu lesen oder zu ändern, mit Cursorsteuerung.   |
| TopLine             | Gibt die Nummer der ersten Zeile an, die im aktuellen Editorfenster sichtbar ist.  |
| Undo                | Macht die zuletzt durchgeführte Aktion im Editor rückgängig. Siehe Abschnitt - <a href="#">und Rückgängig machen</a> .   |
| UnfoldAll           | Ermöglicht es Ihnen, alle geschlossenen Faltblöcke zu erweitern.   |



### 1.10.1 Optionen und Optionen2 Eigenschaft

Es gibt verschiedene Parameter, die über die Eigenschaft "Optionen" von SynEdit konfiguriert werden können. Die übliche Syntax ist:

```
SynEdit1.Options := [eoKeepCaretX, eoTabIndent, ... ];
```

Diese Eigenschaft ist eine Menge des Typs "TSynEditorOptions", die die folgenden Elemente enthalten kann:

| WERT                | BESCHREIBUNG   |
|---------------------|--|
| eoAutoIndent        | Positioniert den Cursor in der neuen Zeile mit der gleichen Anzahl von Leerzeichen wie in der vorherigen Zeile.  |
| eoBracketHighlight  | Dabei werden die Begrenzungszeichen in Klammern, geschweifte Klammern und eckige Klammern hervorgehoben.   |
| eoEnhanceHomeKey    | Die "Home"-Taste springt an den Anfang der Zeile, wenn sie näher ist (ähnlich wie in Visual Studio)  |
| eoGroupUndo         | Fasst alle Änderungen desselben Typs in einer einzigen Rückgängig/Wiederherstellen-Aktion zusammen, anstatt jeden Befehl einzeln zu behandeln.   |
| eoHalfPageScroll    | Beim Blättern nach oben oder unten wird jeweils nur eine halbe Seite übersprungen.   |
| eoHideRightMargin   | Blendet die rechte Randlinie aus.  |
| eoKeepCaretX        | Begrenzt die Position des Cursors, der nur bis zum Ende der Zeile bewegt werden kann. Ansonsten kann er über die Zeile hinausgehen. Siehe Abschnitt - <a href="#">Handhabung des Cursors</a> |
| eoNoCaret           | Macht den Cursor unsichtbar.   |
| eoNoSelection       | Textauswahl deaktivieren.  |
| eoPersistentCaret   | Versteckt den Cursor nicht, wenn der Fokus verloren geht.  |
| eoScrollByOneLess   | Das Hoch- und Runterblättern erfolgt mit einer Zeile weniger.  |
| eoScrollPastEof     | Sie ermöglichen es dem Cursor, sich über die Markierung für das Dateiende hinaus zu bewegen.   |
| eoScrollPastEol     | Ermöglicht es dem Cursor, sich über das letzte Zeichen einer Zeile hinaus zu bewegen. Dies funktioniert auch, wenn eoKeepCaretX vorhanden ist.   |
| eoScrollHintFollows | Der Etikettenlauf (Hinweis) folgt dem Mauslauf bei vertikaler Bewegung.  |
| eoShowScrollHint    | Zeigt ein Etikett (Hinweis) mit der Nummer der ersten sichtbaren Zeile an, wenn vertikal geblättert wird.  |
| eoShowSpecialChars  | Zeigt Sonderzeichen an.  |
| eoSmartTabs         | Wenn Sie die Tabulatortaste drücken, befindet sich der Cursor an der Position des nächsten Leerzeichens in der vorherigen Zeile.   |
| eoTabIndent         | Mit den Tasten <Tab> und <Umschalt><Tab> können Sie einen ausgewählten Block ein- oder ausrücken.  |

|                      |   |
|----------------------|---|
| eoTabsToSpaces       | Bewirkt, dass die <Tab>-Taste Leerzeichen (angegeben in der Eigenschaft "TabWidth") anstelle des TAB-Zeichens einfügt. Vorhandene Tabulatorzeichen werden nicht umgewandelt, es wird lediglich verhindert, dass weitere hinzugefügt werden. |
| eoTrimTrailingSpaces | Leerzeichen am Ende von Zeilen werden abgeschnitten und nicht gespeichert.  |

Zusätzlich zu "Options" können weitere Werte in die Eigenschaft "Options2" aufgenommen werden:

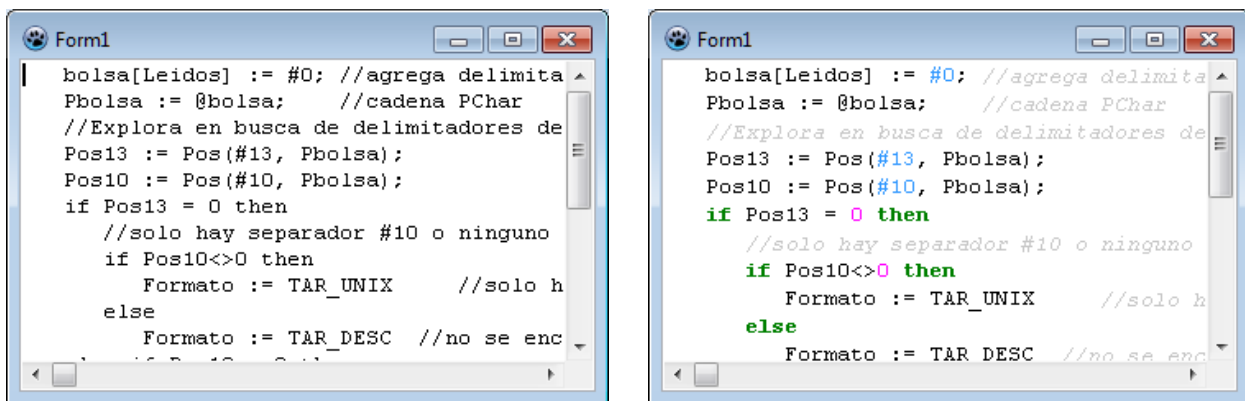
| WERT                  | BESCHREIBUNG   |
|-----------------------|--|
| eoCaretSkipsSelection | Der Cursor springt über die Auswahl, wenn Sie die Pfeiltasten nach links und rechts verwenden.   |
| eoCaretSkipTab        | Der Cursor überspringt alle Leerzeichen, die einen Tabstopp bilden. Ohne diese Option bewegt sich der Cursor durch die Tabstopps, als wären sie Leerzeichen. |
| eoAlwaysVisibleCaret  | Verschiebt den Cursor so, dass er beim Blättern immer sichtbar ist.  |
| eoEnhanceEndKey       | Wenn Sie die Taste <Ende> drücken, wird das Ende der Zeile erreicht, jedoch ohne Berücksichtigung von Leerzeichen am Ende.                                   |
| eoFoldedCopyPaste     | Behält die Falteigenschaften bei Kopier-/Einfügevorgängen bei.   |
| eoPersistentBlock     | Behält Auswahlblöcke bei, auch wenn sich der Cursor außerhalb des Blocks befindet.   |
| eoOverwriteBlock      | Er verwaltet keine persistenten Blöcke. Er überschreibt sie bei Einfüge-/Löschvorgängen.   |
| eoAutoHideCursor      | Ermöglicht das Ausblenden des Cursors bei Tastaturoperationen.   |

## 2 Syntaxhervorhebung und Autovervollständigung mit SynEdit.

### 2.1 Einführung

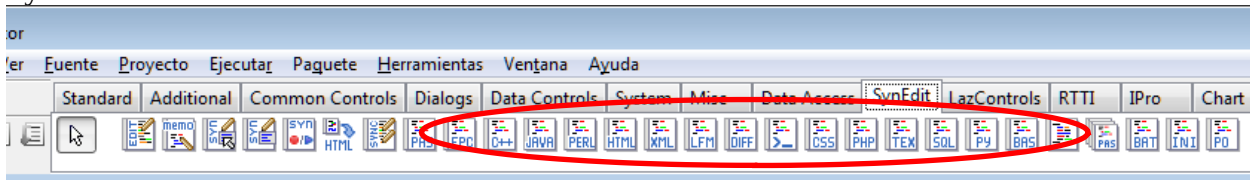
Die Syntaxhervorhebung ist die Möglichkeit, jedem Element eines Textes verschiedene visuelle Attribute zuzuweisen, um ihn besser lesbar zu machen. Diese Attribute werden nur im Editor angezeigt und sind nicht Teil des Textes, wenn er in einer Datei gespeichert wird.

Die folgende Abbildung zeigt ein Stück Code in Pascal, das in einem einfachen Editor und in einem anderen mit Syntaxhervorhebung angezeigt wird:



Der Unterschied ist offensichtlich. Code mit Syntaxhervorhebung ist leichter zu verstehen und zu lesen. Diese Hervorhebung muss jedoch konsequent und einfach erfolgen, da sie sonst eher verwirren als helfen kann.

Die Syntaxhervorhebung wird auch als "Syntaxfärbung" bezeichnet, da die Farbe häufig zur Unterscheidung von Textelementen verwendet wird, aber die Farbe ist nur eines der Attribute, die in einem allgemeineren Kontext verwendet werden können, zu dem beispielsweise auch die Schriftart gehört. Um die Verwendung diverser Attribute zu verallgemeinern, verwenden wir den Begriff "Syntaxhervorhebung" bzw. „Highlighter“, aber wir werden auch "Syntaxfärbung" als Synonym verwenden. Die TSynEdit-Komponente enthält keine eingebauten Highlighter im Steuerelement selbst, sondern muss mit einem externen Highlighter verbunden werden. Zu diesem Zweck enthält das SynEdit-Paket verschiedene Highlighter-Komponenten (aus verschiedenen Sprachen), die mit SynEdit verwendet werden können.



Wir können auch unseren eigenen Highlighter herstellen, wenn wir keinen finden, der unseren Bedürfnissen entspricht.

## 2.1.1 Wichtige Konzepte

Im Abschnitt - **Editor** haben wir gesehen, wie SynEdit die Datenbytes interpretiert, um auf die Ebene der Bildschirmzellen zu gelangen. Und dass im Allgemeinen ein Zeichen eine Zelle einnimmt.

Um eine Syntaxhervorhebung zu implementieren, müssen Sie die Ebene des Parsing erhöhen. Zeichen sind für die Hervorhebung nicht geeignet (es sei denn, Sie wollen eine zeichenweise Hervorhebung), es ist besser, das Konzept des "Tokens" zu verwenden.

Lassen Sie uns zunächst einige wichtige Begriffe wiederholen, bevor wir mit dem Thema fortfahren:

- **Zeichen** - Sie sind die minimalen visuellen Komponenten, aus denen ein Text besteht. Sie gehorchen einer definierten Kodierung (UTF-8 im Fall von SynEdit). Sie ermöglichen die Definition größerer Elemente wie z. B. Token. Sie nehmen normalerweise eine Zelle auf dem Bildschirm ein, können aber auch zwei Zellen einnehmen.
- **Token**: Eine Informationseinheit, die aus einem oder mehreren Zeichen besteht, die nach bestimmten Regeln gruppiert sind und die in einer bestimmten Syntax einen Sinn ergeben. Token sind die Teile, in die eine Programmiersprache auf lexikalischer Ebene unterteilt ist.
- **Attribut** - Dies sind die Hervorhebungseigenschaften, die für ein bestimmtes Token gelten. In SynEdit wird ein Attribut durch ein Objekt der Klasse "TSynHighlighterAttributes" dargestellt und ermöglicht es Ihnen, Eigenschaften wie Textfarbe, Hintergrundfarbe, Schriftart, Fettaktivierung und andere Eigenschaften zu definieren.

Die Syntaxhervorhebung beginnt auf Token-Ebene, und jedes Token kann andere Attribute als der Rest der Token haben. Man kann sagen, dass die minimale Informationseinheit für Syntaxhervorhebungen in SynEdit das Token ist.

Ein Token ist die minimale Informationseinheit für die Syntaxhervorhebung in der Komponente SynEdit.

Die folgende Abbildung soll dies verdeutlichen:

|         |        |         |        |        |    |        |         |    |    |        |        |    |    |    |    |        |        |             |    |           |    |    |    |    |    |    |    |
|---------|--------|---------|--------|--------|----|--------|---------|----|----|--------|--------|----|----|----|----|--------|--------|-------------|----|-----------|----|----|----|----|----|----|----|
| 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 |
| Zeichen | {      | x       | =      | 1      | 2  | ;      | c       | a  | d  | =      | "      | t  | ú  | "  | }  | ;      |        |             |    | /         | /  | c  | o  | m  | e  | n  | t  |
| Zellen  | {      | x       | =      | 1      | 2  | ;      | c       | a  | d  | =      | "      | t  | ú  | "  | }  | ;      |        |             |    | /         | /  | c  | o  | m  | e  | n  | t  |
| tokens  | Symbol | Kennung | Symbol | Nummer |    | Symbol | Kennung |    |    | Symbol | string |    |    |    |    | Symbol | Symbol | Leerzeichen |    | Kommentar |    |    |    |    |    |    |    |
|         |        |         |        |        |    |        |         |    |    |        |        |    |    |    |    |        |        |             |    |           |    |    |    |    |    |    |    |
| Blöcke  |        |         |        |        |    |        |         |    |    |        |        |    |    |    |    |        |        |             |    |           |    |    |    |    |    |    |    |

Wie Sie sehen, können Token verschiedener Art sein, und die Attribute müssen auf jede Art von Token angewendet werden (nicht auf jedes Token). Daher müssen alle Zeichenketten im gleichen Text die gleichen Attribute haben.

Gängige Arten von Token sind:

- Identifier/Bezeichner - Einige von ihnen können Schlüsselwörter sein.
- Nummern - Sie können eine unterschiedliche Darstellung haben.
- Leerzeichen - Alle Zeichen, die nicht sichtbar sind
- Strings/Zeichenkette - In der Regel in einfache oder doppelte Anführungszeichen eingeschlossen.
- Kommentare - Es können eine oder mehrere Zeilen sein.
- Symbole - Verschiedene Symbole als Operatoren.
- Steuerzeichen - Kann Zeilenumbrüche oder Endmarkierungen enthalten.

Die Art und Weise, wie Zeichen in Token (und deren Typen) gruppiert werden, wird von SynEdit nicht vordefiniert. Es gibt keine festgelegten Regeln. Wenn SynEdit eine Syntaxhervorhebung durchführt, verwendet es einen Highlighter, und dieser Highlighter definiert die Art und Weise, wie Token im Text identifiziert werden.

Die vordefinierten Highlighter enthalten diese Definitionen bereits, aber wenn wir einen benutzerdefinierten Highlighter erstellen, müssen wir diese Regeln selbst definieren.

Vereinfacht könnte man sagen, dass der Prozess der Syntaxhervorhebung darin besteht, den Text in Token aufzuteilen und jedem Token eine Farbe (ein Attribut) zu geben. Diese vereinfachte Beschreibung ist zwar sehr zutreffend, aber der Prozess ist komplizierter als er klingt. Die größte Schwierigkeit besteht darin, die einzelnen Token genau und relativ schnell zu identifizieren. Token sind die grundlegenden Elemente, mit denen ein lexikalischer Analysator (Lexer) arbeiten würde. Ein Syntax-Highlighter ist in gewisser Weise ein lexikalischer Analysator, aber sein Ziel ist es nicht, als Grundlage für eine nachfolgende syntaktische oder semantische Analyse zu dienen, sondern die Token und ihre Typen zu identifizieren, um die Hervorhebung zu erleichtern<sup>6</sup>.

---

<sup>6</sup> Die Möglichkeit, einen Textmarker in einen Syntax-Analysator umzuwandeln, bleibt der Freiheit des Programmierers überlassen. Die vordefinierten Highlighter in Lazarus identifizieren nur Token und in einigen Fällen Blockbegrenzer wie BEGIN ..END (für Folding), sind aber nicht dazu gedacht, als Lexer zu dienen. Dies würde mehr Verarbeitung erfordern und zu einer langsameren Analyse führen, was den Zweck eines Textmarkers verfehlt.

## 2.2 Syntaxfärbung mit vordefinierten Komponenten

Bei dieser Art der Syntaxfärbung werden vordefinierte Syntaxkomponenten verwendet. Diese Komponenten sind für die Arbeit in einer vordefinierten Sprache vorbereitet.

### 2.2.1 Verwendung einer vordefinierten Sprache

Diese Syntaxkomponenten werden bereits in der Lazarus Umgebung installiert. Es gibt Komponenten für die meisten gängigen Sprachen wie: Pascal, C++, Java, HTML, Python, etc. Die Methode ist einfach: Wir ziehen das Steuerelement auf das Formular, in dem sich das SynEdit-Steuerelement befindet, das wir verwenden wollen. Dann verknüpfen wir es mit der Eigenschaft "Highlighter" von SynEdit.

Später können wir die Farben und Attribute der Schlüsselwörter, Kommentare, Zahlen und anderer Kategorien auswählen, indem wir auf die Eigenschaften des Objekts "TSynXXSYn" zugreifen (wobei XXX für die gewählte Sprache steht). Jedes "TSynXXSYn"-Kontrollelement wird durch eine Unit dargestellt.

Das "TSynXXSYn"-Objekt ist für die Arbeit in seiner vordefinierten Sprache optimiert und reagiert gut auf die Geschwindigkeit. Die Schlüsselwörter und ihre Erkennung sind jedoch fest in den Code Ihres Geräts ein codiert. Der Versuch, ein weiteres reserviertes Wort hinzuzufügen, ist aufgrund der Art und Weise, wie die Schlüsselwortsuche optimiert ist, nicht so einfach.

Die Verwendung von vordefinierten Komponenten erspart uns die Arbeit, eine komplette Syntax einer bekannten Sprache zu verarbeiten.

### 2.2.2 Verwendung einer benutzerdefinierten Sprache

Zusätzlich zu den vordefinierten Sprachkomponenten gibt es eine Komponente, die für eine andere Sprache angepasst werden kann. Dies ist die Komponente "TSynAnySyn".

Um diese Komponente zu verwenden, fügen Sie sie einfach in das Formular ein, wie jede andere Syntaxkomponente. Bevor Sie sie verwenden, müssen Sie jedoch wissen, aus welchen Schlüsselwörtern ihre Syntax besteht.

Obwohl diese Komponente die Definition mehrerer einfacher Sprachen unterstützt, bietet sie keine große Flexibilität bei der Handhabung von Kommentaren.

Außerdem ist diese Komponente nicht sehr schnell, da sie aufgrund ihrer Mehrsprachigkeit nicht richtig optimiert werden kann. Es wird nicht empfohlen, diese Komponente bei Sprachen mit vielen Schlüsselwörtern oder bei umfangreichen Texten zu verwenden.

Wenn Sie eine neue Sprache mit guter Leistung und hoher Personalisierung implementieren wollen, ist es ratsam, dies per Code zu tun.

## 2.3 Syntaxfärbung mit Code

Diese Art der Einfärbung ermöglicht es Ihnen, eine bestimmte Syntax effizient anzupassen. Sie sollte verwendet werden, wenn die zu verwendende Sprache nicht bereits in einer der vordefinierten Komponenten vorhanden ist oder nicht mit dem erforderlichen Verhalten übereinstimmt.

Bevor wir unsere Klasse für die Einfärbung entwerfen, müssen wir ein wenig darüber wissen, wie die Syntaxfärbung funktioniert:

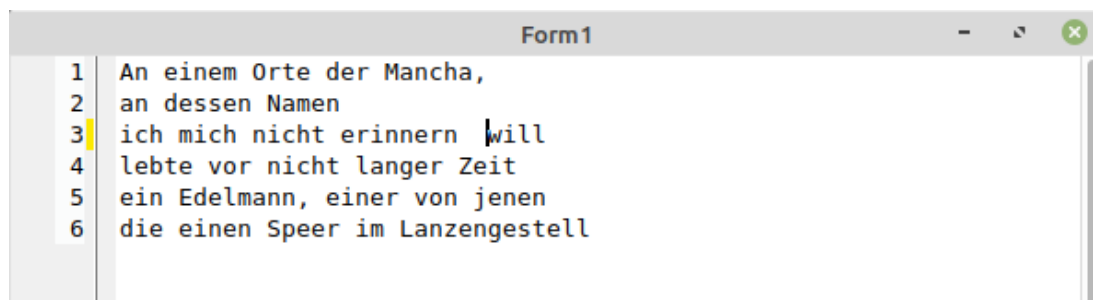
Für die Syntaxfärbung muss eine von "TSynCustomHighlighter" abstammende Klasse erstellt werden, die wir Highlighter oder Syntaxklasse nennen und von der wir einen Highlighter instanziiieren werden. Dieser Highlighter muss dann mit dem SynEdit-Editor verknüpft werden, der die Syntaxfärbung implementieren wird.

Wenn ein Highlighter mit einem "TSynEdit"-Editor verbunden ist, ruft dieser die Highlighter-Routinen immer dann auf, wenn er Informationen zur Syntaxfärbung benötigt. TSynEdit speichert keine Informationen zur Textfärbung irgendwo. Wann immer Sie Farbinformationen benötigen, ruft es den Highlighter auf, um die Farbinformationen "on the fly" zu erhalten.

Dieser Highlighter muss die Textanalyse und die Identifizierung von Textelementen implementieren, um Textattributinformationen an SynEdit weiterzugeben.

Die Erkundung des zu färbenden Textes erfolgt durch SynEdit, wobei Zeile für Zeile bearbeitet wird. Wenn eine Zeile geändert wird, werden die geänderte Zeile und die nachfolgenden Zeilen, beginnend mit der geänderten Zeile, im sichtbaren Fenster angezeigt. Dieses Verhalten ist normal, wenn man bedenkt, dass ein Syntaxelement, das sich auf die anderen Zeilen des Textes auswirkt, in einer Zeile enthalten sein kann (z. B. der Anfang eines mehrzeiligen Kommentars).

Betrachten wir den Fall eines Editors mit dem folgenden Text:



Wenn das Fenster eingeblendet wird, nachdem es ausgeblendet wurde, werden die folgenden Ereignisse erzeugt:

- SetLine: An einem Orte der Mancha,
- SetLine: an dessen Namen
- SetLine: ich mich nicht erinnern will
- SetLine: lebte vor nicht langer Zeit
- SetLine: ein Edelmann, einer von jenen
- SetLine: die einen Speer im Lanzengestell

Die Bezeichnung "SetLine" zeigt an, dass die angezeigte Zeile gescannt wird. Wenn Sie die Zeile Nummer 3 desselben Textes (an beliebiger Stelle) ändern, ändert sich die Abtastreihenfolge leicht:

- SetLine: ich mich nicht erinnern will
- SetLine: lebte vor nicht langer Zeit
- SetLine: ein Edelmann, einer von jenen
- SetLine: an dessen Namen
- SetLine: ich mich nicht erinnern will
- SetLine: lebte vor nicht langer Zeit
- SetLine: ein Edelmann, einer von jenen

Wir können sehen, dass der Editor die nächsten beiden Zeilen durchsucht und dann erneut durchsucht, aber mit einer Zeile vorher beginnt.

### **2.3.1 Fälle von Syntaxhervorhebung.**

Bei der Syntaxhervorhebung müssen wir verschiedene Situationen berücksichtigen. Es können 3 Fälle unterschieden werden:

- Einfache Token-Hervorhebung. Hierbei handelt es sich um die übliche Einfärbung, bei der bestimmte Wörter oder Schlüsselbezeichner im Text in eine bestimmte Farbe umgewandelt werden. Normalerweise werden verschiedene Kategorien definiert, wie reservierte Wörter, Schlüsselwörter, Bezeichner, Variablen, Makros usw. Für jede Kategorie können Sie verschiedene Textattribute definieren. Token werden durch die Zeichen identifiziert, die sie enthalten können.
- Hervorhebung von einzeiligen Kommentaren. Diese Einfärbung ist typisch für einzeilige Kommentare in den meisten Sprachen. Dabei wird der Text eines Kommentars vom Anfang bis zum Ende der Zeile mit einer bestimmten Farbe versehen.
- Hervorhebung von Text- oder Kontextbereichen. Diese Färbung wird auch auf Kommentare oder Ketten angewendet. Die Einfärbung des Textes betrifft eine Gruppe von Wörtern, die in derselben Zeile stehen oder mehrere aufeinander folgende Zeilen einnehmen können. Token werden durch ihre Begrenzungszeichen identifiziert.



### 2.3.2 Zeilenabtastung

Es wird davon ausgegangen, dass jede Zeile in Elemente unterteilt ist, die "Token" genannt werden. Es gibt keinen Teil einer Zeile, der nicht ein Token ist. Ein Token kann ein Bezeichner, ein Symbol, ein Steuerzeichen, ein Leerzeichen usw. sein.

Ein Token kann ein oder mehrere Zeichen lang sein. Jedes Token bzw. jede Art von Token kann bestimmte Attribute haben.

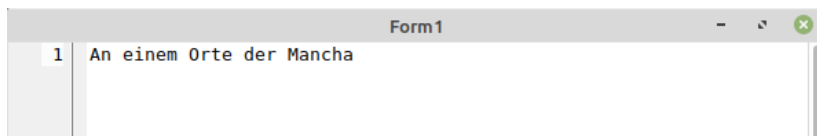
Jedes Mal, wenn SynEdit eine Zeile verarbeiten muss, macht es eine Reihe von Aufrufen an den Highlighter (eigentlich "TSynCustomHighlighter", aber es leitet sie an den verwendeten Highlighter weiter), um die Attribute der einzelnen Elemente in der Zeile zu erhalten:

- Zunächst wird die Methode "SetLine" aufgerufen, um anzuzeigen, dass die Erkundung einer neuen Zeile beginnt. Nach diesem Aufruf sollte die Methode "GetTokenEx" oder "GetTokenAttribute" Informationen über das aktuelle Token zurückgeben.
- Nach dem Aufruf von "SetLine" werden mehrere Aufrufe der Methode "Next" erzeugt, um auf die folgenden Token zuzugreifen.
- SynEdit erwartet, dass nach jedem Aufruf von "Next" die Methoden "GetTokenEx" und "GetTokenAttribute" Informationen über das aktuell angezeigte Token zurückgeben.
- Wenn SynEdit prüfen will, ob das Ende der Arbeitszeile erreicht ist, wird es einen Aufruf von "GetEol" machen. Dies muss funktionieren, da "SetLine" aufgerufen wird.

Diese Methoden werden wiederholt und in großer Zahl für jede Zeile aufgerufen. Daher müssen diese Methoden schnell reagieren und effizient implementiert sein. Eine Verzögerung bei der Verarbeitung einer dieser Methoden beeinträchtigt die Leistung des Editors.

Um eine Vorstellung von der Arbeit zu bekommen, die SynEdit im Hinblick auf die Syntaxfärbung leistet, stellen wir im Folgenden die Abfolge der Methoden dar, die aufgerufen werden, wenn das ausgeblendete Editorfenster angezeigt wird.

Das Beispielfenster enthält diesen Text:

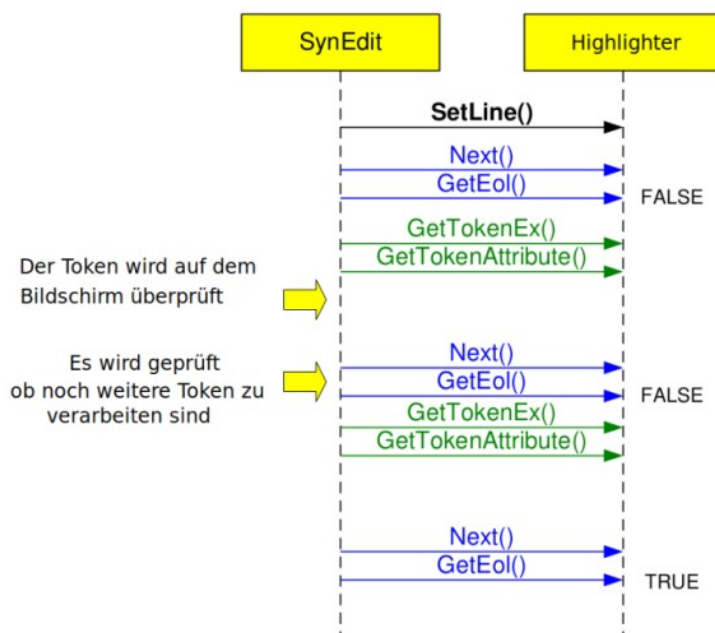


Die Reihenfolge der Ereignisse, die bei der Anzeige dieses Fensters aufgerufen werden, ist wie folgt:

1. SetLine, Zeilennummer: 0, An einem Orte der Mancha
2. Next, TokenPosition: 24
3. GetToken
4. GetEol, Result: false
5. GetTokenEx
6. GetTokenAttribute
7. Next, TokenPosition: 0
8. GetEol, Result: false
9. GetTokenEx
10. GetTokenAttribute
11. Next, TokenPosition: 2
12. GetToken
13. GetEol, Result: false
14. GetTokenEx
15. GetTokenAttribute
16. Next, TokenPosition: 3
17. GetEol, Result: false
18. GetTokenEx
19. GetTokenAttribute
20. Next, TokenPosition: 8
21. GetToken
22. GetEol, Result: false
23. GetTokenEx
24. GetTokenAttribute
25. Next, TokenPosition: 9
26. GetEol, Result: false
27. GetTokenEx
28. GetTokenAttribute
29. Next, TokenPosition: 13
30. GetToken
31. GetEol, Result: false
32. GetTokenEx
33. GetTokenAttribute
34. Next, TokenPosition: 14
35. GetEol, Result: false
36. GetTokenEx
37. GetTokenAttribute
38. Next, TokenPosition: 17
39. GetToken
40. GetEol, Result: false
41. GetTokenEx
42. GetTokenAttribute
43. Next, TokenPosition: 18
44. GetEol, Result: true

Der nach dem "Next"-Ereignis angezeigte Wert entspricht dem ersten Zeichen, das gescannt wird. Es konnte beobachtet werden, dass der Editor nach jedem Aufruf von "Next" immer prüft, ob das Ende erreicht wurde. Wenn der Editor nach einem Aufruf von Next() in GetEol() TRUE zurückgibt, geht er davon aus, dass er sich am letzten Zeichen der Zeile befindet, und fragt nicht mehr nach Attributinformationen.

Das folgende Ablaufdiagramm verdeutlicht den Prozess:



Der Editor zeichnet auf dem Bildschirm, immer Token für Token. Die Methode GetTokenEx() gibt die Token-Erweiterung zurück, und die Methode GetTokenAttribute() gibt das Attribut zurück, das auf den Text anzuwenden ist. Diese Informationen sind alles, was SynEdit braucht, um einen Teil des Textes mit Attributen auf dem Bildschirm zu zeichnen.

Diese Sequenz, Next() - GetEol() - GetTokenEx() - GetTokenAttribute() , wird in der gesamten Zeile wiederholt, bis GetEol() TRUE zurückgibt, was bedeutet, dass es in dieser Zeile keine weiteren Token mehr zu untersuchen gibt.


Wenn wir eine Änderung in unserem Beispieltext vornehmen, z. B. ein Komma am Ende der Zeile einfügen, wird die folgende Sequenz erzeugt:

1. SetLine ,Zeilennummer: 0, An einem Orte der Mancha,
2. Next, TokenPosition: 24
3. GetToken 2
4. GetEol, Result: false
5. Next, TokenPosition: 0
6. GetEol, Result: false
7. Next, TokenPosition: 2
8. GetToken 8
9. GetEol, Result: false
10. Next, TokenPosition: 3
11. GetEol, Result: false
12. Next, TokenPosition: 8
13. GetToken 13
14. GetEol, Result: false
15. Next, TokenPosition: 9
16. GetEol, Result: false
17. Next, TokenPosition: 13
18. GetToken 17
19. GetEol, Result: false
20. Next, TokenPosition: 14
21. GetEol, Result: false
22. Next, TokenPosition: 17
23. GetToken 24
24. GetEol, Result: false
25. Next, TokenPosition: 18
26. GetEol, Result: false
27. Next, TokenPosition: 24
28. GetEol, Result: true
29. SetLine ,Zeilennummer: 0, An einem Orte der Mancha,
30. Next, TokenPosition: 25
31. GetToken 2
32. GetEol, Result: false
33. GetTokenEx 2
34. GetTokenAttribute
35. Next, TokenPosition: 0
36. GetEol, Result: false
37. GetTokenEx 3
38. GetTokenAttribute
39. Next, TokenPosition: 2
40. GetToken 8
41. GetEol, Result: false
42. GetTokenEx 8
43. GetTokenAttribute
44. Next, TokenPosition: 3
45. GetEol, Result: false

```
46. GetTokenEx 9
47. GetTokenAttribute
48. Next, TokenPosition: 8
49. GetToken 13
50. GetEol, Result: false
51. GetTokenEx 13
52. GetTokenAttribute
53. Next, TokenPosition: 9
54. GetEol, Result: false
55. GetTokenEx 14
56. GetTokenAttribute
57. Next, TokenPosition: 13
58. GetToken 17
59. GetEol, Result: false
60. GetTokenEx 17
61. GetTokenAttribute
62. Next, TokenPosition: 14
63. GetEol, Result: false
64. GetTokenEx 18
65. GetTokenAttribute
66. Next, TokenPosition: 17
67. GetToken 24
68. GetEol, Result: false
69. GetTokenEx 24
70. GetTokenAttribute
71. Next, TokenPosition: 18
72. GetEol, Result: false
73. GetTokenEx 25
74. GetTokenAttribute
75. Next, TokenPosition: 24
76. GetEol, Result: true
```

Es ist zu beachten, dass der Editor zunächst eine Vorabprüfung der gesamten Zeile durchführt, bevor er die Attribute anwendet.

Zusätzlich zur Syntaxfärbung führt SynEdit eine eigene unabhängige Überprüfung durch, um "Klammern" (Klammern, eckige Klammern, geschweifte Klammern und Anführungszeichen) zu erkennen und hervorzuheben, wenn der **Cursor** auf eines dieser Elemente zeigt:



SynEdit" erlaubt es dem Highlighter jedoch, bei der Identifizierung dieser Begrenzungszeichen mitzuwirken. Warum? Weil der Highlighter zusätzliche Informationen für die Hervorhebung der Klammern bereitstellen kann, da er die Attribute der verschiedenen Teile des Textes verarbeitet.

Um die "Klammern"-Funktionalität von SynEdit zu nutzen, muss der Highlighter die Methoden korrekt implementieren: "GetToken", und "GetTokenPos" und "GetTokenKind". Wie funktionieren sie? Damit eine öffnende "Klammer" mit der entsprechenden schließenden "Klammer" verknüpft werden kann, wird überprüft, ob "tokenKind" für beide den gleichen Wert liefert. Wird bei der Überprüfung eine "Klammer" mit einem anderen Attribut gefunden, so wird dies nicht berücksichtigt.

Diese Methoden werden zwar nicht für die Syntaxfärbung verwendet, bestimmen aber das Verhalten der Klammerhervorhebung.

Diese Methoden werden weniger häufig aufgerufen als die Methoden zur Syntaxfärbung. Sie werden nur ausgeführt, wenn der Cursor auf eine "Klammer" zeigt oder wenn eine hinzugefügt oder entfernt wird.

Wenn Sie den Prozess der Syntaxeinfärbung verstanden haben, sind wir nun bereit, die ersten Schritte zur Implementierung eines Code-Highlighters zu unternehmen.

### **2.3.3 Erste Schritte**

Zunächst einmal ist es ratsam, eine spezielle Unit zu erstellen, in der der Code unseres neuen Highlighters gespeichert wird.

Für dieses Beispiel werden wir eine Unit mit dem Namen "uSyntax" erstellen und die für die Erstellung der zu verwendenden Objekte erforderlichen Units einbeziehen.

In dieser neuen Unit müssen wir unbedingt eine von "TSynCustomHighlighter" (definiert in der Einheit "SynEditHighlighter") abgeleitete Klasse für die Erstellung unseres Highlighters erstellen.

Dies ist wahrscheinlich die einfachste Klasse, die für die Syntaxhervorhebung implementiert werden kann. Allerdings wird diese Klasse keinen Text hervorheben, da sie keine Anweisungen zur Änderung der Textattribute enthält. Sie beschränkt sich darauf, Standardwerte an "SynEdit"-Anfragen zurückzugeben. Sie hat keinen Nutzen, sie ist lediglich ein minimalistisches Demonstrationsbeispiel.

```
{
Minimalunit zur Demonstration der Struktur einer einfachen Klasse, die für die
Syntaxhervorhebung verwendet wird.
Sie ist nicht funktional, sondern dient nur zur Veranschaulichung.
Erstellt von Tito Hinostroza: 04/08/2013
}
unit uSintax;

{$mode objfpc}{$H+}

interface

uses
Classes, SysUtils, Graphics, SynEditHighlighter;
type
{Klasse für die Erstellung eines Highlighters}
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);
//Konstruktor der Klasse. Hier müssen die zu verwendenden Attribute angelegt
werden.
begin
  inherited Create(AOwner);
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
{Sie wird vom Editor aufgerufen, wenn er die Informationen über die
Färbung einer Zeile aktualisieren muss. Nach dem Aufruf dieser Funktion wird
erwartet, dass GetTokenEx, das aktuelle Token zurückgibt. Und auch nach jedem
Aufruf von "Next".}
begin
  inherited;
  linAct := NewValue;
//Kopieren der aktuellen Zeile
  posFin := 1;
//zeigt auf das erste Zeichen
  Next;
end;
```

```
procedure TSynMiColor.Next;
{Sie wird von SynEdit aufgerufen, um auf das nächste Token zuzugreifen. Und wird
von jedem Token in der aktuellen Zeile ausgeführt. In diesem Beispiel wird es
immer um ein Zeichen verschoben.}
begin
  posIni := posFin;
  //verweist auf das folgende Token
  If posIni > length(linAct) then //Ende der Zeile erreicht?
    exit
  //gehe zu
  else
    inc(posFin); //um ein Zeichen verschieben
end;

function TSynMiColor.GetEol: Boolean;
{Zeigt an, wenn das Ende der Zeile erreicht ist.}
begin
  Result := posIni > length(linAct);
end;

procedure TSynMiColor.GetTokenEx(out TokenStart:PChar;out TokenLength:integer);
{Gibt Informationen über den aktuellen Token zurück}
begin
  TokenStart := @linAct[posIni];
  TokenLength := posFin - posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
{Gibt Informationen über den aktuellen Token zurück}
begin
  Result := nil;
end;

{Die folgenden Funktionen werden von SynEdit verwendet,
um geschweifte Klammern, eckige Klammern, Klammern und
Anführungszeichen zu verarbeiten. Sie sind für die Token-Färbung
nicht entscheidend, sollten aber gut reagieren.}
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.
```



Die Methoden, die als "override" erscheinen, sind diejenigen, die implementiert werden müssen, um unserem Highlighter die Färbefunktionalität zu geben.

Bei jedem Aufruf von "SetLine" wird eine Kopie der Zeichenkette in "linAct" gespeichert, und diese Zeichenkette wird dann zur Extraktion der Token verwendet.

Bei jeder "Next"-Anfrage gibt diese Einheit nur das nächste in der Zeile gefundene Zeichen zurück, und das von "GetTokenAttribute" zurückgegebene Attribut ist immer NIL, was bedeutet, dass es keine Attribute gibt.

Auch die Methoden "GetToken", "GetTokenPos" und "GetTokenKind" geben keine signifikanten Werte zurück, sondern die entsprechenden Nullwerte.

Die Highlighter-Klasse, die wir erstellt haben, heißt "TSynMiColor". Es ist nicht möglich, die Klasse "TSynCustomHighlighter" als Highlighter zu verwenden, da diese Klasse abstrakt ist und nur dazu dient, die Anforderungen von TSynEdit bei der Syntaxfärbung richtig zu kanalisieren.

Um die neue Syntax zu verwenden, müssen wir ein Objekt erstellen und es mit der TSynEdit-Komponente verknüpfen, die wir verwenden wollen. Wenn wir unser Hauptformular in Unit1 haben und unser TSynEdit-Objekt "editor" heißt, könnte der Code für die Verwendung dieser Syntax so aussehen:

```
unit Unit1;
{$mode objfpc}{$H+}
interface
uses ... uSyntax;
...
var
Syntaxis : TSynMiColor;
...
procedure TForm1.FormCreate(Sender: TObject);
...
    Syntaxis := TSynMiColor.Create(Self); //Highlighter erstellen
    editor.Highlighter := Syntaxis; //weist die Syntax dem Editor zu
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
    editor.Highlighter := nil; //Highlighter-Entfernen
    Syntaxis.Destroy; //Objekt freigeben
end;
```

Das Verständnis der grundlegenden Funktionsweise dieses Arbeitsschemas ist der erste Schritt zur Erstellung funktionaler Highlighter. Wenn Sie die Funktionsweise dieses einfachen Beispiels nicht verstehen, empfehle ich Ihnen, den Code zu überprüfen, bevor Sie mit den folgenden Abschnitten fortfahren.

### 2.3.4 Hinzufügen von Funktionen zur Syntax

Das vorstehende Beispiel wurde nur zu Lehrzwecken erstellt. Die gewünschte Funktionalität ist nicht erfüllt, aber die Struktur, die alle Arten von Syntaxfärbung haben sollten, wird gezeigt. Zu Beginn müssen wir bedenken, dass die zu implementierenden Methoden schnell ausgeführt werden müssen. Sie sollten nicht mit viel Verarbeitung geladen werden, weil sie wiederholt für jede geänderte Zeile des Editors aufgerufen werden, so dass sie keine Verzögerungen erlauben, sonst wird der Editor schwer und langsam.

Die erste Änderung, die wir vornehmen müssen, ist die Methode zur Speicherung von Zeichenketten. Wenn Sie "SetLine" aufrufen, müssen Sie Informationen über die Zeichenkette haben. Aber die übergeordnete Klasse "TSynCustomHighlighter" speichert bereits eine Kopie der Zeichenkette, bevor sie "SetLine" aufruft.

Daher ist es nicht effizient, eine neue Kopie für uns zu erstellen. Es reicht aus, einen Verweis, einen Zeiger auf diese Zeichenkette, in "TSynCustomHighlighter" zu speichern.

Dazu muss die Variable "linAct" so geändert werden, dass sie ein "PChar" anstelle eines Strings ist. Dies wird in der Klassendefinition vorgenommen. Die Methoden "SetLine", "Next", "GetEol", "GetTokenEx" und "GetTokenAttribute" müssen ebenfalls geändert werden:

Das Gerüst unserer Unit würde wie folgt aussehen:

```
{Minimaleinheit zur Demonstration der Struktur einer einfachen Klasse, die für  
die Syntaxhervorhebung verwendet wird.  
Sie ist nicht funktional, sondern dient nur zur Veranschaulichung.  
Erstellt von Tito Hinostroza: 04/08/2013}
```

```
unit uSyntax;  
{$mode objfpc}{$H+}  
interface  
uses  
Classes, SysUtils, Graphics, SynEditHighlighter;  
  
type  
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);
begin
  inherited Create(AOwner);
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
{Sie wird vom Editor aufgerufen, wenn er die Informationen über die
Färbung einer Zeile aktualisieren muss. Nach dem Aufruf dieser Funktion wird
erwartet, dass GetTokenEx, das aktuelle Token zurückgibt. Und auch nach jedem
Aufruf von "Next".}
begin
  inherited;
  linAct := PChar(NewValue); //Kopieren der aktuellen Zeile als Nullterminierten
String
  posFin := 0; //zeigt auf das erste Zeichen, PChar beginnt bei 0!
  Next;
end;

procedure TSynMiColor.Next;
{Sie wird von SynEdit aufgerufen, um auf das nächste Token zuzugreifen. Und wird
von jedem Token in der aktuellen Zeile ausgeführt. In diesem Beispiel wird es
immer um ein Zeichen geschoben.}
begin
  posIni := posFin; //verweist auf das folgende Token
  if linAct[posIni] = #0 then exit; //das Ende der Zeile erreicht?
  inc(posFin); //ein Zeichen verschieben
end;

function TSynMiColor.GetEol: Boolean;
{Zeigt an, wenn das Ende der Zeile erreicht ist.}
begin
  Result := linAct[posIni] = #0;
end;

procedure TSynMiColor.GetTokenEx(out TokenStart:PChar;out TokenLength :integer);
{Gibt Informationen über den aktuellen Token zurück}
begin
  TokenLength := posFin - posIni;
  TokenStart := linAct + posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
{Gibt Informationen über den aktuellen Token zurück}
begin
  Result := nil;
end;
```

{Die folgenden Funktionen werden von SynEdit verwendet, um Klammern, geschweifte Klammern und Anführungszeichen zu behandeln. Sie sind nicht entscheidend für Token, aber sie sollten gut reagieren.}

```
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.
```

Jetzt sehen wir, dass wir "posFin" bei Null beginnen lassen müssen, wo die Zeichen-Kette nun beginnt, in "linAct".

Aber auch diese Klasse ist leer an Attributen. Als erstes sollten wir unsere Attribute erstellen. Diese müssen als Eigenschaften des Objekts "TSynMiColor" deklariert werden:

```
fAtriComent      : TSynHighlighterAttributes ;//Kommentar
fAtriIdent       : TSynHighlighterAttributes ;//Identifizier(Bezeichner)
fAtriClave       : TSynHighlighterAttributes ;//Schlüsselwort
fAtriNumero      : TSynHighlighterAttributes ;//Nummern
fAtriEspac       : TSynHighlighterAttributes ;//Leerzeichen
fAtriCadena      : TSynHighlighterAttributes ;//Strings
```

Alle Attribute sind vom Typ "TSynHighlighterAttributes". Diese Klasse enthält die verschiedenen Attribute, die mit einem Token verknüpft werden können, wie z. B. Textfarbe, Hintergrundfarbe, Rahmenfarbe usw.

Dann müssen wir im Konstruktor die Eigenschaften dieser Attribute erstellen und definieren:

```
constructor TSynMiColor.Create(AOwner: TComponent);
//Konstruktor der Klasse. Hier müssen die zu verwendenden Attribute angelegt
werden.
begin
  inherited Create(AOwner);
  //Kommentar-Attribut
  fAtriComent := TSynHighlighterAttributes.Create('Comment');
  fAtriComent.Style := [fsItalic]; //Kursivschrift
  fAtriComent.Foreground := clGray;
  AddAttribute(fAtriComent);
  //Schlüsselwort-Attribut
  fAtriClave := TSynHighlighterAttributes.Create('Key');
  fAtriClave.Style := [fsBold]; //fett gedruckt
  fAtriClave.Foreground:=clGreen; //grüne Schriftfarbe
  AddAttribute(fAtriClave);
```

```
//Nummern-Attribut
fAtriNumero := TSynHighlighterAttributes.Create('Number');
fAtriNumero.Foreground := clFuchsia;
AddAttribute(fAtriNumero);
//Leerzeichen-Attribut. Keine Attribute
fAtriEspac := TSynHighlighterAttributes.Create('space');
AddAttribute(fAtriEspac);
//String-Attribut
fAtriCadena := TSynHighlighterAttributes.Create('String');
fAtriCadena.Foreground := clBlue; //blaue Schriftfarbe
AddAttribute(fAtriCadena);
end;
```

Beachten Sie, dass die Konstanten `fsBold`, `fsItalic`, ... in der Einheit "Grafik" definiert sind.

Es wurden Attribute für verschiedene Kategorien von Token definiert. Hier legen Sie fest, wie der Text des Tokens aussehen soll.

Um ein Attribut zu erstellen, empfiehlt es sich, die vordefinierten Konstanten ([Resourcestrings](#)) in der Unit "SynEditStrConst" zu verwenden :

|                          |                       |
|--------------------------|-----------------------|
| SYNS_AttrASP             | = 'Asp';              |
| SYNS_AttrCDATA           | = 'CDATA';            |
| SYNS_AttrDOCTYPE         | = 'DOCTYPE';          |
| SYNS_AttrAnnotation      | = 'Annotation';       |
| SYNS_AttrAssembler       | = 'Assembler';        |
| SYNS_AttrAttributeName   | = 'Attribute Name';   |
| SYNS_AttrAttributeValue  | = 'Attribute Value';  |
| SYNS_AttrBlock           | = 'Block';            |
| SYNS_AttrBrackets        | = 'Brackets';         |
| SYNS_AttrCDATASection    | = 'CDATA Section';    |
| SYNS_AttrCharacter       | = 'Character';        |
| SYNS_AttrClass           | = 'Class';            |
| SYNS_AttrComment         | = 'Comment';          |
| SYNS_AttrIDEDirective    | = 'IDE Directive';    |
| SYNS_AttrCondition       | = 'Condition';        |
| SYNS_AttrDataType        | = 'Data type';        |
| SYNS_AttrDefaultPackage  | = 'Default packages'; |
| SYNS_AttrDir             | = 'Direction';        |
| SYNS_AttrDirective       | = 'Directive';        |
| SYNS_AttrDOCTYPESection  | = 'DOCTYPE Section';  |
| SYNS_AttrDocumentation   | = 'Documentation';    |
| SYNS_AttrElementName     | = 'Element Name';     |
| SYNS_AttrEmbedSQL        | = 'Embedded SQL';     |
| SYNS_AttrEmbedText       | = 'Embedded text';    |
| SYNS_AttrEntityReference | = 'Entity Reference'; |
| SYNS_AttrEscapeAmpersand | = 'Escape ampersand'; |
| SYNS_AttrEvent           | = 'Event';            |
| SYNS_AttrException       | = 'Exception';        |
| SYNS_AttrFloat           | = 'Float';            |
| SYNS_AttrForm            | = 'Form';             |
| SYNS_AttrFunction        | = 'Function';         |
| SYNS_AttrHexadecimal     | = 'Hexadecimal';      |
| SYNS_AttrIcon            | = 'Icon reference';   |
| SYNS_AttrIdentifier      | = 'Identifier';       |

```

SYNS_AttrIllegalChar      = 'Illegal char';
SYNS_AttrInclude          = 'Include';
SYNS_AttrIndirect         = 'Indirect';
SYNS_AttrInvalidSymbol   = 'Invalid symbol';
SYNS_AttrInternalFunction = 'Internal function';
SYNS_AttrKey              = 'Key';
SYNS_AttrLabel            = 'Label';
SYNS_AttrMacro            = 'Macro';
SYNS_AttrMarker           = 'Marker';
SYNS_AttrMessage          = 'Message';
SYNS_AttrMiscellaneous    = 'Miscellaneous';
SYNS_AttrNamespaceAttrName = 'Namespace Attribute Name';
SYNS_AttrNamespaceAttrValue = 'Namespace Attribute Value';
SYNS_AttrNonReservedKeyword = 'Non-reserved keyword';
SYNS_AttrNull             = 'Null';
SYNS_AttrNumber           = 'Number';
SYNS_AttrOctal            = 'Octal';
SYNS_AttrOperator         = 'Operator';
SYNS_AttrPLSQL            = 'Reserved word (PL/SQL)';
SYNS_AttrPragma           = 'Pragma';
SYNS_AttrPreprocessor      = 'Preprocessor';
SYNS_AttrProcessingInstr   = 'Processing Instruction';
SYNS_AttrQualifier        = 'Qualifier';
SYNS_AttrRegister         = 'Register';
SYNS_AttrReservedWord     = 'Reserved word';
SYNS_AttrRpl              = 'Rpl';
SYNS_AttrRplKey           = 'Rpl key';
SYNS_AttrRplComment       = 'Rpl comment';
SYNS_AttrSASM             = 'SASM';
SYNS_AttrSASMComment      = 'SASM Comment';
SYNS_AttrSASMKey          = 'SASM Key';
SYNS_AttrSecondReservedWord = 'Second reserved word';
SYNS_AttrSection          = 'Section';
SYNS_AttrSpace            = 'Space';
SYNS_AttrSpecialVariable   = 'Special variable';
SYNS_AttrSQLKey           = 'SQL keyword';
SYNS_AttrSQLPlus          = 'SQL*Plus command';
SYNS_AttrString           = 'String';
SYNS_AttrSymbol           = 'Symbol';
SYNS_AttrProcedureHeaderName = 'Procedure header name';
SYNS_AttrCaseLabel        = 'Case label';
SYNS_AttrSyntaxError      = 'SyntaxError';
SYNS_AttrSystem           = 'System functions and variables';
SYNS_AttrSystemValue      = 'System value';
SYNS_AttrTerminator       = 'Terminator';
SYNS_AttrText             = 'Text';
SYNS_AttrUnknownWord      = 'Unknown word';
SYNS_AttrUser             = 'User functions and variables';
SYNS_AttrUserFunction     = 'User functions';
SYNS_AttrValue            = 'Value';
SYNS_AttrVariable         = 'Variable';
SYNS_AttrWhitespace       = 'Whitespace';
SYNS_AttrTableName        = 'Table Name';
SYNS_AttrMathMode         = 'Math Mode';

```

```

SYNS_AttrTextMathMode      = 'Text in Math Mode';
SYNS_AttrSquareBracket     = 'Square Bracket';
SYNS_AttrRoundBracket     = 'Round Bracket';
SYNS_AttrTeXCommand        = 'TeX Command';
SYNS_AttrOrigFile          = 'Diff Original File';
SYNS_AttrNewFile           = 'Diff New File';
SYNS_AttrChunkMarker       = 'Diff Chunk Marker';
SYNS_AttrChunkOrig         = 'Diff Chunk Original Line Count';
SYNS_AttrChunkNew          = 'Diff Chunk New Line Count';
SYNS_AttrChunkMixed        = 'Diff Chunk Line Counts';
SYNS_AttrLineAdded         = 'Diff Added line';
SYNS_AttrLineRemoved       = 'Diff Removed Line';
SYNS_AttrLineChanged       = 'Diff Changed Line';
SYNS_AttrLineContext       = 'Diff Context Line';
SYNS_AttrPrevValue         = 'Previous value';
SYNS_AttrMeasurementUnitValue = 'Measurement unit';
SYNS_AttrSelectorValue     = 'Selector';
SYNS_AttrFlags             = 'Flags';

```

Um ein Attribut mit Hilfe dieser Konstanten zu erstellen, müssen Sie zunächst die Art des Attributs bestimmen, das Sie erstellen möchten, und dann die Konstante wählen, die es am besten beschreibt. Für die meisten Syntaxe wären dies:

```

SYNS_AttrKommentar
SYNS_AttrReservedWord
SYNS_AttrNummer
SYNS_AttrSpace
SYNS_AttrString
SYNS_AttrSymbol
SYNS_AttrDirektive
SYNS_AttrAssembler

```

Um ein Attribut für Schlüsselwörter zu erstellen, könnten wir also Folgendes tun:

```

fAttrClave:=TSynHighlighterAttributes.Create(SYNS_AttrReservedWord,
                                              SYNS_XML_AttrReservedWord );

```

Die Verwendung von vordefinierten Konstanten, um die Attribute zu erstellen, ist nicht zwingend oder notwendig, damit der Highlighter funktioniert, aber es ist eine gute Praxis, wenn wir wollen, dass unsere Highlighter in der Lage sind, korrekt mit anderen Lazarus Programmen zu arbeiten. Für weitere Informationen über Attribute, siehe Abschnitt - [Attribute](#).

Es ist zu beachten, dass alle Elemente der zu untersuchenden Zeile zwangsläufig ein Token sein müssen, einschließlich Leerzeichen und Symbole.

Das folgende Beispiel zeigt, wie Sie eine Zeichenfolge in verschiedene Token aufteilen können:

|   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 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  |

In diesem Beispiel ist das erste Token durch die Zeichen 1 und 2 definiert und wird in Gelb dargestellt. Das zweite Token ist ein Leerzeichen und wird durch die Farbe Grün angezeigt. Die Zeichen 4 und 5 können als ein einziges Token oder als zwei verschiedene Token betrachtet werden. Das Zeichen 12 ist ein Token, das mit ziemlicher Sicherheit in die Kategorie "Zahl" fällt. Die Zeichen 14, 15 und 16 müssen zu einem einzigen Tokenraum von 3 Zeichen Breite zusammengefasst werden (es wäre ineffizient, sie als 3 Token zu behandeln). Ab Zeichen 17 gibt es ein Token, das bis zum Ende der Zeile reicht.

Die Grenzen des Tokens werden durch den Highlighter definiert (der als Token-Extraktor oder "Lexer" fungiert). Der Editor folgt unterwürfig den Angaben dieses Objekts und färbt es entsprechend den angegebenen Attributen ein.

Wie im Beispiel zu sehen, müssen alle Zeichen in der Zeile zu einem Token gehören. Die Größe eines Tokens liegt zwischen einem Zeichen und der Gesamtzahl der Zeichen in der Zeile. Der Editor wird die Zeile umso schneller verarbeiten, je weniger Token sie enthält.

Zur einfachen Identifizierung von Attributen ist es sinnvoll, eine Aufzählung für Token-Attribute zu erstellen:

```
// ID zur Kategorisierung von Token
TtkTokenKind = ( tkComment, tkKey, tkNull, tkNumber, tkSpace, tkString,
                  tkUnknown );
```

Die Kennung "tkUnknown" zeigt an, dass das aktuelle Token nicht identifiziert wurde. In diesem Fall wird davon ausgegangen, dass es keine Attribute hat.

Außerdem benötigen wir ein Feld zur Identifizierung des aktuellen Tokens:

```
TSynMiColor = class ( TSynCustomHighlighter )
...
  fTokenID : TtkTokenKind ; // Id des aktuellen Tokens
...
end;
```

Wenn wir nun dem aktuellen Token ein Attribut zuweisen wollen, müssen wir "fTokenID", den Bezeichner des Tokens, eingeben.

Sobald die Attribute erstellt sind, müssen wir der "Next"-Methode eine Funktionalität hinzufügen, damit sie die Token richtig aus der Arbeitslinie extrahieren kann. Die Implementierung muss so effizient wie möglich sein, weshalb wir die Methode der Funktions- oder Methodentabelle verwenden werden.



Die Idee ist, das Zeichen eines Tokens zu lesen und je nach ASCII-Wert eine entsprechende Funktion aufzurufen, um dieses Zeichen zu verarbeiten. Um den Aufruf effizient zu gestalten, erstellen wir eine Tabelle und füllen sie mit Zeigern auf die entsprechenden Funktionen.

```
Type
TProcTableProc = procedure of object; // Prozedurtyp zur Verarbeitung der
                                     // Token für das Anfangszeichen.
...
TSynMiColor = class(TSynCustomHighlighter)
protected
...
fProcTable: array[#0..#255] of TProcTableProc; // Funktionstabelle
...
end;
```

Der Typ "TProcTableProc" ist eine einfache Methode, die Prozeduren ohne Parameter definiert (damit der Aufruf schneller wird). Diese Art von Prozedur wird aufgerufen, wenn das Anfangszeichen eines Tokens identifiziert wird.

Nachdem nun der Typ der zu verwendenden Prozedur definiert wurde, müssen diese Token-Behandlungsprozeduren erstellt und die Methodentabelle mit ihren Adressen gefüllt werden. Der folgende Code ist ein einfaches Beispiel für das Füllen der Methodentabelle:

```
...
procedure TSynMiColor.CreateMethodTable;
{Erstellen Sie die Tabelle mit den Funktionen, die für jedes Anfangszeichen der
Token zu verarbeiten sind. Sie bietet eine schnelle Möglichkeit zur Verarbeitung
eines Tokens durch das Anfangszeichen}
var I: Char;
begin
  for I := #0 to #255 do
    case I of
      '_', 'A'..'Z', 'a'..'z': fProcTable[I] := @ProcIdent;
      #0 : fProcTable[I] := @ProcNull; // Zeichen für das Ende der Zeichenfolge
      #1..#9, #11, #12, #14..#32: fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
  end;
end;
```

Diese Methode ordnet die Adresse einer Funktion jeder der 256 Positionen in der Tabelle "fProcTable[]" zu.

Die Prozedur "ProcIdent" wird aufgerufen, wenn ein alphabetisches Zeichen (oder ein Bindestrich) erkannt wird, da es dem Anfang eines Bezeichners entspricht. Ihre Implementierung ist einfach:

```
procedure TSynMiColor.ProcIdent;
// Verarbeitet einen Bezeichner oder ein Schlüsselwort
begin
  while linAct[posFin] in [ '_', 'A'...'Z', 'a'...'z' ] do
    Inc (posFin);
  fTokenID := tkKey ;
end;
```

Die Zeichenkette "linAct" wird so lange gescannt, bis ein Zeichen gefunden wird, das kein gültiges Zeichen für einen Bezeichner ist. Beachten Sie, dass die erweiterten ASCII-Code-Zeichen (á,é,í,ä, etc.) nicht berücksichtigt werden. In diesem einfachen Beispiel wird nicht nach dem Typ des Bezeichners unterschieden, sondern das Attribut "tkKey" wird allen Bezeichnern zugewiesen. Wenn Sie nur einige wenige Wörter als "tkKey" markieren möchten, sollten Sie dies hier tun. Die Prozedur "ProcNull" wird aufgerufen, wenn das NUL-Zeichen erkannt wird, d. h. das Ende der Zeichenkette. Ihre Verarbeitung beschränkt sich also darauf, "fTokenID" als "tkNull" zu markieren.

```
Procedure TSynMiColor.ProcNull ;  
// Verarbeitet das Auftreten von Zeichen #0  
begin  
  fTokenID := tkNull; // Dies wird nur benötigt, um anzuzeigen, dass das Ende der  
                        Zeile erreicht wurde  
end;
```

Beachten Sie, dass bei der Erkundung der Kette kein weiterer Fortschritt erzielt wird. Dieses Verfahren ist wichtig, um das Ende der Kette zu erkennen, und ermöglicht es Ihnen, "GetEol" auf einfache Weise zu implementieren:

```
function TSynMiColor.GetEol : Boolean ;  
// Zeigt an, wenn das Ende der Zeile erreicht ist  
begin  
  Result := fTokenId = tkNull ;  
end;
```

Die "ProcSpace"-Prozedur ermöglicht die Verarbeitung von Blöcken mit Leerzeichen. Für Syntaxzwecke werden die ersten 32 Zeichen des ASCII-Codes als Leerzeichen betrachtet, mit Ausnahme der Zeichen #10 und #13, die Zeilenumbrüchen entsprechen:

```
procedure TSynMiColor.ProcSpace ;  
// Verarbeitet das Zeichen, das den Beginn des Leerzeichens darstellt  
begin  
  fTokenID := tkSpace;  
  repeat  
    Inc(posFin);  
  until (linAct[posFin] > #32) or (linAct[posFin] in [#0, #10, #13]);  
end;
```

Das Tabulatorzeichen #9 und das Leerzeichen #32 werden als Leerzeichen betrachtet. Diesen Leerzeichen wird das Attribut "tkSpace" zugewiesen, das normalerweise nicht hervorgehoben werden sollte.

Die andere wichtige Prozedur ist "ProcUnknown", die dazu dient, alle Token zu verarbeiten, die nicht zu einer speziellen Kategorie gehören. In unserem Fall sind das alle Symbole und Zahlen:

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

Es ist wichtig, immer ein Verfahren dieser Art zu haben, um alle Token zu berücksichtigen, die nicht in vordefinierte Gruppen eingeteilt sind. Beachten Sie, dass auch die UTF-8-Zeichen des erweiterten ASCII-Codes berücksichtigt werden. Dies ist normal, da SynEdit nur mit UTF-8 arbeitet.

Sobald diese grundlegenden Verfahren definiert sind, muss der Aufruf in der Methode "Next" implementiert werden. Der Code würde die folgende Form haben:

```
procedure TSynMiColor.Next;  
//Es wird von SynEdit aufgerufen, um auf das nächste Token zuzugreifen.  
begin  
  posIni := posFin; // zeigt auf das nächste Token  
  fProcTable[linAct[posFin]]; // Die entsprechende Funktion wird ausgeführt.  
end;
```

Obwohl es nicht offensichtlich ist, können Sie den Aufruf der entsprechenden Verarbeitungsfunktion für jedes Zeichen sehen. Offensichtlich muss "fProcTable" zuerst gefüllt worden sein.

Dieser Verarbeitungsmodus ist im Vergleich zu einer Reihe von Bedingungen oder sogar einer "case .. of"-Anweisung recht schnell.

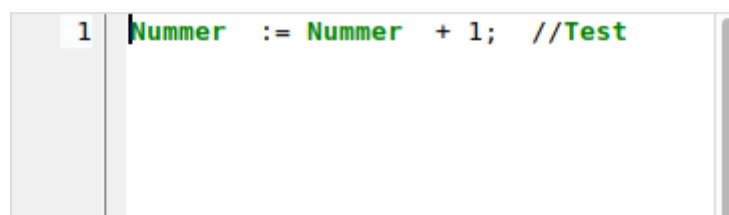
Die zugewiesene Verarbeitungsfunktion ist für die Aktualisierung des Index "posFin" zuständig, der immer auf den Anfang des nächsten Tokens oder das Ende der Kette zeigen muss.

Damit die Syntax erkannt wird, muss nur noch "GetTokenAttribute" geändert werden, um dem Editor mitzuteilen, welches Attribut für jedes Token zu verwenden ist:

```
function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;  
// Gibt Informationen über das aktuelle Token zurück  
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;
```

So wie wir unseren Highlighther definiert haben, werden alle Wörter als Schlüsselwörter erkannt und in Grün angezeigt. Symbole und andere druckbare Zeichen werden ohne Attribute angezeigt, d.h. sie nehmen die Standardfarbe des Textes an.

Der folgende Screenshot zeigt, wie ein einfacher Text unter Verwendung dieses Highlighthers aussehen würde:



Der vollständige Code der Unit würde wie folgt aussehen:

```
{Minimaleinheit zur Demonstration der Struktur einer einfachen Klasse, die für
die Syntaxhervorhebung verwendet wird.
Sie ist nicht funktional, sondern dient nur zur Veranschaulichung.
Erstellt von Tito Hinostroza: 04/08/2013}

unit uSyntax;

{$mode objfpc}{$H+}

interface

uses
Classes, SysUtils, Graphics, SynEditHighlighter;

type
{Klasse für die Erstellung eines Highligthers}

//ID zur Kategorisierung von Token
TtkTokenKind      = (tkComment, tkKey, tkNull, tkNumber, tkSpace, tkString,
tkUnknown);
TProcTableProc = procedure of object; //Prozedurtyp zur Verarbeitung des Tokens
nach dem Anfangszeichen.

{ TSynMiColor }
TSynMiColor = class(TSynCustomHighlighter)
protected
  posIni, posFin : Integer;
  linAct          : PChar;
  fProcTable      : array[#0..#255] of TProcTableProc; //Prozedurentabelle
  fTokenID        : TtkTokenKind; //Die Id des aktuellen Tokens definiert die
Token-Kategorien.
  fAtriComent     : TSynHighlighterAttributes; //Kommentar-Attribut
  fAtriClave      : TSynHighlighterAttributes; //Schlüsselwort-Attribut
  fAtriNumero     : TSynHighlighterAttributes; //Nummern-Attribut
  fAtriEspac      : TSynHighlighterAttributes; //Leerzeichen
  fAtriCadena     : TSynHighlighterAttributes; //String-Attribut
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);
//Konstruktor der Klasse. Hier müssen die zu verwendenden Attribute angelegt
werden.
begin
  inherited Create(AOwner);

  //Kommentar-Attribut
  fAtriComent      := TSynHighlighterAttributes.Create('Comment');
  fAtriComent.Style := [fsItalic]; //kursiv geschrieben
  fAtriComent.Foreground := clGray; //graue Schriftfarbe
  AddAttribute(fAtriComent);

  //Schlüsselwort-Attribut
  fAtriClave       := TSynHighlighterAttributes.Create('Key');
  fAtriClave.Style  := [fsBold]; //in Fettschrift
  fAtriClave.Foreground := clGreen; //grüne Schriftfarbe
  AddAttribute(fAtriClave);

  //Nummern-Attribut
  fAtriNumero      := TSynHighlighterAttributes.Create('Number');
  fAtriNumero.Foreground := clFuchsia;
  AddAttribute(fAtriNumero);

  //Leerzeichen. Keine Attribute
  fAtriEspac := TSynHighlighterAttributes.Create('space');
  AddAttribute(fAtriEspac);

  //String-Attribut
  fAtriCadena      := TSynHighlighterAttributes.Create('String');
  fAtriCadena.Foreground := clBlue; //blaue Schriftfarbe
  AddAttribute(fAtriCadena);

CreaTablaDeMetodos;
end;

//Tabelle zur Erstellung der Methoden
{Erstellt die Tabelle der Funktionen, die für jedes Anfangszeichen des zu
verarbeitenden Tokens verwendet werden.
Bietet eine schnelle Möglichkeit, ein Token nach Anfangszeichen zu verarbeiten.}
procedure TSynMiColor.CreaTablaDeMetodos;
var I: Char;
begin
  for I := #0 to #255 do
    case I of
      '_', 'A'..'Z', 'a'..'z'      : fProcTable[I] := @ProcIdent;
      #0                          : fProcTable[I] := @ProcNull; //Das Zeichen zur
Markierung des Zeichenkettenendes wird gelesen.
      #1..#9, #11, #12, #14..#32 : fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
  end;
end;

```

```
    end;
end;

procedure TSynMiColor.ProcIdent;
//Verarbeitet einen Bezeichner oder ein Schlüsselwort
begin
    while linAct[posFin] in ['_', 'A'..'Z', 'a'..'z'] do
        Inc(posFin);
        fTokenID := tkKey;
    end;

procedure TSynMiColor.ProcNull;
//Verarbeitet das Auftreten von Zeichen #0
begin
    fTokenID := tkNull; //Sie braucht dies nur, um anzuzeigen, dass das Ende der
    Leitung erreicht ist.
end;

procedure TSynMiColor.ProcSpace;
//Verarbeitet Zeichen, die den Beginn eines Leerzeichens darstellen
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 //fortgesetzter utf8-Subcode
        ((linAct[posFin] <> #0) and (fProcTable[linAct[posFin]] = @ProcUnknown))
    do inc(posFin);
    fTokenID := tkUnknown;
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
{Sie wird vom Editor aufgerufen, wenn er die Informationen über die
Färbung einer Zeile aktualisieren muss. Nach dem Aufruf dieser Funktion wird
erwartet, dass GetTokenEx, das aktuelle Token zurückgeben. Und auch nach jedem
Aufruf von "Next".}
begin
    inherited;
    linAct := PChar(NewValue); //Kopieren der aktuellen Zeile
    posFin := 0; //zeigt auf das erste Zeichen
    Next;
end;

procedure TSynMiColor.Next;
//Sie wird von SynEdit aufgerufen, um auf das nächste Token zuzugreifen.
begin
    posIni := posFin; //verweist auf das folgende Token
    fProcTable[linAct[posFin]]; //Die entsprechende Funktion wird ausgeführt.
end;
```

```
function TSynMiColor.GetEol: Boolean;
{Zeigt an, wenn das Ende der Zeile erreicht ist.}
begin
  Result := fTokenId = tkNull;
end;

procedure TSynMiColor.GetTokenEx(out TokenStart: PChar; out TokenLength:
integer);
{Gibt Informationen über den aktuellen Token zurück}
begin
  TokenLength := posFin - posIni;
  TokenStart := linAct + posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
//Gibt Informationen über den aktuellen Token zurück
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;

{Die folgenden Funktionen werden von SynEdit verwendet, um Klammern, geschweifte
Klammern und Anführungszeichen zu behandeln. Sie sind nicht entscheidend für
Token, aber sie sollten gut reagieren.}

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.
```

### 2.3.5 GetDefaultAttribute-Eigenschaft

Jemand hat sich vielleicht schon gefragt: Wie kann man von außerhalb der Klasse auf die Attribute von z.B. Schlüsselwörtern zugreifen? Denken Sie daran, dass die Attribute der Token im Highlighter und nicht in der übergeordneten Klasse "TSynCustomHighlighter" deklariert werden müssen.

Eine einfache Antwort wäre: "Wir setzen die Attributeigenschaften auf public, und dann können wir sie als beliebige Eigenschaft unseres Highlighters referenzieren.

Und es ist wahr, das würde funktionieren, aber wenn die Frage wäre: Wie kann man vom Editor aus auf die Attribute der Token zugreifen? Hier wird die Situation etwas kompliziert, denn obwohl der Editor (der Klasse TSynEdit) die Eigenschaft "Highlighter" hat, bezieht sie sich nur auf die Klasse "TSynCustomHighlighter" und nicht auf die abgeleitete Klasse (Highlighter), die wir immer zur Implementierung der Färbung verwenden.

Um dieses Problem teilweise zu lösen, wird empfohlen, eine zusätzliche Methode zu implementieren. Diese Methode heißt "GetDefaultAttribute" und wird es unserem Highlighter ermöglichen, auf Anfragen für den Zugriff auf die von "TSynCustomHighlighter" erzeugten Attribute zu reagieren.

Obwohl die Klasse "TSynCustomHighlighter" keine Eigenschaften des Attributtyps enthält (es bleibt dem Programmierer überlassen, die gewünschten Eigenschaften zu erstellen), enthält sie eine Möglichkeit, auf die wichtigsten Attribute der Token zuzugreifen. In der Klasse wurden feste Eigenschaften definiert die das Lesen oder Ändern der angegebenen Attribute ermöglichen:

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

Damit diese Eigenschaften jedoch funktionieren, müssen wir unseren Highlighter mit der folgenden Methode überschreiben:

```
function SynMiColor.GetDefaultAttribute(Index : integer): TSynHighlighterAttributes;  
{Diese Methode wird von der Klasse "TSynCustomHighlighter" aufgerufen, wenn auf eine der  
folgenden Komponenten zugegriffen wird seine Eigenschaften : CommentAttribute,  
IdentifierAttribute, KeywordAttribute, StringAttribute, SymbolAttribute oder  
WhitespaceAttribute.}  
begin  
  Case Index of  
    SYN_ATTR_COMMENT : Ergebnis := fCommentAttri ;  
    SYN_ATTR_IDENTIFIER : Ergebnis := fIdentifierAttri ;  
    SYN_ATTR_KEYWORD : Ergebnis := fKeyAttri ;  
    SYN_ATTR_WHITESPACE : Ergebnis := fSpaceAttri ;  
    SYN_ATTR_STRING : Ergebnis := fStringAttri ;  
    else Ergebnis := nil ;  
  end;  
end;
```



Wie Sie sehen, geht es darum, den Zugriff auf unsere Attribute je nach Art des angeforderten Attributs zu ermöglichen. Wenn wir ein bestimmtes Attribut nicht definiert haben, können wir natürlich NIL zurückgeben. Ebenso können wir zusätzliche Attribute definiert haben, auf die von außerhalb der Klasse nicht zugegriffen werden kann, weil sie nicht in die gewünschte Kategorie fallen.

Wenn jemand auf die Eigenschaft "CommentAttribute" von "TSynCustomHighlighter" zugreift, ruft er "GetDefaultAttribute" auf und übergibt den Parameter "SYN\_ATTR\_COMMENT". Es ist die Entscheidung des Programmierers, das Attribut zurückzugeben, das er für notwendig hält.

Normalerweise wird das Attribut zurückgegeben, das die Kommentare repräsentiert, aber die Klasse führt keine weitere Validierung durch. Theoretisch könnten wir jedes beliebige Attribut zurückgeben.

Wenn auf die oben genannten Eigenschaften von "TSynCustomHighlighter" nicht zugegriffen werden soll, ist es nicht notwendig, "GetDefaultAttribute" zu implementieren, aber es ist ratsam, es immer zu implementieren.

### 2.3.6 Erkennen von Schlüsselwörtern.

Im vorherigen Beispiel haben wir alle Bezeichner als Schlüsselwörter gekennzeichnet, indem wir ihnen das Attribut "tkKey" zugewiesen haben. Dies geschah in der Methode "ProcIdent":

```
procedure TSynMiColor.ProcIdent ;
// Verarbeitet einen Bezeichner oder ein Schlüsselwort
begin
  while linAct[posFin] in [ '_' , 'A'...'Z', 'a'...'z' ] do
    Inc(posEnd);
  fTokenID := tkKey ;
end;
```

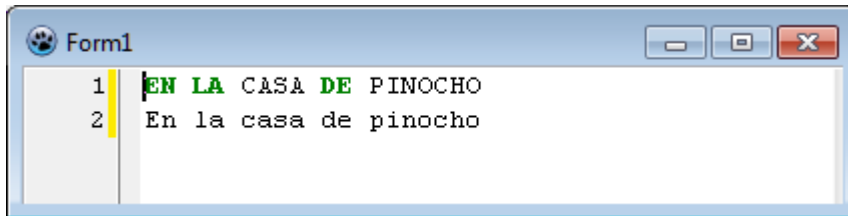
Im Normalfall werden jedoch nur einige Bezeichner als Schlüsselwörter markiert. Am einfachsten wäre es, das aktuelle Token mit einer Gruppe von Schlüsselwörtern zu vergleichen und sie nur bei Übereinstimmung als Schlüsselwörter zu markieren.

Der zu verwendende Code könnte folgendermaßen aussehen:

```
procedure TSynMiColor.ProcIdent;
// Verarbeitet einen Bezeichner oder ein Schlüsselwort
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 ; // gemeinsamer Bezeichner
end;
```

Der String-Vergleich wird mit der Funktion "strlcomp" durchgeführt, da es sich um eine "PChar"-Variable handelt.

In diesem Code werden nur die Wörter "EN", "LA" und "DE" als reservierte Wörter erkannt. Durch die Anwendung dieser Änderung könnten wir einen Bildschirm wie diesen erhalten.



Beachten Sie, dass die Routinen nur Großbuchstaben erkennen, da der String-Vergleich auf diese Weise durchgeführt wird.

Um weitere Wörter hinzuzufügen, können Sie die Liste erweitern und die Attribute für die einzelnen Wortkategorien auswählen. Diese Methode wird jedoch umständlich, wenn die Anzahl der hinzuzufügenden Wörter und Bedingungen wächst, und ist daher nicht der ideale Weg, um eine angemessene Syntax zu implementieren.

Später werden wir sehen, wie wir die Erkennung von Identifikatoren optimieren können.

### 2.3.7 Optimierung der Syntax.

Wie wir in dieser Beschreibung immer wieder betonen, ist es für eine gute Leistung des Editors wichtig, schnellen Code zu erhalten. Dazu müssen wir die Punkte identifizieren, an denen wir die Ausführungszyklen reduzieren können.

Bei der Analyse des vorherigen Codes wird deutlich, dass die Prozedur "ProcIdent" den größten Verarbeitungsaufwand verursacht. Aufgrund ihrer Implementierung müssen mehrere Vergleiche und Überprüfungen durchgeführt werden, um die zu färbenden Identifikatoren zu ermitteln.

Die erste Optimierung, die wir durchführen werden, hat mit der Bedingung zu tun:

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

Obwohl die Verwendung von Sets effizient ist, kann dieser Code durch die Verwendung einer Auswahltablette erheblich optimiert werden.

Die schnelle Feststellung, ob ein Zeichen in einer Liste enthalten ist, ist einfach, wenn wir das Problem aus einer anderen Perspektive betrachten. Stellen wir uns vor, dass jedes Zeichen mit einer Tabelle verbunden ist, die als Wert ein einfaches Flag enthält, das angibt, ob die Variable Teil der Liste ist oder nicht.

Eine solche Struktur wäre vom Typ:

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

Nun erstellen wir eine Ausfüllprozedur, die nur die Kästchen für gültige Zeichen für Bezeichner prüft, z. B. "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;
```

Sobald diese Tabelle gefüllt ist, können wir sie verwenden, um schnell zu erkennen, welche Zeichen als Teil eines Bezeichners gelten, wiederum mit einem einfachen while:

```
procedure TSynMiColor.ProcIdent;  
// Verarbeitet einen Bezeichner oder ein Schlüsselwort  
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; // gemeinsamer Bezeichner  
end;
```

Der nächste zu optimierende Punkt sind Mehrfachvergleiche. In diesem Beispiel gibt es logischerweise nur 4 Vergleiche, aber normalerweise können wir mit mehr als 100 arbeiten. Unter diesen Bedingungen kann, auch wenn es nicht so aussieht, wertvolle Zeit mit der Erkennung von Ketten verschwendet werden, da oft redundante Überprüfungen durchgeführt werden.

Das Problem besteht darin, den Vergleich einer Zeichenkette in einer Liste mit mehreren zu optimieren. Für die Optimierung dieser Aufgabe gibt es verschiedene Methoden.

Die meisten der Lazarus Syntax Komponenten benutzen die Hash-Funktionen Methode, die ein wenig komplex ist, aber im Grunde beinhaltet sie, dass jedem zu erkennenden Schlüsselwort ein numerischer Wert zugewiesen wird, mehr oder weniger einzigartig, was es erlaubt, es in eine kleine Anzahl von Gruppen zu kategorisieren (Siehe Anhang für mehr Details über diese Methode).

Obwohl diese Methode schnell ist, ist sie unübersichtlich und leicht verwirrend. Außerdem erfordern einfache Änderungen, wie das Hinzufügen eines neuen Schlüsselworts, eine sorgfältige Berechnung, bevor der Code geändert wird.

Hier werden wir eine Methode verwenden, die im Allgemeinen schneller und viel lesbarer ist und sich leichter ändern lässt. Dieser Algorithmus ist eine vereinfachte Form eines Präfixbaums. Er kann als ein Baum betrachtet werden, in dem nur die erste Ebene implementiert ist. Das erste Zeichen des zu suchenden Bezeichners wird als Präfix verwendet. Wir nennen diese Methode den Algorithmus Erstes Zeichen als Präfix.

Die Methode wird implementiert, indem eine erste Kategorisierung der Wörter unter Verwendung derselben Methodentabelle, die in "CreateMethodTable" erstellt wurde, vorgenommen wird, wobei für jeden Anfangsbuchstaben des Bezeichners eine Funktion erstellt wird. Der Code für "CreaTableOfMethods" würde also wie folgt aussehen:

```

Procedure TSynMiColor.CreateMethodTable ;
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 ;

      ...
      'H' , 'h' : fProcTable[I] := @ProcH ;

    end;

```

Anschließend wird in den Prozeduren ProcA, ProcB, ... usw. die Verarbeitung einer reduzierten Gruppe von Identifikatoren durchgeführt, wodurch die Anzahl der Vergleiche erheblich reduziert wird.

Die Prozedur, die für die Identifizierung der mit "L" beginnenden Schlüsselwörter zuständig ist, wäre zum Beispiel folgende:

```

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; // keine Attribute
end;

```

Es ist festzustellen, dass die Anzahl der durchzuführenden Vergleiche erheblich reduziert wird. Wären die zu vergleichenden Schlüsselwörter gleichmäßig im Alphabet verteilt, würde sich die Anzahl der Vergleiche sogar um das 26-fache verringern.

In seiner jetzigen Form erkennt dieses Verfahren nur reservierte Wörter in Großbuchstaben<sup>7</sup>. Um die Groß-/Kleinschreibung nicht zu berücksichtigen, müsste eine zusätzliche Verarbeitung vorgenommen werden.

Wir haben diese fehlende Funktionalität genutzt, um Vergleiche zu optimieren, indem wir eine Schnellvergleichsfunktion verwendet haben, die auch das Feld (Groß- oder Kleinschreibung) ignoriert. Dazu werden wir wieder einmal die unschätzbare Hilfe der Tabellen nutzen. Die Methode besteht darin, eine Tabelle zu erstellen, die jedem alphabetischen Zeichen unabhängig von seiner Groß- und Kleinschreibung eine Ordnungszahl zuweist. Wir nennen diese Tabelle "mHashTable" und nutzen die Gelegenheit, sie mit "CreaTableIdentif" zu füllen:

```
procedure CreateTableIdentif ;
var
  i, j : Char ;
begin
  for i := #0 to #255 do
    begin
      case i of
        '_' , '0'...'9', 'a'...'z', 'A'...'Z' : Identifier[i] := true ;
      else Identifier[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;
end;
```

---

<sup>7</sup> Sie können auch sehen, dass die Worterkennung nicht effektiv ist, weil sie Wörter auch dann erkennt, wenn sie nur mit den ersten paar Buchstaben übereinstimmen.

Nachdem wir diese Funktion erstellt haben, können wir nun eine Funktion für schnelle Vergleiche erstellen. Wir werden die Funktion verwenden, die in den Lazarus-Bibliotheken verwendet wird:

```
function TSynMiColor.KeyComp(const aKey: String): Boolean;
// Vergleicht schnell eine Zeichenkette mit dem aktuellen Token, auf
//das "fToIdent" verweist. Tokengröße muss in "fStringLen" enthalten sein
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
                if mHashTable[Temp^] <> mHashTable[aKey[i]] then
                begin
                    Result := False;
                    break;
                end;
            end;
            inc(Temp);
        end;
    end else Result := False;
end;
```

Diese Vergleichsfunktion verwendet den Zeiger "fToIdent" und die Variable "fStringLen" zur Auswertung des Vergleichs. Der einzige Parameter, den sie benötigt, ist die zu vergleichende Zeichenfolge.

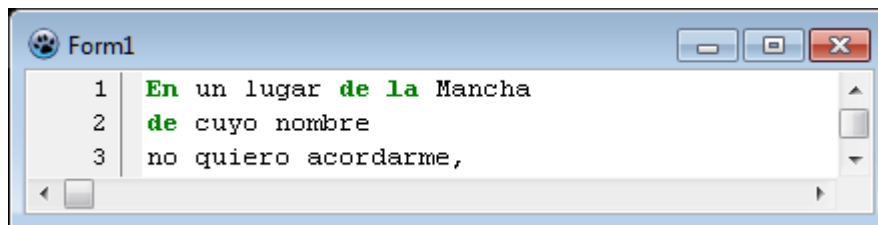
Mit Hilfe der Tabelle "mHashTable" wird der Vergleich ohne Berücksichtigung der Box durchgeführt.

Jetzt können wir das "ProcL"-Verfahren neu überdenken:

```
Procedure TSynMiColor.ProcL ;
begin
    while Identifier[linAct[posFin]] do inc ( posFin );
    fStringLen := EndPos - StartPos - 1 ; // Größe berechnen - 1
    fToIdent := linAct + posIni + 1 ; // Zeiger auf Bezeichner + 1
    if KeyComp ( 'A' ) then fTokenID := tkKey else
    if KeyComp ( 'OS' ) then fTokenID := tkKey else
        fTokenID := tkUnknown ; // keine Attribute
end;
```

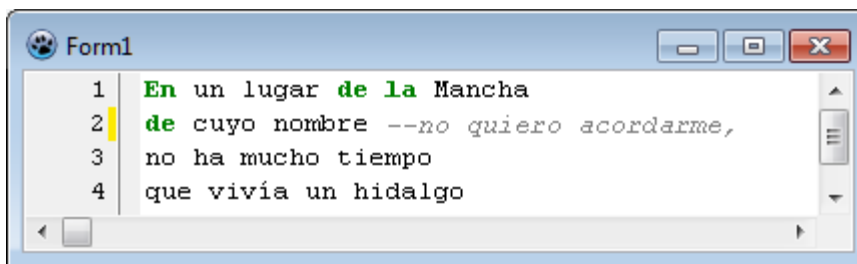
Diese Prozedur prüft, ob die Wörter "LA" oder "LOS" erkannt werden. Als weitere Optimierung wird der Vergleich des ersten Zeichens übersprungen, da es bereits vor dem Aufruf dieser Funktion erkannt wurde.

Jetzt haben wir unseren einfachen Highlighter fertig. Ein Test des Programms mit ein paar weiteren Schlüsselwörtern wird uns das folgende Ergebnis liefern:



### 2.3.8 Einzeiliger Kommentar Farbgebung

Dies sollte mit Highlighter geschehen. Das Verfahren ist einfach. Zunächst muss die Zeichenfolge ermittelt werden, die dem Anfang des Kommentars entspricht, und dann das Ende der Zeile.



In unserem Beispiel haben wir die Funktion "MakeMethodTables" um die Erkennung von Kommentaren erweitert, indem wir das Bindestrichzeichen "-" identifiziert haben, da das Token für einzeilige Kommentare der doppelte Bindestrich "--" ist.

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

Da wir in "CreateMethodTable" nur ein Zeichen erkennen können, müssen wir das nächste Zeichen in der Funktion "ProcMinus" erkennen.

Diese Funktion muss die folgende Form haben:

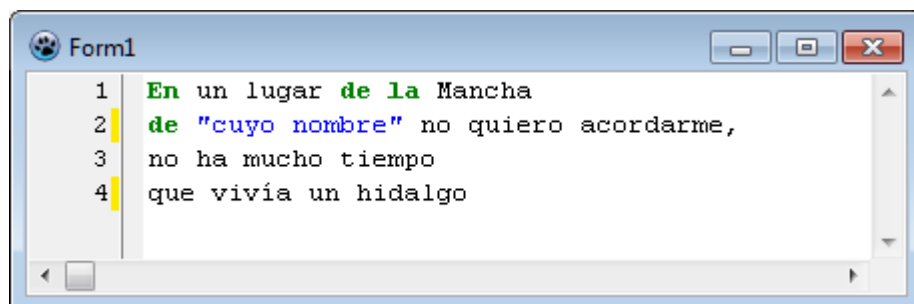
```
Procedure TSynMiColor.ProcMinus ;  
// Verarbeitet das Symbol ' - '  
begin  
  case LinAct[EndPos + 1] of // siehe nächstes Zeichen  
    ' - ' : // ist ein einzeliger Kommentar  
      begin  
        fTokenID := tkComment ;  
        inc ( PosFin, 2 ); // Sprung zum nächsten Token  
        while not ( linAct[EndPos] in [#0, #10, #13] ) do Inc ( PosFin );  
      end;  
    else // muss der "less"-Operator sein.  
      begin  
        inc ( PosEnd );  
        fTokenID := tkUnknown ;  
      end;  
    end  
  end;  
end;
```

Sie müssen das nächste Zeichen sehen, um festzustellen, ob es sich um das Kommentar-Token handelt. Wenn ja, wird das Ende der Zeile durchsucht, um den gesamten Block als ein einziges Token mit dem Attribut "tkComment" zu betrachten.

Wenn es sich nicht um das Kommentar-Token handelt, gehen wir einfach zum nächsten Token über und markieren das "-"-Symbol als vom Typ "tkUnknown". Wenn wir das "-"-Zeichen in eine besondere Kategorie einordnen wollen, ist dies der Punkt, an dem dies geschehen sollte.

### 2.3.9 Färbung der Kette

Der nächste Fall der Färbung entspricht der Färbung von Zeichenketten. Dies ist ein einfacher Fall, da die Zeichenketten höchstens eine Zeile einnehmen. Eine Zeichenkette hat einen Begrenzer, der sowohl für den Anfang als auch für das Ende der Zeichenkette verwendet wird.





In unserem Fall werden wir die durch Anführungszeichen getrennten Zeichenfolgen verwenden. Zunächst müssen wir seine Erkennung in die Prozedur "CreateMethodTable" einbeziehen:

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

Wenn das Anführungszeichen erkannt wird, wird die Kontrolle an "ProcString" übergeben, das das letzte Trennzeichen der Zeichenfolge findet und die gesamte Zeichenfolge als ein einziges Token markiert:

```
procedure TSynMiColor.ProcString ;  
// Verarbeitet das Anführungszeichen.  
begin  
  fTokenID := tkString ; // als Zeichenfolge markieren  
  Inc ( PosFin );  
  while ( not ( linAct[EndPos] in [#0, #10, #13] )) do  
    begin  
      if linAct[EndPos] = ''' then  
        begin // Suche nach Ende der Zeichenkette  
          Inc ( PosFin );  
          if ( linAct[EndPos] <> ''' ) then break ; // wenn nicht Anführungszeichen  
        end;  
        Inc ( PosFin );  
      end;  
    end;  
  end;  
end;
```

Beachten Sie, dass vor der Feststellung, ob das Ende der Zeichenkette gefunden wurde, zunächst überprüft wird, ob es sich nicht um doppelte Anführungszeichen handelt. Normalerweise stellt ein doppeltes Anführungszeichen "in Anführungszeichen" das Anführungszeichen dar.

Daraus lässt sich auch ableiten, dass das Zeichenketten-Token notwendigerweise in der gleichen Zeile endet, in der es begonnen hat. Es ist möglich, eine Einfärbung von mehrzeiligen Zeichenketten zu erzeugen, als ob es sich um mehrzeilige Kommentare handelte, was wir im Folgenden sehen werden.

## 2.3.10 Verwaltung eines Bereiches

Bevor wir uns ansehen, wie die Bereichsfärbung implementiert wird, ist es nützlich zu wissen, wie Bereiche im SynEdit-Editor behandelt werden.

Mit Hilfe von Bereichen können Sie Elemente einfärben, die über eine Zeile hinausgehen können. Dies ist der Fall bei Kommentaren oder mehrzeiligen Zeichenfolgen, die viele Sprachen implementieren.

Um diese Funktionalität zu implementieren, verarbeitet der Editor drei Methoden, die in der Klasse "TSynCustomHighlighter" der Unit "SynEditHighlighter" definiert sind:

```
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;
```

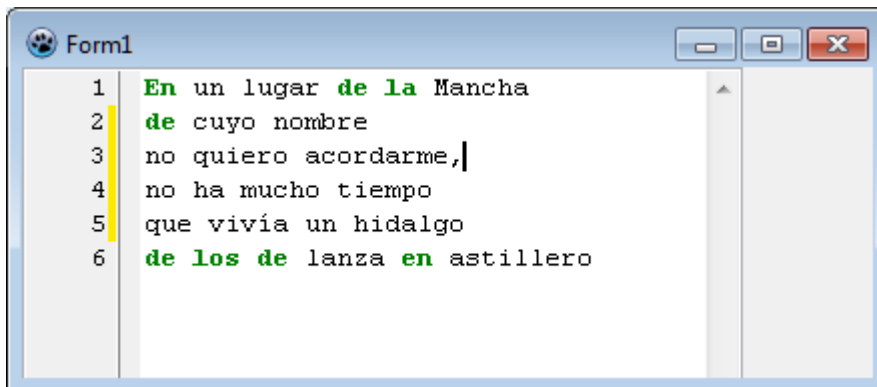
Diese Methoden haben die folgende Funktion:

- ResetRange - wird vor dem Scannen der ersten Textzeile ausgeführt, da es keine vorherigen Zeilen gibt, die den Bereich beeinflussen.
- GetRange - Wird nach Abschluss der Erkundung einer Zeile aufgerufen, um die Ebene am Ende der Zeile zu ermitteln. Dieser Wert wird intern gespeichert.
- SetRange - Wird vor dem Scannen einer Zeile aufgerufen. Ändern Sie die Ebene anhand der Ebene der vorherigen Zeile.

Die Reihenfolge der Ausführung hängt von der durchgeführten Aktion ab. Wenn ein Dokument zum ersten Mal geöffnet wird, durchsucht SynEdit alle Zeilen mit GetRange() (Lesen aller Token), um den Wert zu erhalten, der jeder Zeile entspricht. Der gelesene Wert wird intern für spätere Vergleiche gespeichert.

Jedes Mal, wenn ein Teil des Dokuments geändert wird, macht SynEdit mehrere Aufrufe an GetRange() und SetRange(), um den neuen Zustand des Dokuments zu rekonstruieren. Der Scan kann von der aktuellen Zeile bis zum Ende des Dokuments gehen, wenn SynEdit es für notwendig hält. Es ist jedoch üblich, dass das Dokument nur für einige Zeilen gescannt wird.

Das folgende Beispiel zeigt einen Editor und die Aufrufe der Methoden "SetRange" und "SetLine", wenn Zeile 3 geändert wird:



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

Der Editor durchsucht in der Regel den Text ab einer Zeile vor der geänderten Zeile, bis er feststellt, dass eine Zeile den gleichen Pegel wie vorher hat.

Wenn keine Syntaxverarbeitung durchgeführt werden soll, ist es nicht notwendig, diese Methoden außer Kraft zu setzen. Sie sollten nur geändert werden, wenn eine Bereichsfärbung oder eine Codefaltung implementiert werden soll.

Der Wert, den "SetRange" im Parameter "Value" sendet, ist ein Zeiger, ebenso wie der Wert, den "GetRange" zu empfangen erwartet, da sie für die Arbeit mit Objekten konzipiert wurden. Es ist jedoch nicht notwendig, mit Zeigern zu arbeiten. In der Praxis wird häufig ein Aufzählungstyp verwendet, um die Ebenen der Bereiche zu identifizieren, wobei darauf zu achten ist, dass die erforderlichen Konvertierungen vorgenommen werden.

```

Typ
...
TRangeState = ( rsUnknown, rsComment );
...

TSynMiColor = class ( TSynCustomHighlighter )
...
fRange : TRangeState ;
...
end;
...
function TSynMiColor.GetRange : Pointer ;
begin
    Result := Pointer( PtrInt ( fRange ));
Ende;

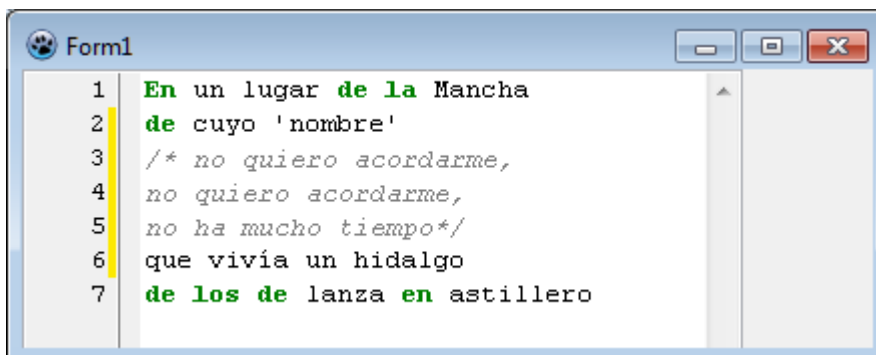
procedure TSynMiColor.SetRange ( Value : Pointer );
begin
    fRange := TRangeState ( PtrUInt ( Value ));
end;

```

Die Funktionen `PtrInt` und `PtrUInt` wandeln einen Zeiger in eine Ganzzahl der gleichen Größe wie der Zeiger um.

Der Wert der Zeiger ist an sich nicht wichtig<sup>8</sup>, da auf die Objekte, auf die sie zeigen, unter normalen Bedingungen nicht zugegriffen wird. Wichtig ist, dass sie andere Werte annehmen, wenn ein bestimmter Bereich gefunden wird (Kommentare, Blöcke usw.), als wenn es keine aktiven Bereiche gibt.

Im folgenden Beispiel wurde die Kommentarfärbung (`/* ... */`) implementiert, und die Werte der übergebenen und gelesenen Parameter werden angezeigt, wenn ab Zeile 3 ein Kommentar eingefügt wird:



8 Im Design von "TSynCustomHighlighter" wurde die Verwendung von Zeigern definiert, um Referenzen auf reale Objekte, die mit einem bestimmten Bereich assoziiert werden können, verwenden zu können. Für die meisten Fälle wird es ausreichen, einen einfachen Integer oder eine Aufzählung zu verwenden. In bestimmten Entwicklungen, wie der Klasse "TSynCustomFoldHighlighter", werden jedoch Objekte für ihre Funktionalität verwendet. Die Verwendung von Zeigern zur Verwaltung von Bereichen ist zunächst verwirrend, da man das Gefühl hat, dass ein einfacher Integer ausgereicht hätte, aber das aktuelle Design lässt mehr Freiheiten zu.

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
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\*/

Neben "SetRange" oder "GetRange" wird die Ordnungszahl von "fRange" als visuelle Hilfe angezeigt, um zu sehen, wie sie sich ändert. Wir stellen klar, dass die Änderung von "fRange" nicht fortlaufend sein muss. Es reicht aus, wenn "fRange" verschiedene Werte annimmt, damit die Einfärbefunktion funktioniert.

Das Konzept der Bereiche kann anfangs etwas schwierig zu verstehen sein. Es kann hilfreich sein, sich diese als eine Möglichkeit vorzustellen, zusätzliche Informationen zu speichern, die jeder Zeile entsprechen. Daher gibt es für jede gescannte Textzeile einen Bereichswert (Zeiger auf ein Objekt). Der Bereich ist nicht mit dem Konzept der "Ebene"<sup>9</sup>, oder "Umfang", es ist einfach: Zusätzliche Informationen, die mit jeder Zeile verbunden sind und dazu verwendet werden können, den Zustand einer Zeile am Ende der Zeile zu speichern. Wenn Sie diese Informationen nicht verwenden möchten, können Sie natürlich die Standardwerte beibehalten.

Es kann viele zusätzliche Informationen geben, die Sie jeder Zeile zuordnen möchten. Bereiche sind eine nützliche Möglichkeit, diese Informationen zu speichern. Der Editor liest den Bereich jeder Zeile am Anfang (durch den Highlighter), wenn er die Zeile fertig gelesen hat, und speichert diese Information.

Wenn der Text dann geändert wird, führt der Editor sukzessive Scans durch, um die Bereiche in den betroffenen Zeilen zu "aktualisieren".

In der Regel wird der Bereich der vorherigen Zeilen durch die Änderung einer Zeile nicht beeinflusst. Aber aufeinander folgende Zeilen können in ihrem Bereich geändert werden. Wenn dies der Fall ist, untersucht der Editor die folgenden Zeilen so lange, bis er feststellt, dass der Bereich, der ihr entspricht, dem bisherigen ähnlich ist, und stoppt die Aktualisierung.

Wenn sich die Informationen, die wir assoziieren wollen, nicht auf eine Zeile, sondern auf kleinere Elemente beziehen, sind Bereiche nicht direkt hilfreich<sup>10</sup>. Sie sind am nützlichsten, wenn die zu speichernde Information direkt oder leicht aus dem Zustand der vorherigen Zeile gewonnen werden kann.

---

9 Diese Bedeutung kann in einer bestimmten Umsetzung gegeben werden, ist aber nicht erforderlich.

10 Es könnten jedoch Wege gefunden werden, um die Informationen innerhalb der Linie in einem Bereichsobjekt zu rekonstruieren.

Bei der Verwaltung von Sortimenten muss das erfüllt sein:

"Die Kenntnis des Endzustands der vorherigen Zeile (Bereich oder Objekt, auf das der Bereich verweist) reicht aus, um mit der aktuellen Zeile korrekt zu arbeiten.

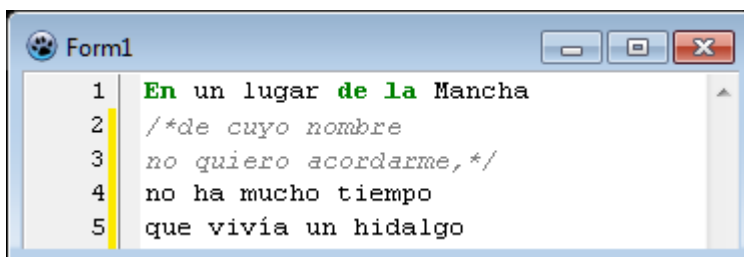
Eine andere Möglichkeit, die Nützlichkeit von Bereichen zu sehen, besteht darin, sie als Hilfsmittel zur Übertragung von Informationen von einer Zeile zur anderen zu betrachten, wenn man bedenkt, dass der Editor nicht den gesamten Text in geordneter Weise untersucht, sondern versucht, so wenig Untersuchungen wie möglich vorzunehmen, indem er<sup>11</sup> mehrere kleine Untersuchungen in verschiedenen Teilen des Textes durchführt, je nach dem geänderten Text.

Das bedeutet, dass wir, wenn wir zum Beispiel die Variable XXX ändern, während wir die Zeile "n" abfragen, nicht sicher sein können, dass der Wert von XXX beim Abfragen der Zeile "n+1" gelesen wird, da die nächste zu untersuchende Zeile nicht unbedingt "n+1" sein wird. Wenn wir wollen, dass der gewünschte Wert der Variablen XXX beim Scannen der Zeile "n+1" zu sehen ist, müssen wir sehen, wie wir ihn speichern und als Teil des Bereichs wiederherstellen können.

Der nächste Abschnitt wird zum Verständnis der Bereiche beitragen, indem er eine tatsächliche Implementierung in der Syntaxfärbung zeigt.

### 2.3.11 Bereich oder Kontextfärbung

Bei der Kontextfärbung wird ein Textabschnitt, der sich über eine oder mehrere Zeilen erstrecken kann, als ein einziges Token betrachtet. Logischerweise kann ein Token aufgrund der verwendeten Hierarchie nicht größer als eine Zeile sein. Erstreckt sich der Bereich also über mehr als eine Zeile, wird er als mehrere Token (eines für jede Zeile) derselben Kategorie identifiziert.



Bei unserer Klasse kann diese Einfärbung mit einer vom verwendeten Highlighter abgeleiteten Klasse erfolgen, oder die Funktionalität kann in den ursprünglichen Syntaxcode selbst aufgenommen werden.

---

11 Nur innerhalb einer Zeile werden die Token gescannt, immer der Reihe nach bis zum Ende der Zeile.

Wenn Sie eine abgeleitete Klasse erstellen möchten, muss diese eine ähnliche Struktur haben wie diese:

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

In den Methoden "Next" und "GetTokenAttribute" sollte das zusätzliche Verhalten, das zur Einfärbung der Textbereiche erforderlich ist, hinzugefügt werden.

Für weitere Informationen wird empfohlen, das mit Lazarus gelieferte Beispiel anzusehen: in \examples\SynEdit\NewHighlighterTutorial\

Am effizientesten wäre es jedoch, diese Funktion in den Highlighter selbst zu integrieren.

Betrachten wir nun den Fall der Einfärbung eines Bereichs von einer oder mehreren Zeilen. In diesem Beispiel betrachten wir die Einfärbung von mehrzeiligen Kommentaren.

Zuerst wählen wir die Begrenzungszeichen für den Kommentar. Für unser Beispiel werden wir die typischen C-Zeichen verwenden: "/\*" und "\*/". Der gesamte Text zwischen diesen Zeichen wird als Kommentar betrachtet und erhält das Attribut "fStringAttri".

Wir müssen das Anfangstrennzeichen durch das Zeichen "/" erkennen, aber die Validierung durchführen, da es sich um den Divisionsoperator handeln könnte.

Auch hier erfolgt das Abfangen in der Funktion "CreateMethodTable":

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

Und wir identifizieren Kommentare in "ProcSlash":

```
procedure TSynMiColor.ProcSlash ;
// Verarbeitet das Symbol ' / '
begin
  case linAct[EndPos + 1] of
    ' * ' :           // mehrzeiliger Kommentar
      begin
        fRange := rsComment ; // Bereich markieren
        inc ( PosFin, 2 );
        CommentProc; //Bearbeitung im Kommentar-Modus
      end;
    else // muss der "between"-Operator sein.
      begin
        inc ( PosEnd );
        fTokenID := tkUnknown ;
      end;
  end
end;
end;
```

Wir stellen fest, dass wir mit einer Prozedur "CommentProc" und mit einem neuen Flag namens "fRange" arbeiten. Dieses sollte wie gezeigt deklariert werden:

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

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

Diese Aussage ist wichtig für die Verwaltung von Bereichen. Die Erkennung von Färbungen in Bereichen erfordert diese Art der Verwaltung.

In "TRangeState" können je nach den Anforderungen der Syntax verschiedene Arten von Bereichen erstellt werden. Die Reihenfolge der aufgeführten Bereiche ist nicht wichtig.



Um die Bereichsfunktionalität zu implementieren, müssen Sie außerdem die drei Bereichsmethoden außer Kraft setzen:

```
TSynMiColor = class ( TSynCustomHighlighter )
    ...
    function GetRange : Pointer ; override ;
    procedure SetRange ( Value : Pointer ); override ;
    procedure ResetRange ; override ;
    ...
end;
...
{Implementierung von Bereichsfunktionalitäten}
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;
```

Sobald das Verhalten dieser Methoden definiert ist. Der Editor ist für die Verwaltung Ihrer Aufrufe zuständig und erspart uns die Arbeit, den Status der Zeilen zu kontrollieren.

Aber es ist immer noch notwendig, die Zeilen zu verarbeiten, die im Kommentarbereich liegen.

Zu diesem Zweck implementieren wir die Methode "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;
```

Diese Methode durchsucht die Zeilen nach dem letzten Begrenzungszeichen des Kommentars. Wird es nicht gefunden, betrachtet sie alles, was gescannt wurde (einschließlich der gesamten Zeile) als ein einziges Token vom Typ "tkComment". Beachten Sie, dass das Kommentarendebegrenzungszeichen "\*/" nirgendwo sonst in der Klasse erkannt wird.

Dieser sollte aufgerufen werden, wenn festgestellt wird, dass wir uns mitten in einem Kommentar befinden, wie es in "ProcSlash" geschieht, aber wir müssen ihn auch in "Next" einfügen:

```
procedure TSynMiColor.Next ;
begin
  posIni := PosFin; // zeigt auf das erste Element
  if fRange = rsComment then
    KommentarProc
  else
    begin
      fRange := rsUnknown ;
      fProcTable[linAct[PosFin]] ; // Die entsprechende Funktion wird
ausgeführt.
    end;
end;
```

Auf diese Weise übergeben wir die Kontrolle an "CommentProc", wenn wir uns inmitten eines Kontextkommentars befinden.

## 2.4 Die Klasse `TSynCustomHighlighter`

Bisher haben wir gezeigt, wie man einen einfachen Highlighter implementiert und dabei nur die notwendigen Eigenschaften und Methoden von `TSynCustomHighlighter` verwendet. Jetzt werden wir ein wenig über die Klasse selbst und einige zusätzliche Funktionalitäten sprechen, die sie mit sich bringt.

Ein kurzer Blick auf den Klassencode zeigt, dass es sich um eine mehr oder weniger umfangreiche Klasse handelt, wenn man bedenkt, dass sie eine abstrakte Klasse ist. Aber trotz allem speichert die Klasse `TSynCustomHighlighter` selbst keine Informationen. Alle Informationen, die sie verarbeitet, werden im Editor gespeichert, und zwar in `Lines[]`.

Jeder Editor, der eine Syntaxhervorhebung implementieren möchte, muss mit einem Highlighter verbunden sein. Die Beziehungen zwischen einem Editor und einem Highlighter sind:

- Einem Editor kann nur ein einziger Highlighter zugeordnet werden.
- Ein Highlighter kann einem oder mehreren Editoren dienen.

Diese Beziehung lässt sich ableiten, wenn man berücksichtigt, dass der Highlighter selbst keine Informationen über den von ihm untersuchten Text speichert.

Eine Folge der oben genannten Beziehungen ist, dass ein Highlighter nicht fest mit einem bestimmten Editor verbunden ist.

Aber unter normalen Bedingungen, wenn ein Editor einen einzelnen Highlighter verwendet, ist das vereinfachte Verhältnis eins zu eins.

Ein hervorstechendes Merkmal von `TSynCustomHighlighter` ist die Art und Weise, wie es Zeilen unter Verwendung des Konzepts der Reichweite erforscht.

Die vom Highlighter erzeugten Bereichsinformationen werden im Editor gespeichert und für den Zugriff auf jede Zeile mit dem entsprechenden Ausgangszustand verwendet.

### 2.4.1 `CurrentLines[]` und `CurrentRanges[]`

Eine der nützlichsten Eigenschaften von `TSynCustomHighlighter` ist `CurrentLines[]`. Diese Anordnung ermöglicht den Zugriff auf den "Puffer" des aktuellen Editors, d. h. desjenigen, der den Highlighter in diesem Moment verwendet.

Das Array `CurrentLines[]` wird `Lines[]` zugewiesen, bevor der Editor den Highlighter verwendet, so dass `CurrentLines[]` garantiert immer auf den aktuellen Editor verweist.

Genauso wie `CurrentLines[]` den Zugriff auf die Informationen der aktuellen Zeilen ermöglicht, erlaubt die Eigenschaft `CurrentRanges[]`<sup>12</sup> den Zugriff auf die Werte des Bereichs, der jeder Zeile des Textes entspricht.

`CurrentRanges[]` ist ein komplexes Objekt, aber für praktische Zwecke können wir es als eine einfache Tabelle mit Zeigern betrachten. Diese Werte sind diejenigen, die jeder Zeile zugewiesen werden, wenn `TSynCustomHighlighter` sie mit `GetRange()` vom Highlighter anfordert.

---

<sup>12</sup> Es sollte klargestellt werden, dass `CurrentRanges[]` als `PROTECTED` deklariert ist, also nicht direkt von außerhalb der Klasse zugänglich ist, aber es kann zugänglich gemacht werden, wenn wir unseren eigenen Textmarker erstellen.

Es ist wichtig zu wissen, dass `CurrentLines[]` und `CurrentRanges[]` wie Tabellen sind, die bei Null beginnen. Wenn wir zum Beispiel den Bereichswert von Zeile 3 sehen wollen, müssen wir `CurrentRanges[2]` betrachten.

Logischerweise haben `CurrentLines[]` und `CurrentRanges[]` immer die gleiche Größe, und diese Größe entspricht der Anzahl der Zeilen, die der aktuelle Editor hat (wenn der Highlighter einem zugewiesen ist).

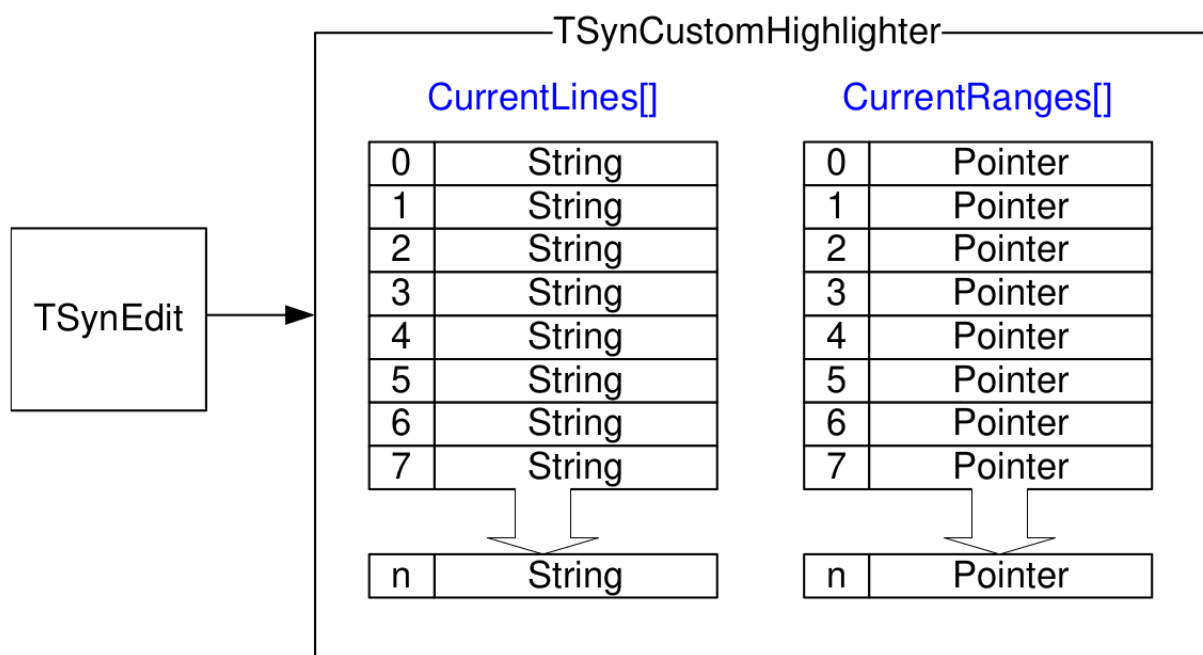
Der Versuch, auf `CurrentLines[]` oder `CurrentRanges[]` zuzugreifen, wenn der Highlighter nicht einem Editor zugewiesen ist, führt zu einem Laufzeitfehler.

Die Tabelle `CurrentLines[]` gibt Strings zurück, `CurrentRanges[]` dagegen Zeiger. Sie gibt dieselben Zeiger zurück, die bei der Verwendung der Methoden zugewiesen werden:

- `TSynCustomHighlighter.GetRange()`
- `TSynCustomHighlighter.SetRange()`

Das heißt, mit `CurrentRanges[]` können wir die in jeder Zeile gespeicherten Bereichsinformationen wiederherstellen.

Das folgende Diagramm soll das Gesagte verdeutlichen:



Natürlich ist die Bedeutung dieses Zeigers die gleiche wie die, die im Highlighter verwendet wird (enumeriert, Integer, Objektreferenz, etc.). Es obliegt dem Programmierer, die entsprechende Typkonvertierung vorzunehmen.

## 2.4.2 Einige Methoden und Eigenschaften

Die Klasse `TSynCustomHighlighter` verfügt zusätzlich über verschiedene nützliche Eigenschaften, die für uns irgendwann einmal nützlich sein können.

Die Methode `ScanAllRanges()` veranlasst den Editor, alle Zeilen erneut zu durchsuchen, um die Verweise auf den Bereich, den jede Zeile hat, zu aktualisieren. Diese Methode ist z. B. nützlich, wenn die Syntax der Arbeitssprache geändert wird.

Die Eigenschaft `"LanguageName"` gibt eine Zeichenkette mit dem Text zurück, der dem Namen der Sprache entspricht, für die der Highlighter vorbereitet ist. Es liegt in der Verantwortung des Highlighters, diese Eigenschaft zu aktivieren, indem die Methode `GetLanguageName()` überschrieben wird.

Die Eigenschaft `"SampleSource"` gibt eine Zeichenkette mit einem Stück Beispielcode in der verwendeten Sprache zurück, um den Highlighter zu testen. Auch hier liegt es in der Verantwortung des Highlighters, diese Eigenschaft zu aktivieren, indem die Methode `GetLanguageName()` überschrieben wird.

Es gibt eine interne Eigenschaft, die für die Arbeit mit Identifikatoren verwendet werden kann: `IdentChars`. Intern handelt es sich um einen einfachen Verweis auf `GetIdentChars()`, der die folgende Definition hat:

```
function TSynCustomHighlighter.GetIdentChars : TSynIdentChars ;  
begin  
    Result := [#33..#255] ;  
end;
```

Es kann notwendig sein, diese Methode für unsere Bedürfnisse zu überschreiben.

Mit der Methode `StartAtLineIndex()` können Sie den Highlighter auf einer beliebigen Zeile des Textes positionieren, um mit der Erkundung dieser Zeile zu beginnen. Dies ist nützlich, wenn wir einen zusätzlichen Scan (außerhalb des normalen Highlighter-Scanprozesses) durchführen wollen, um zusätzliche Informationen vom Highlighter zu erhalten.

Ein Beispiel für die Verwendung dieser Funktion ist in der Methode zu sehen: `TSynEdit.GetHighlighterAttriAtRowCol()`, aus dem Editor, die es ermöglicht, das Attribut und das Token für eine beliebige Position im Text zu erhalten.

## 2.4.3 Attribute

Ein Großteil des Codes in `TSynCustomHighlighter` bezieht sich auf die Attributverwaltung. Damit können Sie konfigurieren, wie die Token im Editor angezeigt werden sollen.

Attribute sind Objekte, die vor der Verwendung des Highlighters erstellt werden sollten. Um ein neues Attribut zu erstellen, führen Sie folgendes aus:

```
fKeyAttri:= TSynHighlighterAttributes.Create ( SYNS_AttrKey, SYNS_XML_AttrKey );  
fKeyAttri.Style := [fsBold] ;  
AddAttribute ( fKeyAttri ); // Speichern des Verweises auf das Attribut
```

Normalerweise wird dieser Code immer im Konstruktor des Highlighters platziert. Sie können aber auch an jeder anderen Stelle des Prozesses dynamisch erstellt werden.

Verweise auf Attribute werden in einer internen Struktur der Klasse (fAttributes) gespeichert.

Außerdem werden sie freigegeben, wenn die Klasse zerstört wird, so dass es nicht notwendig ist (und auch nicht sein sollte), die dem Highlighter hinzugefügten Attribute zu zerstören.

Alle Highlighter haben mehrere Attribute definiert (statisch oder dynamisch), weil sie notwendig sind, um die Hervorhebungseigenschaften des Textes zuzuweisen.

Attribute sind Objekte der Klasse `TSynHighlighterAttributes`, die in derselben `SynEditHighlighter`-Unit definiert sind, die auch die Klasse `TSynCustomHighlighter` enthält.

Die Klasse `TSynCustomHighlighter` verfügt nicht über intern definierte Attribute, wie es ihrer Funktionsweise entspricht, aber sie hat definierte Eigenschaften, die den Zugriff auf einige der gängigsten Attribute ermöglichen:

```
TSynCustomHighlighter = class (TComponent)
...
public
  property AttrCount : integer read GetAttrCount ;
  property Attribut[idx : integer] : TSynHighlighterAttributes
    read GetAttribute ;
  property Capabilities : TSynHighlighterCapabilities
    read {$IFDEF SYN_LAZARUS}FCapabilities{$ ELSE }GetCapabilities{$ENDIF} ;
  property SampleSource : string read GetSampleSource write SetSampleSource ;
  property CommentAttribute : TSynHighlighterAttributes
    index SYN_ATTR_COMMENT read GetDefaultAttribute ;
  property IdentifierAttribute : TSynHighlighterAttributes
    index SYN_ATTR_IDENTIFIER read GetDefaultAttribute ;
  property KeywordAttribute : TSynHighlighterAttributes
    index SYN_ATTR_KEYWORD read GetDefaultAttribute ;
  property StringAttribute : TSynHighlighterAttributes
    index SYN_ATTR_STRING read GetDefaultAttribute ;
  property SymbolAttribute : TSynHighlighterAttributes // mh 2001 - 09 - 13
    index SYN_ATTR_SYMBOL read GetDefaultAttribute ;
  property WhitespaceAttribute : TSynHighlighterAttributes
    index SYN_ATTR_WHITESPACE read GetDefaultAttribute ;
```

Dieser Zugriff hängt von der korrekten Implementierung der Methode `GetDefaultAttribute()` ab. Zur Vereinfachung der Implementierung gibt es einige vordefinierte Konstanten in der Unit `"SynEditHighlighter"`:

```
const
  SYN_ATTR_COMMENT      = 0 ;
  SYN_ATTR_IDENTIFIER   = 1 ;
  SYN_ATTR_KEYWORD      = 2 ;
  SYN_ATTR_STRING       = 3 ;
  SYN_ATTR_WHITESPACE   = 4 ;
  SYN_ATTR_SYMBOL       = 5 ;
```

Diese Konstanten decken nicht die Anzahl der Attribute ab, die ein ziemlich kompletter Highlighter verarbeiten kann, aber sie sind eine einfache Hilfe, um auf die häufigsten Attribute zuzugreifen.

Eine typische Implementierung von `GetDefaultAttribute()`, in einem Highlighter ist:

```
function
    TSynLFMSyn.GetDefaultAttribute(Index:integer):TSynHighlighterAttributes;
begin
    case Index of
        SYN_ATTR_COMMENT      : Result := fCommentAttri ;
        SYN_ATTR_IDENTIFIER   : Result := fIdentifierAttri ;
        SYN_ATTR_KEYWORD       : Result := fKeyAttri ;
        SYN_ATTR_STRING        : Result := fStringAttri ;
        SYN_ATTR_WHITESPACE    : Result := fSpaceAttri ;
        SYN_ATTR_SYMBOL        : Result := fSymbolAttri ;
    else
        Result := nil ;
    end;
end;
```

Die Idee ist, den entsprechenden Verweis auf den Highlighter zurückzugeben, wenn er über `GetDefaultAttribute()` angefordert wird. Die Attribute sind im Highlighter definiert. Aus dem, was wir über die Attribute gesehen haben, lässt sich ableiten, dass es verschiedene Möglichkeiten gibt, auf die Attribute eines Highlighter zuzugreifen. Der einfachste Weg, auf Attribute zuzugreifen, ist über `GetDefaultAttribute()`:

```
var attribute : TSynHighlighterAttributes ;
...
attribute := SynLFMSyn1.GetDefaultAttribute ( SYN_ATTR_KEYWORD );
attribute.Foreground := clRed ;
```

Dieser Weg funktioniert jedoch nur für Attribute, die in der Methode `GetDefaultAttribute()` korrekt eingegeben wurden, was von der korrekten Implementierung des Highlighters abhängt. Außerdem sind nur einige der gebräuchlichsten Attribute sichtbar.

Eine andere Möglichkeit, auf Attribute zuzugreifen, ist die Verwendung der öffentlichen Eigenschaften, die viele Highlighter implementieren.

Diese sind:

```
TSynLFMSyn = class ( TSynCustomFoldHighlighter )
...
private
    fCommentAttri      : TSynHighlighterAttributes ;
    fIdentifierAttri    : TSynHighlighterAttributes ;
    fKeyAttri           : TSynHighlighterAttributes ;
    fNumberAttri        : TSynHighlighterAttributes ;
    fSpaceAttri         : TSynHighlighterAttributes ;
    fStringAttri        : TSynHighlighterAttributes ;
    fSymbolAttri        : TSynHighlighterAttributes ;
...
published
    property CommentAttri : TSynHighlighterAttributes read fCommentAttri
        write fCommentAttri ;
    property IdentifierAttri : TSynHighlighterAttributes read fIdentifierAttri
        write fIdentifierAttri ;
    property KeyAttri : TSynHighlighterAttributes read fKeyAttri write fKeyAttri ;
    property NumberAttri : TSynHighlighterAttributes read fNumberAttri
        write fNumberAttri ;
    property SpaceAttri : TSynHighlighterAttributes read fSpaceAttri
        write fSpaceAttri ;
    property StringAttri : TSynHighlighterAttributes read fStringAttri
        write fStringAttri ;
end;
```

Die Attribute selbst sind ausgeblendet, aber ihre jeweiligen Eigenschaften sind veröffentlicht, so dass sie über den Objektinspektor geändert werden können.

Abhängig von der Implementierung des Highlighters könnten diese Eigenschaften eine bessere Sichtbarkeit für den Zugriff auf die Attribute des Highlighters ermöglichen.

Eine Möglichkeit, auf die Attribute zuzugreifen, wäre also die Verwendung dieser Eigenschaften:

```
var attribute : TSynHighlighterAttributes ;
...
    attribute := SynLFMSyn1.KeywordAttribute ;
    attribute.Foreground := clRed ;
```

Eine andere Möglichkeit, auf alle Attribute des Highlighters zuzugreifen, wäre die Verwendung der Attributtabelle `Attribute[]`, die ein einfacher Verweis auf die geschützte Methode `GetAttribute()` ist:

```
function TSynCustomHighlighter.GetAttribute (idx:integer):
    TSynHighlighterAttributes ;
begin
    Result := nil ;
    if ( idx >= 0 ) and ( idx < fAttributes.Count ) then
        Result := TSynHighlighterAttributes (fAttributes.Objects[idx]);
end;
```



Attribute werden intern in fAttributes, einer TStringList, im Objektfeld gespeichert.  
Ein typischer Zugriff auf ein Attribut über den Highlighter wäre:

```
var attribute : TSynHighlighterAttributes ;  
...  
    attribute := SynLFMSyn1.Attribute[2] ;  
    attribute.Foreground := clRed ;
```

Der Nachteil ist, dass wir den Index des Attributs, mit dem wir arbeiten wollen, kennen müssen. Als Hilfe könnte man den Namen des Attributs verwenden.  
Eine typische Iteration von Attributen wäre also:

```
var i : integer;  
begin  
...  
    for i := 0 to SynLFMSyn1.AttrCount - 1 do  
        ShowMessage ( SynLFMSyn1.Attribute[i].Name );  
...  
end;
```

Der in der Eigenschaft "Name" gespeicherte Name ist der Name, der dem Attribut zugewiesen wird, wenn es mithilfe seines Konstruktors erstellt wird:

```
Attribut := TSynHighlighterAttributes.Create ( 'Name' );
```

Gleichzeitig können wir auch den zweiten Parameter des Konstruktors (aStoredName) verwenden:

```
constructor TSynHighlighterAttributes.Create (aCaption : string; aStoredName :  
String = '' );
```

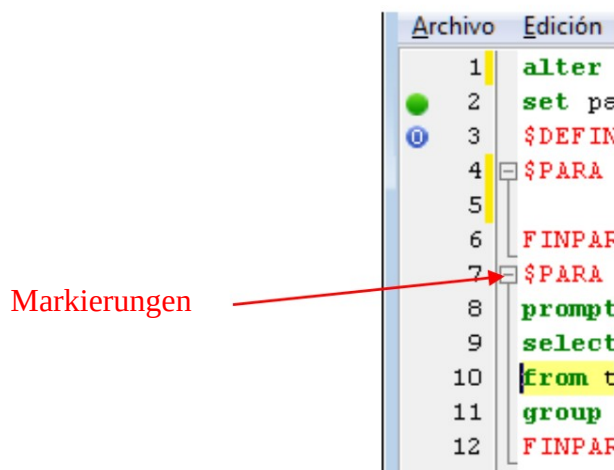
Dadurch können Sie dem Attribut einen anderen internen Namen zuweisen. Auf diesen Namen können wir dann über die Eigenschaft "StoredName" des Attributs zugreifen.

Um die Anzahl der Attribute zu kennen, haben wir die Eigenschaft "AttrCount".

Durch Iteration über Attribut[] können Sie effektiv auf alle Attribute zugreifen, die im Highlighter erstellt wurden.

## 2.5 Code-Faltfunktionalität

Die Code-Folding-Funktion bezieht sich auf die Möglichkeit, die Länge eines vordefinierten Textblocks in der linken Leiste des Editors anzuzeigen:



Die meisten der Syntaxkomponenten, die in Lazarus enthalten sind, beinhalten keine Folding-Funktionalität, so dass diese im Code erstellt werden muss. Der Code für das Falten muss sich im selben Highlighter befinden.

### 2.5.1 Basic Folding

Betrachten wir das Hinzufügen von Codefaltung, beginnend mit einem einfachen Highlighter, wie wir ihn oben beschrieben haben.

Um die Faltung zu implementieren, müssen Sie die Highlighter-Klasse von der Klasse "TSynCustomFoldHighlighter" ableiten (definiert in der Unit "SynEditHighlighterFoldBase"), anstatt die Klasse "TSynCustomHighlighter" zu verwenden.

Diese neue Klasse enthält den Verarbeitungscode für die Faltung.

Zunächst ändern wir die Deklaration der Klasse "Highlighter", die wir verwenden:

```
TSynMiColor = class (TSynCustomHighlighter)
...
```

in:

```
TSynDemoHlFold = class (TSynCustomFoldHighlighter)
...
```

Aber wir müssen auch die Behandlungsmethoden für das Sortiment ändern:

```
{Implementierung von Bereichsfunktionalitäten}
procedure TSynMiColor.ReSetRange;
begin
  inherited;
  fRange := rsUnknown;
end;

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

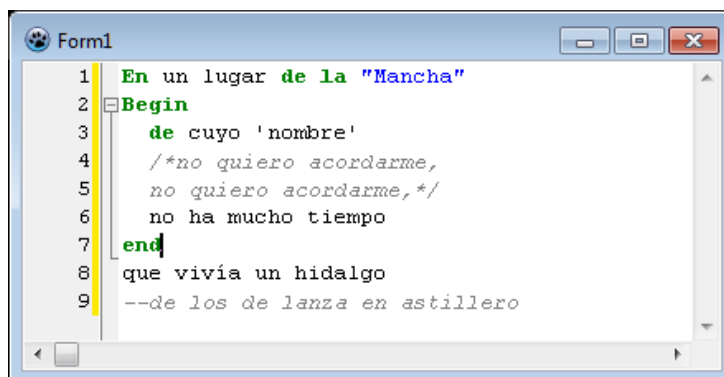
procedure TSynMiColor.SetRange(Value: Pointer);
begin
  inherited;
  fRange := TRangeState(PtrUInt(CodeFoldRange.RangeType));
end;
```

"inherited" kümmert sich darum um angemessen auf die Anforderungen der Klasse zu reagieren und wir verschieben die Kontextinformationen nach "CodeFoldRange", denn dort arbeitet nun "TsynCustomFoldHighlighter". Sobald diese Änderungen vorgenommen wurden, können wir die "Faltung" hinzufügen. Dazu müssen wir den Anfang und das Ende des Blocks erkennen.

Der einfachste Weg ist der Aufruf der Methoden: "StartCodeFoldBlock" und "EndCodeFoldBlock", wenn die Start- bzw. End-Token erkannt werden. Wenn wir zum Beispiel den Anfang mit der Erkennung des Wortes BEGIN und das Ende des Blocks mit dem Wort END hinzufügen würden, würde der Code lauten:

```
procedure TSynMiColor.ProcB;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1; // Größe berechnen - 1
  fToIdent := linAct + posIni + 1; // Zeiger auf Bezeichner + 1
  if KeyComp('EGIN') then
  begin
    fTokenID := tkKey;
    StartCodeFoldBlock(nil);
  end
  else
  if KeyComp('Y')
  then fTokenID := tkKey
  else
  fTokenID := tkUnknown; // gemeinsamer Bezeichner
  end;
  ...
procedure TSynMiColor.ProcE;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1; // Größe berechnen - 1
  fToIdent := linAct + posIni + 1; // Zeiger auf Bezeichner + 1
  if KeyComp('N') then fTokenID := tkKey else
  if KeyComp('ND') then
  begin
    fTokenID := tkKey;
    EndCodeFoldBlock();
  end
  else
  fTokenID := tkUnknown; // gemeinsamer Bezeichner
end;
```

Ein Testtext könnte folgendermaßen aussehen:



## 2.5.2 Beispiel-Code

Nachfolgend ist ein vollständiger Beispielcode mit Syntaxfärbung von Bezeichnern, Zeichenketten, ein- und mehrzeiligen Kommentaren und Faltungen dargestellt:

```
{Minimale Funktionseinheit, die die Struktur einer einfachen Klasse demonstriert
die vollständige Syntaxfärbung und Codefaltung implementiert.
Von Tito Hinostroza: 07/08/2013}
unit asyntax;

{$mode objfpc}{$H+}

interface

uses
Classes, SysUtils, Graphics, SynEditHighlighter, SynEditHighlighterFoldBase;

type
{Klasse für die Erstellung eines Highlighters}
TRangeState = (rsUnknown, rsComment);
//ID zur Kategorisierung der Token
TtkTokenKind = (tkComment, tkKey, tkNull, tkSpace, tkString, tkUnknown);
TProcTableProc = procedure of object; //Prozedurtyp zur Verarbeitung des Tokens nach dem
Anfangszeichen.

{ TSynMiColor }
TSynMiColor = class(TSynCustomFoldHighlighter)
protected
  posIni, posFin : Integer;
  fStringLen      : Integer; //Aktuelle Tokengröße
  fToIdent        : PChar; //Zeiger auf Bezeichner
  linAct          : PChar;
  fProcTable      : array[#0..#255] of TProcTableProc; //Tabelle der Verfahren
  fTokenID        : TtkTokenKind; //Id des aktuellen Tokens
  fRange          : TRangeState; //definiert die Kategorien von Token
  fAtriComent     : TSynHighlighterAttributes;
  fAtriClave      : 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 CommentProc;
  procedure CreaTablaDeMetodos;
  function KeyComp(const aKey: String): Boolean;
  procedure ProcMinus;
//Funktionen zur Verarbeitung von Identifikatoren
  procedure ProcNull;
```

```
procedure ProcSlash;
procedure ProcSpace;
procedure ProcString;
procedure ProcUnknown;
procedure ProcB;
procedure ProcC;
procedure ProcD;
procedure ProcE;
procedure ProcL;
function GetRange: Pointer; override;
procedure SetRange(Value: Pointer); override;
procedure ResetRange; override;
end;

implementation

var
  Identifiers: array[#0..#255] of ByteBool;
  mHashTable: array[#0..#255] of Integer;

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;
end;

constructor TSynMiColor.Create(AOwner: TComponent);
//Konstruktor der Klasse. Hier müssen die zu verwendenden Attribute angelegt werden.
begin
  inherited Create(AOwner);
  //Kommentar-Attribut
  fAtriComent := TSynHighlighterAttributes.Create('Comment');
  fAtriComent.Style := [fsItalic]; //kursiv geschrieben
  fAtriComent.Foreground := clGray; //graue Schriftfarbe
  AddAttribute(fAtriComent);

  //Schlüsselwort-Attribut
  fAtriClave := TSynHighlighterAttributes.Create('Key');
  fAtriClave.Style := [fsBold]; //fettgedruckt
  fAtriClave.Foreground:=clGreen; //grüne Schriftfarbe
  AddAttribute(fAtriClave);

  //Leerzeichen Attribute. Keine Attribute
  fAtriEspac := TSynHighlighterAttributes.Create('space');
  AddAttribute(fAtriEspac);
```

```
//String-Attribut
fAtriCadena := TSynHighlighterAttributes.Create('String');
fAtriCadena.Foreground := clBlue; //blaue Schriftfarbe
AddAttribute(fAtriCadena);

CreaTablaDeMetodos;
end;

//Tabelle zur Erstellung von Methoden
procedure TSynMiColor.CreaTablaDeMetodos;
var I: Char;
begin
  for I := #0 to #255 do
    case I of
      '-' : fProcTable[I] := @ProcMinus;
      '"' : fProcTable[I] := @ProcString;
      '/' : fProcTable[I] := @ProcSlash;
      'B', 'b': fProcTable[I] := @ProcB;
      'C', 'c': fProcTable[I] := @ProcC;
      'D', 'd': fProcTable[I] := @ProcD;
      'E', 'e': fProcTable[I] := @ProcE;
      'L', 'l': fProcTable[I] := @ProcL;
      #0 : fProcTable[I] := @ProcNull;
    //Das Zeichen zur Markierung des Zeichenkettenendes wird gelesen.
      #1..#9, #11, #12, #14..#32 : fProcTable[I] := @ProcSpace;
      else fProcTable[I] := @ProcUnknown;
    end;
  end;

function TSynMiColor.KeyComp(const akey: String): Boolean;
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
              Result := False;
              break;
            end;
          inc(Temp);
        end;
      end else Result := False;
    end;

procedure TSynMiColor.ProcMinus;
//Verarbeitet das Symbol '-'.
begin
  case LinAct[PosFin + 1] of
    //siehe nächstes Zeichen
    '-':
    //ist ein einzeiliger Kommentar
    begin
      fTokenID := tkComment;
```

```
        inc(PosFin, 2);
//zum nächsten Token springen
        while not (linAct[PosFin] in [#0, #10, #13]) do Inc(PosFin);
    end;
    else //muss der "Minus"-Operator sein.
    begin
        inc(PosFin);
        fTokenID := tkUnknown;
    end;
end;

procedure TSynMiColor.ProcString;
//Verarbeitet das Anführungszeichen.
begin
    fTokenID := tkString;
//Token als String
    Inc(PosFin);
    while (not (linAct[PosFin] in [#0, #10, #13])) do
    begin
        if linAct[PosFin] = '"' then begin //sucht das Ende des Strings
            Inc(PosFin);
            if (linAct[PosFin] <> '"') then break; //wenn nicht doppelte Anführungszeichen
        end;
        Inc(PosFin);
    end;
end;

procedure TSynMiColor.ProcSlash;
//Verarbeitet das '/'-Symbol
begin
    case linAct[PosFin + 1] of
        '*':
//mehrzeiliger Kommentar
        begin
            fRange := rsComment;
//Tokenstatus
            fTokenID := tkComment;
            inc(PosFin, 2);
            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;
//muss der "zwischen"-Operator sein.
        begin
            inc(PosFin);
            fTokenID := tkUnknown;
        end;
    end;
end;
```



```
end;

procedure TSynMiColor.ProcB;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1; //Größe berechnen - 1
  fToIdent := linAct + posIni + 1; //Zeiger auf Bezeichner + 1
  if KeyComp('EGIN') then
    begin
      fTokenID := tkKey;
      StartCodeFoldBlock(nil);
    end else
    if KeyComp('Y')
    then fTokenID := tkKey else
    fTokenID := tkUnknown; //gemeinsamer Bezeichner
end;

procedure TSynMiColor.ProcC;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1; //Größe berechnen - 1
  fToIdent := linAct + posIni + 1; //Zeiger auf Bezeichner + 1
  fTokenID := tkUnknown; //gemeinsamer Bezeichner
end;

procedure TSynMiColor.ProcD;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1; //Größe berechnen - 1
  fToIdent := linAct + posIni + 1; //Zeiger auf Bezeichner + 1
  if KeyComp('E') then fTokenID := tkKey else
  fTokenID := tkUnknown; //gemeinsamer Bezeichner
end;

procedure TSynMiColor.ProcE;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1; //Größe berechnen - 1
  fToIdent := linAct + posIni + 1; //Zeiger auf Bezeichner + 1
  if KeyComp('N')
  then fTokenID := tkKey else
  if KeyComp('ND')
  then
    begin
      fTokenID := tkKey;
      EndCodeFoldBlock();
    end else
    fTokenID := tkUnknown; //gemeinsamer Bezeichner
end;

procedure TSynMiColor.ProcL;
begin
  while Identifiers[linAct[posFin]] do inc(posFin);
  fStringLen := posFin - posIni - 1; //Größe berechnen - 1
  fToIdent := linAct + posIni + 1; //Zeiger auf Bezeichner + 1
  if KeyComp('A')
  then fTokenID := tkKey else
  if KeyComp('OS')
```

```
then fTokenID := tkKey else
  fTokenID := tkUnknown; //ohne Attribute
end;

procedure TSynMiColor.ProcNull;
//Verarbeitet das Auftreten von Zeichen #0
begin
  fTokenID := tkNull;
  //Sie braucht dies nur, um anzuzeigen, dass das Ende der Zeile erreicht ist.
end;

procedure TSynMiColor.ProcSpace;
//Verarbeitet Zeichen, die den Beginn eines Leerzeichens darstellen
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 // fortgesetzter utf8-Subcode
    ((linAct[posFin]<>#0)
    and (fProcTable[linAct[posFin]] = @ProcUnknown)) do
    inc(posFin);
  fTokenID := tkUnknown;
end;

procedure TSynMiColor.SetLine(const NewValue: String; LineNumber: Integer);
begin
  inherited;
  linAct := PChar(NewValue); //Kopieren der aktuellen Zeile
  posFin := 0; //zeigt auf das erste Zeichen
  Next;
end;

procedure TSynMiColor.Next;
begin
  posIni := PosFin; //verweist auf das erste Element
  if fRange = rsComment then
    CommentProc
  else
    begin
      fRange := rsUnknown;
      fProcTable[linAct[posFin]]; //Die entsprechende Funktion wird ausgeführt.
    end;
end;

function TSynMiColor.GetEol: Boolean;
{Zeigt an, wenn das Ende der Zeile erreicht ist.}
begin
  Result := fTokenId = tkNull;
end;
```

```
procedure TSynMiColor.GetTokenEx(out TokenStart: PChar; out TokenLength: integer);
{Gibt Informationen über den aktuellen Token zurück}
begin
  TokenLength := posFin - posIni;
  TokenStart := linAct + posIni;
end;

function TSynMiColor.GetTokenAttribute: TSynHighlighterAttributes;
//Gibt Informationen über den aktuellen Token zurück
begin
  case fTokenID of
    tkComment : Result := fAtriComent;
    tkKey      : Result := fAtriClave;
    tkSpace    : Result := fAtriEspac;
    tkString   : Result := fAtriCadena;
    else Result := nil; //tkUnknown, tkNull
  end;
end;

{Die folgenden Funktionen werden von SynEdit verwendet, um Klammern, geschweifte Klammern
und Anführungszeichen und Hochkommata zu behandeln.
Sie sind nicht entscheidend für Token von Token, aber sie sollten gut reagieren.}

function TSynMiColor.GetToken: String;
begin
  Result := '';
end;

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

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

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);
    end;
  end;
  #10: break;
```

```
#13: break;
    else inc(PosFin);
end;
end;

////////// Implementierung der Bereichsfunktionalitäten //////////
procedure TSynMiColor.ReSetRange;
begin
    inherited;
    fRange := rsUnknown;
end;

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

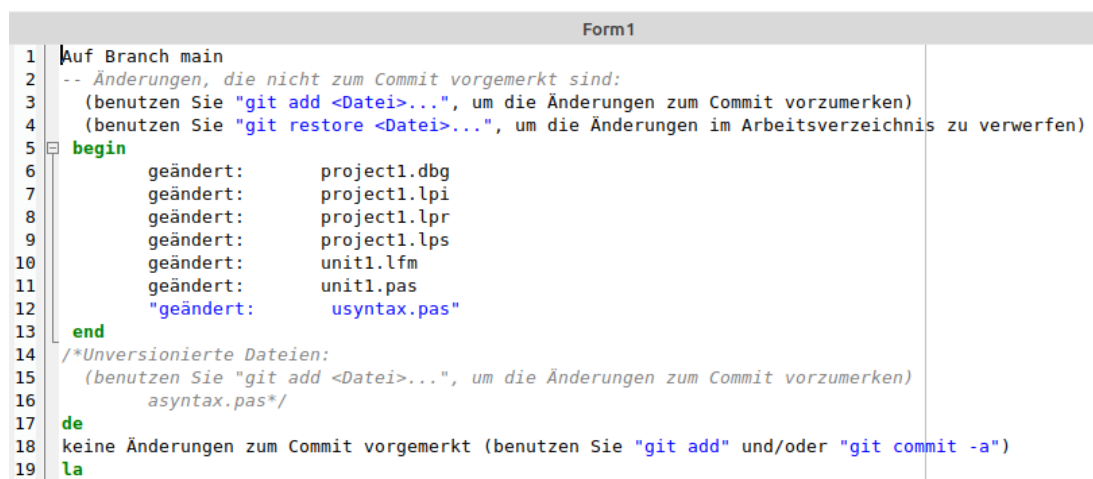
procedure TSynMiColor.SetRange(Value: Pointer);
begin
    inherited;
    fRange := TRangeState(PtrUInt(CodeFoldRange.RangeType));
end;

initialization
CreaTablaIdentif; //Erstellen der Tabelle für die Schnellsuche

end.
```

Aus Platzgründen konnten nicht alle Kommentare aufgenommen werden.

Es könnte so aussehen:



```
1 Auf Branch main
2 -- Änderungen, die nicht zum Commit vorgemerkt sind:
3 (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
4 (benutzen Sie "git restore <Datei>...", um die Änderungen im Arbeitsverzeichnis zu verwerfen)
5 begin
6     geändert:      project1.dbg
7     geändert:      project1.lpi
8     geändert:      project1.lpr
9     geändert:      project1.lps
10    geändert:      unit1.lfm
11    geändert:      unit1.pas
12    "geändert:      usyntax.pas"
13 end
14 /*Unversionierte Dateien:
15 (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
16 asyntax.pas*/
17 de
18 keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/oder "git commit -a")
19 la
```

## 2.5.3 Verbesserte Faltung

So wie das Falten gezeigt wurde, ist es ein recht einfacher Prozess. Das Falten in SynEdit ist jedoch ziemlich vollständig und umfasst zusätzliche Funktionen.

Bisher können wir zusammenfassen, dass zur Steuerung der Faltung die Methoden `StartCodeFoldBlock()` und `EndCodeFoldBlock()` erforderlich sind, die die folgenden Deklarationen haben:

```
function StartCodeFoldBlock(ABlockType : Zeiger;  
    IncreaseLevel : Boolean = true ): TSynCustomCodeFoldBlock ;virtual ;  
procedure EndCodeFoldBlock ( DecreaseLevel : Boolean = True ); virtual ;
```

Die Funktionsweise dieser Methoden ist einfach. `StartCodeFoldBlock()` fügt einen Faltblock hinzu und `EndCodeFoldBlock()` entfernt die zuletzt hinzugefügte Methode.

Darin steckt keine größere Magie. Jeder Aufruf von `EndCodeFoldBlock()` löscht immer die letzte mit `StartCodeFoldBlock()` hinzugefügte Falte.

In der einfachsten Form würden wir einen Faltblock mit der folgenden Formel beginnen:

```
StartCodeFoldBlock (nil);
```

Und um einen Faltblock zu schließen, würden wir das tun:

```
EndCodeFoldBlock ();
```

Diese Arbeitsweise würde für eine einfache Faltung ausreichen. Wir müssen jedoch wissen, welchen Block wir behandeln, um entscheiden zu können, ob er gültig ist oder nicht, und den aktuellen Block schließen. Wir wissen zum Beispiel, dass in Pascal das reservierte Wort `UNTIL` den `REPEAT`-Block abschließt, aber nicht einen `BEGIN`-Block.

Um zu wissen, in welchem Block wir uns gerade befinden, könnten wir eine Hilfsstruktur wie eine Warteschlange erstellen (da Faltblöcke verschachtelt werden können) und immer lesen können, in welchem Block wir uns zuletzt befinden.

Diese Arbeit ist jedoch nicht notwendig, da eine solche Struktur bereits in der Klasse "TSynCustomFoldHighlighter" existiert und speziell für diese Arbeit vorbereitet ist.

Um unseren aktuellen Faltblock zu identifizieren, müssen wir den Parameter "ABlockType" von `StartCodeFoldBlock()` verwenden.

Um dies zu tun, könnte es praktisch sein, eine Liste mit den Bezeichnungen der Faltblöcke in unserem Highlighter zu haben:

```
TMyBlockType = (
    cfbt_BeginEnd,
    cfbt_RepeatUntil,
    cfbt_RecordEnd,
    cfbt_verwendet,
    cfbt_var
);
```

Wenn wir dann die Art des Blocks angeben wollen, würden wir folgenden Code verwenden:

```
StartCodeFoldBlock (Pointer(PtrInt(ABlockType)));
```

Die Typumwandlung ist notwendig, weil der Parameter "ABlockType" vom Typ Zeiger ist. Das Schließen dieses Blocks hat keine Auswirkungen auf andere Blöcke. Ein einfacher Aufruf von EndCodeFoldBlock() würde ausreichen:

```
EndCodeFoldBlock ();
```

Weitere Informationen sind nicht erforderlich, da bekannt ist, dass es sich immer um den letzten Block handelt, der geschlossen wird.

Um zu wissen, welcher der letzte Block im Faltstapel ist, gibt es die Methode TopCodeFoldBlockType():

```
function TopCodeFoldBlockType (DownIndex : Integer = 0): Pointer ;
```

Der Parameter ist optional und sollte im Allgemeinen auf Null belassen werden, es sei denn, Sie möchten einen weiteren innersten Faltungsblock erhalten.

Jetzt, mit den Methoden: StartCodeFoldBlock(), EndCodeFoldBlock() und TopCodeFoldBlockType() haben wir alles, was wir brauchen, um komplexe Faltstrukturen innerhalb eines Highlighters zu erstellen.

Es können nun Typenvergleiche durchgeführt werden:

```
var
    p : Pointer;
...
    p := TopCodeFoldBlockType ; // Lesen des zuletzt eingegebenen Blocks
    // prüft, ob es angebracht ist, den aktuellen Block zu schließen
    if TMyBlockType (PtrUInt(p)) in [cfbt_BeginEnd, cfbt_RecordEnd] then
        EndCodeFoldBlock (DecreaseLevel);
...
```

## 2.5.4 Mehr über Folding

In den Methoden `StartCodeFoldBlock()` und `EndCodeFoldBlock()` gibt es einen Parameter, mit dem die Sichtbarkeit der Faltmarkierung gesteuert werden kann. Dies ist "IncreaseLevel" für `StartCodeFoldBlock()` und "DecreaseLevel" für `EndCodeFoldBlock()`.

Durch den Aufruf von `StartCodeFoldBlock()` mit "IncreaseLevel" auf `FALSE` wird die typische Faltungsmarkierung auf der linken Seite des Editors (Gutter) nicht angezeigt. Die Faltung wird jedoch intern durchgeführt, oder besser gesagt, es werden interne Informationen generiert, wie bei einer sichtbaren Faltung, aber ohne dass der Code gefaltet wird.

Das kann verwirrend sein. Jemand könnte fragen, warum sollte ich so etwas tun? Warum sollte man einen Falz erstellen, der nicht sichtbar ist? Welchen Nutzen hätte ein Faltblock, der nicht gefaltet werden kann?

Die einfache Antwort wäre: Weil es auf diese Weise möglich ist, das Falten zu aktivieren oder zu deaktivieren, ohne die Struktur eines Textes zu verändern.

Wenn wir mit mehreren Faltblöcken arbeiten und es Beziehungen zwischen diesen Blöcken gibt, ist es nicht ratsam, einen dieser Blöcke zu entfernen, da die Struktur der anderen Faltblöcke verändert werden könnte. Wenn Sie also einen Faltblock "eliminieren" wollen, müssen Sie ihn einfach ausblenden, indem Sie "IncreaseLevel" auf `FALSE` setzen, wenn Sie `StartCodeFoldBlock()` aufrufen.

Um konsistent zu sein, sollten Sie "DecreaseLevel" in `EndCodeFoldBlock()` immer auf `FALSE` setzen, wenn der Block mit "IncreaseLevel" auf `FALSE` erstellt wurde.

Lassen Sie uns die Hauptmethoden etwas besser kennenlernen. Die Methode `StartCodeFoldBlock()` hat die folgende Implementierung:

```
function TSynCustomFoldHighlighter.StartCodeFoldBlock ( ABlockType : Pointer ;
    IncreaseLevel : Boolean = True ) : TSynCustomCodeFoldBlock ;
begin
    Result := CodeFoldRange.Add(ABlockType, IncreaseLevel);
end;
```

Die Methode `EndCodeFoldBlock()` hat die folgende Implementierung:

```
procedure TSynCustomFoldHighlighter.EndCodeFoldBlock ( DecreaseLevel : Boolean =
    True );
begin
    CodeFoldRange.Pop(DecreaseLevel);
end;
```

Das "CodeFoldRange"-Objekt funktioniert wie ein Stapel, zu dem Faltinformationen hinzugefügt und entfernt werden.

Die Eigenschaft "CodeFoldRange" ist eine Referenz auf "FCodeFoldRange" der Klasse TSynCustomHighlighterRange:

```
TSynCustomHighlighterRange = class
private
  FCodeFoldStackSize: integer; // EndLevel
  FMinimumCodeFoldBlockLevel: integer;
  FRangeType: Pointer;
  FTop: TSynCustomCodeFoldBlock;
public
  constructor Create(Template: TSynCustomHighlighterRange); virtual;
  destructor Destroy; override;
  function Compare(Range: TSynCustomHighlighterRange): integer; virtual;
  function Add(ABlockType: Pointer = nil; IncreaseLevel: Boolean = True):
    TSynCustomCodeFoldBlock; virtual;
  procedure Pop(DecreaseLevel: Boolean = True); virtual;
  function MaxFoldLevel: Integer; virtual;
  procedure Clear; virtual;
  procedure Assign(Src: TSynCustomHighlighterRange); virtual;
  procedure WriteDebugReport;
  property FoldRoot: TSynCustomCodeFoldBlock read FTop write FTop;
public
  property RangeType: Pointer read FRangeType write FRangeType;
  property CodeFoldStackSize: integer read FCodeFoldStackSize;
  property MinimumCodeFoldBlockLevel: integer
    read FMinimumCodeFoldBlockLevel write FMinimumCodeFoldBlockLevel;
  property Top: TSynCustomCodeFoldBlock read FTop;
end;
```

Dieses Objekt funktioniert wie ein Stapel in Bezug auf die gefalteten Objekte. Die Methode Add() fügt ein neues Element hinzu und Pop() entnimmt das letzte Element.

Die Pop()-Methode hat einen Schutz gegen den Versuch, eine nicht existierende Falte zu löschen. Sie kann also ausgeführt werden, ohne dass ein Überlauf von "CodeFoldRange" zu befürchten ist. Um die Größe des Stacks zu sehen, können Sie jederzeit "CodeFoldStackSize" aufrufen. Dieser Zähler wird nur aktualisiert, wenn StartCodeFoldBlock() mit dem auf TRUE gesetzten Parameter "IncreaseLevel" oder EndCodeFoldBlock() mit dem auf TRUE gesetzten Parameter "DecreaseLevel" aufgerufen wird, was der Standard ist.



## 2.5.5 Aktivieren und Deaktivieren von Faltblöcken

Ein wünschenswertes Merkmal der Faltfunktion ist, dass bestimmte Faltblöcke aktiviert oder deaktiviert werden können.

Im vorangegangenen Abschnitt haben wir gesehen, wie Sie mit den Parametern "IncreaseLevel" und "DecreaseLevel" Faltblöcke im Highlighter deaktivieren können. Jetzt werden wir sehen, wie man einige vordefinierte Strukturen im Highlighter verwenden kann, um die Verwaltung der Aktivierung/Deaktivierung von Faltblöcken zu erleichtern.

Die Klasse "TSynCustomFoldHighlighter" enthält Methoden und Eigenschaften zur Handhabung der so genannten "Folding Settings":

```
TSynCustomFoldHighlighter = class(TSynCustomHighlighter)
protected
// Faltkonfiguration
FFoldConfig: Array of TSynCustomFoldConfig;
function GetFoldConfig(Index: Integer): TSynCustomFoldConfig; virtual;
procedure SetFoldConfig(Index: Integer; const AValue: TSynCustomFoldConfig);
    virtual;
function GetFoldConfigCount: Integer; virtual;
function GetFoldConfigInternalCount: Integer; virtual;
function GetFoldConfigInstance(Index: Integer): TSynCustomFoldConfig; virtual;
procedure InitFoldConfig;
procedure DestroyFoldConfig;
procedure DoFoldConfigChanged(Sender: TObject); virtual;
private
...
protected
...
public
    property FoldConfig[Index: Integer]: TSynCustomFoldConfig
        read GetFoldConfig write SetFoldConfig;
    property FoldConfigCount: Integer read GetFoldConfigCount;
end;
```

Diese Eigenschaften bieten uns eine interne Struktur für die Speicherung der Eigenschaften der Faltkonfiguration und stellen außerdem die Eigenschaften FoldConfig und FoldConfigCount für den Zugriff auf diese Eigenschaften zur Verfügung.

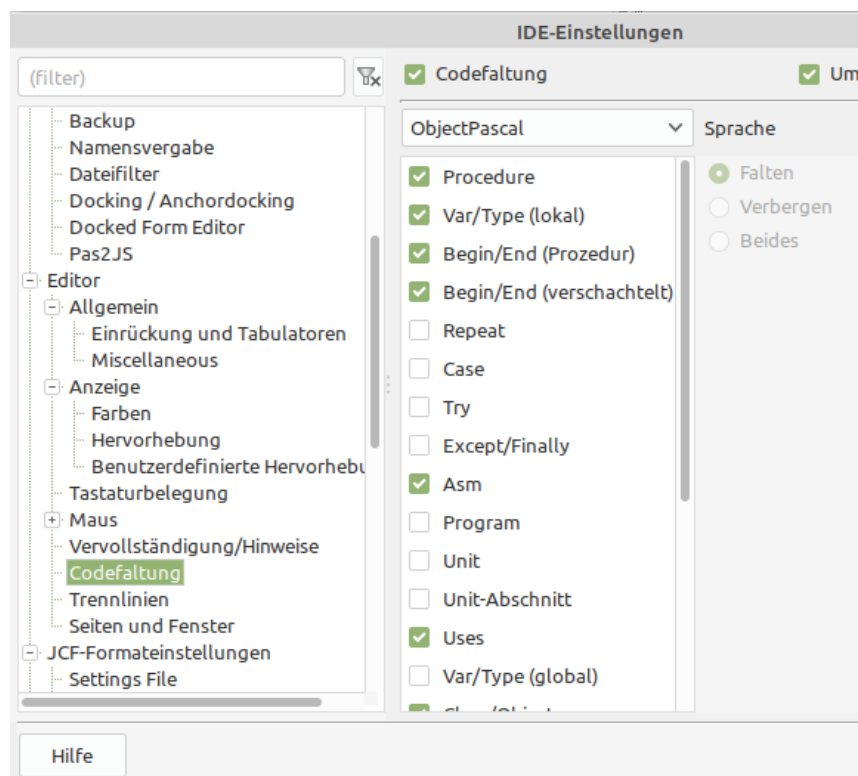
Es ist möglich, jede andere benutzerdefinierte Struktur zu verwenden, aber TSynCustomFoldHighlighter enthält diese Klasse und die Methoden und Eigenschaften, die für ihre Verwaltung notwendig sind, so dass sie uns die Arbeit erspart, die Konfigurationen der Faltbereiche zu verwalten. Da sie außerdem in derselben Klasse TSynCustomFoldHighlighter enthalten sind, sind sie von den Highlightern aus immer zugänglich.

Die Konfigurationen eines Faltbereichs werden in dem Array `FFoldConfig` gespeichert. Ein `FoldConfig`-Objekt ist eine Instanz von `TSynCustomFoldConfig`:

```
TSynCustomFoldConfig = class ( TPersistent )
  privat
    FEnabled : Boolean ;
    FFoldActions : TSynFoldActions ;
    FModes : TSynCustomFoldConfigModes ;
    FOnChange : TNotifyEvent ;
    FSupportedModes : TSynCustomFoldConfigModes ;
  ...
  published
    property Enabled : Boolean read FEnabled write SetFEnabled ;
    property Modes : TSynCustomFoldConfigModes read FModes write SetModes default
      [fmFold] ;
  end;
```

Die wichtigste Eigenschaft ist vielleicht "Aktiviert", denn sie erlaubt es uns zu entscheiden, wann ein Faltbereich aktiviert ist oder nicht.

Diese Struktur ist eine Basis mit den grundlegenden Eigenschaften für einen Faltbereich. Um eine Vorstellung davon zu bekommen, wie sie verwendet werden, können sie in die Lazarus Umgebung gehen und das Menü "Werkzeuge>Einstellungen>Editor>Codefaltung>" wählen:



In diesem Fall werden die Faltblöcke für den Lazarus Pascal Highlighter angezeigt. Wenn Sie eine dieser Boxen aktivieren oder deaktivieren, ändern Sie die "Enabled" Eigenschaft eines der "TSynCustomFoldConfig" Objekte, die im Highlighter existieren.

In der Implementierung der meisten Highlighter mit Folding in Lazarus werden sie sehen, dass, um zu entscheiden, ob der Fold sichtbar gemacht werden soll oder nicht, das FFoldConfig[] Array zuerst konsultiert wird.

```
function TSynLFMSyn.StartLfmCodeFoldBlock(ABlockType:TLfmCodeFoldBlockType):
                                           TSynCustomCodeFoldBlock ;
var
  FoldBlock : Boolean ;
  p : PtrInt ;
begin
  FoldBlock := FFoldConfig[ord ( ABlockType ) ].Enabled ;
  p := 0 ;
  if not FoldBlock then
    p := PtrInt ( CountLfmCodeFoldBlockOffset );
  Result := StartCodeFoldBlock ( p + Pointer ( PtrInt ( ABlockType ) ) ,
                                FoldBlock );
end;

procedure TSynLFMSyn.EndLfmCodeFoldBlock ;
var
  DecreaseLevel : Boolean ;
begin
  DecreaseLevel := TopCodeFoldBlockType < CountLfmCodeFoldBlockOffset ;
  EndCodeFoldBlock ( DecreaseLevel );
end;
```

In diesem Code wird in StartLfmCodeFoldBlock() der Trick angewendet, "p" einen vom Originalwert abweichenden Wert zu geben, wenn eine nicht sichtbare Falte verwendet wird. So können Sie später in EndLfmCodeFoldBlock() den ursprünglichen Wert wiederherstellen und wissen, ob die Falte sichtbar war oder nicht.

Diese Methode ist eine Möglichkeit, zusätzliche Informationen in eine Aufzählung zu kodieren (getarnt als Zeiger). Infolgedessen muss diese Dekodierung auch angewendet werden, wobei TopLfmCodeFoldBlockType() außer Kraft gesetzt wird:

```
function TSynLFMSyn.TopLfmCodeFoldBlockType ( DownIndex : Integer ):
                                           TLfmCodeFoldBlockType ;
var  p : Pointer;
begin
  p := TopCodeFoldBlockType ( DownIndex );
  if p >= CountLfmCodeFoldBlockOffset then
    p := p - PtrUInt ( CountLfmCodeFoldBlockOffset );
  Result := TLfmCodeFoldBlockType ( PtrUInt ( p ));
end;
```

In der Regel wird erwartet, dass es so viele TSynCustomFoldConfig-Objekte gibt, wie es verschiedene Faltblöcke gibt. Zum Beispiel gibt es im SynLFMSyn-Highlighter die folgenden Blöcke:

```
TLfmCodeFoldBlockType = (  
  cfbtLfmObject, // Objekt , geerbt , inline  
  cfbtLfmList, // <>  
  cfbtLfmItem, // Gegenstand  
  cfbtLfmNone  
);
```

Daher müssen die Methoden GetFoldConfigInternalCount() und GetFoldConfigInstance() überschrieben werden, damit die erforderlichen Konfigurationen erstellt werden:

```
function TSynLFMSyn.GetFoldConfigInternalCount : Integer ;  
begin  
  Result:=ord(high(TLfmCodeFoldBlockType))-ord( low(TLfmCodeFoldBlockType))+1;  
end;  
  
function TSynLFMSyn.GetFoldConfigInstance(Index:Integer):TSynCustomFoldConfig;  
begin  
  Result := inherited GetFoldConfigInstance ( Index );  
  Result.Aktiviert := true ;  
end;
```

Diese Methoden werden beim Starten ausgeführt, um die Konfigurationsstrukturen in FFoldConfig[] zu erstellen.

## 2.6 Die Klasse TSynCustomFoldHighlighter

Wie wir gesehen haben, müssen alle Highlighter, die Codefaltung implementieren wollen, von der Klasse TSynCustomFoldHighlighter anstelle von TSynCustomHighlighter abgeleitet werden. Man könnte sagen, dass die Klasse TSynCustomFoldHighlighter ein Nachkomme von TSynCustomHighlighter ist, der lediglich die Funktionalität der Codefaltung hinzufügt. Wie erwartet, fügt die Klassendeklaration die für die Codefaltung notwendigen Informationen und einige nützliche Methoden hinzu, um Informationen über die Faltung zu erhalten.

### 2.6.1 Low Level Folding

Um besser zu verstehen, wie die Faltung in der Klasse implementiert ist, werden wir die Arbeitsmechanik auf einer niedrigen Ebene beschreiben.

Betrachten wir einen typischen Code in Pascal, mit Faltung:

```
procedure primo;  
begin  
    max := 0;  
    if a>b then  
        begin  
            max := a;  
        end  
    else  
        begin  
            max := b;  
        end  
    end;  
end;
```

In diesem Code sehen Sie, dass die gesamte Prozedur innerhalb eines Faltblocks liegt. Der Körper der Prozedur öffnet einen weiteren Faltblock. Darin öffnet die IF-Struktur zwei Faltblöcke: den IF-Body und den ELSE-Body.

Wie üblich können wir einige Zeilen einsparen, indem wir einige Schlüsselwörter in derselben Zeile zusammenfassen:

```
procedure primo;  
begin  
    max := 0;  
    if a>b then begin  
        max := a;  
    end else begin  
        max := b;  
    end  
    end;  
end;
```

In diesem Code endet der Körper des IF in der gleichen Zeile, in der der Körper des ELSE beginnt, daher kann man eine Art "Überlappung" der Blöcke sehen, was aber nicht der Fall ist. Tatsächlich könnte man sagen, dass der Körper des IF nach dem reservierten Wort END "endet" und dass der Körper des ELSE mit dem reservierten Wort BEGIN beginnt.

Wie Faltblöcke können auch sie verschachtelt werden, daher führen wir das Konzept der "Ebene" ein. Wenn wir keine Faltblöcke geöffnet haben, sagen wir, dass wir uns auf Ebene Null befinden. Wenn ein Faltblock geöffnet wird, sagen wir, dass wir uns auf Ebene 1 befinden und so weiter. Aus diesem Grund spricht man im Allgemeinen von "Verschachtelungsebenen" innerhalb eines Faltblocks.






Wie bei der Verwaltung der Bereiche werden aus Gründen der Wirtschaftlichkeit und Einfachheit die Informationen zu den Faltblöcken für jede Zeile gespeichert, z. B. der Zustand des Highlighters, wenn er die Erkundung jeder Zeile beendet.

Um die Code-Faltung korrekt zu verarbeiten, speichert die Klasse `TSynCustomFoldHighlighter` zwei Werte für jede Zeile:

- `EndLevel` - Dies ist die Verschachtelungsebene der Blöcke am Ende der Zeile.
- `MinLevel` - Dies ist die kleinste Verschachtelungsebene in der Zeile.

Diese Variablen und weitere werden als Teil der Informationen gespeichert, die der Highlighter in jeder Zeile des Editors speichert.

Eine visuelle Übung wird uns helfen zu verstehen, wie die Code-Faltung innerhalb von `TSynCustomFoldHighlighter` abläuft:

|   | <b>EndLevel</b> | <b>MinLevel</b> |
|---|-----------------|-----------------|
|  <b>procedure</b> primero;       | 1               | 0               |
|  <b>begin</b>                    | 2               | 1               |
| max := 0;   | 2               | 2               |
|  <b>if</b> a>b <b>then begin</b> | 3               | 2               |
| max := a;   | 3               | 3               |
|  <b>end else begin</b>           | 3               | 2               |
| max := b;   | 3               | 3               |
| <b>end;</b>   | 2               | 2               |
|  <b>end;</b>                     | 0               | 0               |

In der ersten Zeile ist der Mindestpegel 0, da vor dem reservierten Wort `PROCEDURE` noch kein Block geöffnet wurde.

In der zweiten Zeile wird ein neuer Block mit dem reservierten Wort `BEGIN` eröffnet. Für die Zwecke der Codefaltung ist es nicht so wichtig, ob der Block vor oder nach `BEGIN` beginnt.

Die anderen Zeilen folgen der gleichen Logik.

Diese Information ist alles, was der Editor benötigt, um die Faltmarken im Seitenpanel anzuzeigen. Immer wenn `EndLevel > MinLevel` gefunden wird, bedeutet dies, dass ein oder mehrere Faltblöcke geöffnet wurden, und die Blockmarkierung sollte angezeigt werden. Wenn `EndLevel` in der vorherigen Zeile größer ist als `MinLevel` in einer Zeile, bedeutet dies, dass der Block der vorherigen Zeile geschlossen wurde. Diese Information ermöglicht es Ihnen, die Zeilen schnell zu durchsuchen, um Informationen über die Faltblöcke zu erhalten.

Natürlich verarbeitet `TSynCustomFoldHighlighter` mehr Informationen, um mit den Blöcken umzugehen, aber für die Anzeige von Markierungen würde diese Information ausreichen, und sie ist Teil der Informationen, die in jeder Zeile des Editors gespeichert werden.

Es mag den Anschein erwecken, dass Faltungsinformationen verloren gehen, wenn ein Block in derselben Zeile geöffnet und geschlossen wird, aber es ist zu bedenken, dass dieser Block für die Zwecke der Codefaltung irrelevant ist.

Um die Werte für `EndLevel` und `MinLevel` anzuzeigen, können Sie auf die Eigenschaften zugreifen:

```
function TSynCustomFoldHighlighter.FoldBlockEndLevel ( ALineIndex : TLineIdx ;  
const AFilter : TSynFoldBlockFilter ) : integer ;  
  
function TSynCustomFoldHighlighter.FoldBlockMinLevel ( ALineIndex : TLineIdx ;  
const AFilter : TSynFoldBlockFilter ) : integer ;
```

Es muss nur die gewünschte Zeilennummer (beginnend bei Null) übergeben werden. Das Feld "AFilter" wird nicht verwendet und sollte daher auf NIL belassen werden.

## 2.6.2 CurrentLines[] und CurrentRanges[]

Diese `TSynCustomFoldHighlighter`-Eigenschaften geben mehr oder weniger die gleichen Informationen zurück, die auch `TSynCustomHighlighter` zurückgeben würde.

`CurrentLines[]`, ist genau dasselbe und ermöglicht den Zugriff auf die Zeilen des aktuellen Editors. Bis dahin funktioniert alles gut.

`CurrentRanges[]` hat sich jedoch geändert. Es ist jetzt ein Zeiger auf Objekte des Typs `TSynCustomHighlighterRange`.

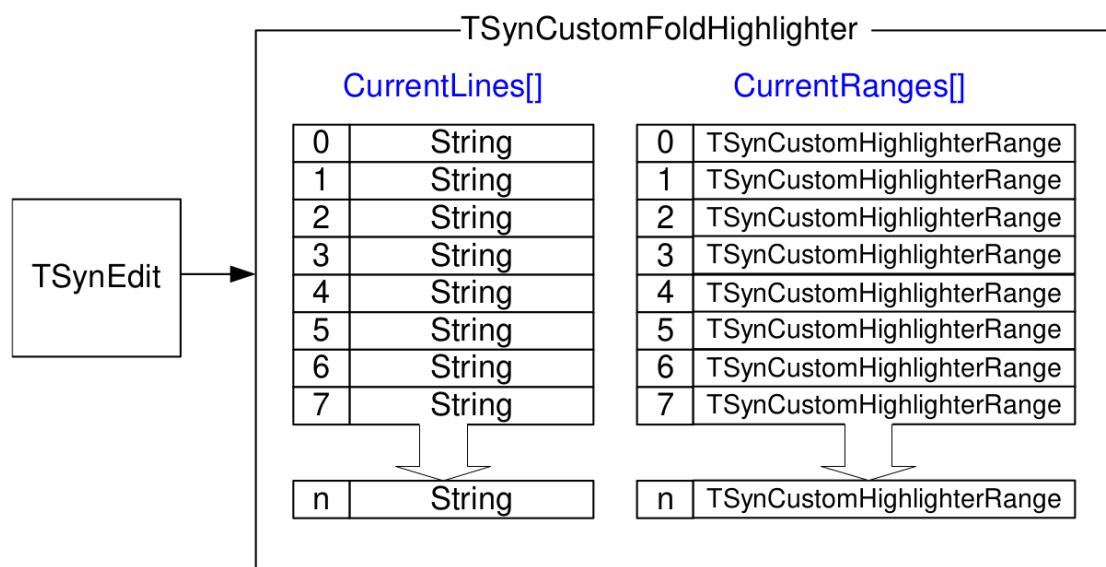
Die Tatsache, dass `TSynCustomFoldHighlighter` nicht nur Bereichsinformationen speichert, macht es notwendig, eine Möglichkeit zu haben, zusätzliche Informationen zu jeder Zeile des Editors hinzuzufügen.

Die Handhabung von Faltblöcken erfordert zusätzliche Informationen, daher wurde eine spezielle Struktur geschaffen, die es ermöglicht, die Bereichsinformationen (denselben Zeiger wie immer) und zusätzlich andere Felder zu verpacken.

Aus diesem Grund gibt es die Klasse `TSynCustomHighlighterRange`, die wie folgt definiert ist:

```
TSynCustomHighlighterRange = class
private
  FCodeFoldStackSize: integer; // EndLevel
  FMinimumCodeFoldBlockLevel: integer;
  FRangeType: Pointer;
  FTop: TSynCustomCodeFoldBlock;
public
  constructor Create(Template: TSynCustomHighlighterRange); virtual;
  destructor Destroy; override;
  function Compare(Range: TSynCustomHighlighterRange): integer; virtual;
  function Add(ABlockType: Pointer = nil; IncreaseLevel: Boolean = True):
    TSynCustomCodeFoldBlock; virtual;
  procedure Pop(DecreaseLevel: Boolean = True); virtual;
  function MaxFoldLevel: Integer; virtual;
  procedure Clear; virtual;
  procedure Assign(Src: TSynCustomHighlighterRange); virtual;
  procedure WriteDebugReport;
  property FoldRoot: TSynCustomCodeFoldBlock read FTop write FTop;
public
  property RangeType: Pointer read FRangeType write FRangeType;
  property CodeFoldStackSize: integer read FCodeFoldStackSize;
  property MinimumCodeFoldBlockLevel: integer
    read FMinimumCodeFoldBlockLevel write FMinimumCodeFoldBlockLevel;
  property Top: TSynCustomCodeFoldBlock read FTop;
end;
```

Das bedeutet, dass wir jetzt in `TSynCustomFoldHighlighter`, wenn `CurrentRanges[]` für eine Zeile gelesen wird, nicht direkt den Bereichswert lesen, sondern den Verweis auf ein `TSynCustomHighlighterRange`-Objekt erhalten, das zusätzliche Informationen (eine Menge zusätzlicher Informationen) enthält, die für die Handhabung der Faltblöcke notwendig sind. Das folgende Diagramm verdeutlicht die Situation besser:





Die Bereichsinformationen werden nun zu einem weiteren bescheidenen Feld dieses neuen Objekts. Das Feld ist `RangeType`. Das ist der Grund, warum in einem Highlighter mit Faltung, wenn die Methoden implementiert sind:

```
TSynFacilSyn.GetRange: Zeiger;  
TSynFacilSyn.SetRange(Value: Pointer);
```

müssen Sie mit dem Feld `"CodeFoldRange.RangeType"` arbeiten, anstatt mit dem direkten Wert der Funktion.

`CurrentRanges[]` speichert immer noch Zeiger, wie es in `TSynCustomHighlighter` der Fall ist, aber jetzt sind diese Zeiger Referenzen auf Objekte des Typs `TSynCustomFoldHighlighterRange`.

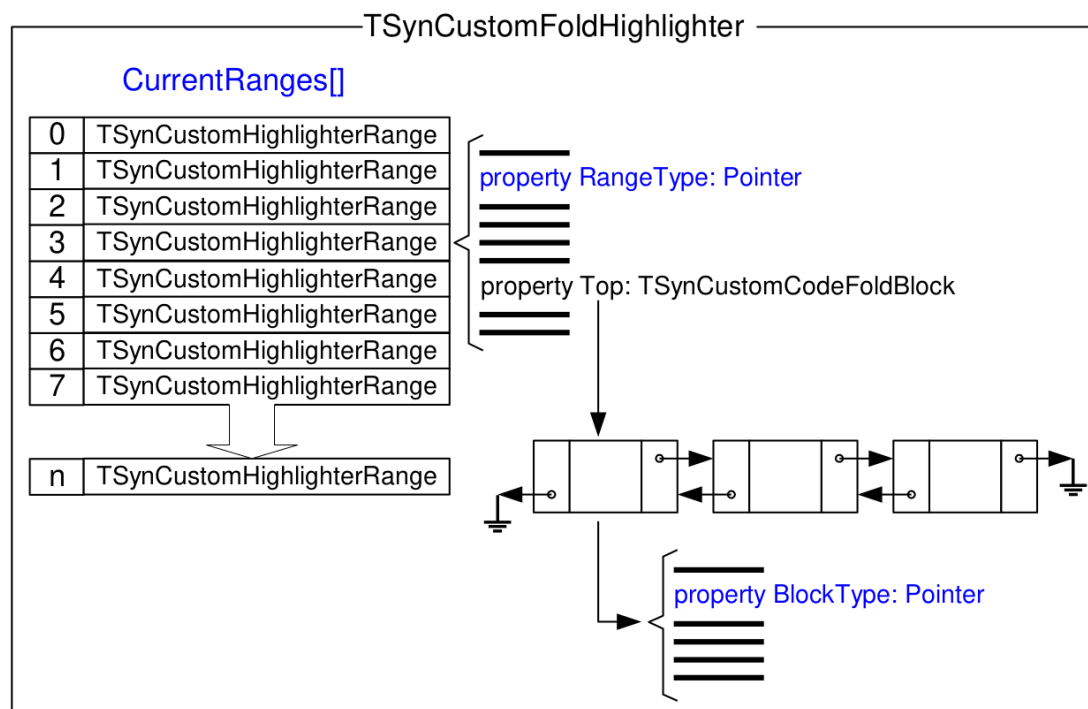
So können wir beim Lesen von `CurrentRanges[]` eine Typumwandlung vornehmen:

```
p := TSynCustomHighlighterRange ( CurrentRanges[SynEdit1.CaretY - 1] )
```

Die wohl wichtigsten Eigenschaften eines `TSynCustomFoldHighlighterRange`-Objekts sind `RangeType` und `Top`.

Die Eigenschaft `Top` ist ein Verweis auf ein `TSynCustomCodeFoldBlock`-Objekt, das den zuletzt geöffneten Folding Block am Ende der entsprechenden Zeile von `CurrentRanges[]` darstellt. Ein `TSynCustomCodeFoldBlock`-Objekt ist nicht der eigentliche Faltblock (den wir mit `StartCodeFoldBlock()` geöffnet haben), sondern ein Objekt, das als Knoten einer verknüpften Liste fungiert.

Das liegt daran, dass alle mit `StartCodeFoldBlock()` geöffneten Blöcke in einer verknüpften Liste gespeichert werden und nicht in einem LIFO-Stapel, wie man vielleicht erwarten würde.



Die Methoden Add() und Pop() von TSynCustomHighlighterRange ermöglichen das Hinzufügen oder Entfernen von Knoten in der Liste.

Jeder Knoten in der Liste ist ein Objekt des Typs TSynCustomCodeFoldBlock, der die folgende Definition hat:

```
TSynCustomCodeFoldBlock = class
private
  FBlockType: Pointer;
  FParent, FChildren: TSynCustomCodeFoldBlock;
  FRight, FLeft: TSynCustomCodeFoldBlock;
  FBalance: Integer;
  function GetChild(ABlockType: Pointer): TSynCustomCodeFoldBlock;
protected
  function GetOrCreateSibling(ABlockType:Pointer):TSynCustomCodeFoldBlock;
  property Right: TSynCustomCodeFoldBlock read FRight;
  property Left: TSynCustomCodeFoldBlock read FLeft;
  property Children: TSynCustomCodeFoldBlock read FChildren;
public
  destructor Destroy; override;
  procedure WriteDebugReport;
public
  procedure InitRootBlockType(AType: Pointer);
  property BlockType: Pointer read FBlockType;
  property Parent: TSynCustomCodeFoldBlock read FParent;
  property Child[ABlockType:Pointer]:TSynCustomCodeFoldBlock read GetChild;
end;
```

Die Verknüpfungen zu den anderen Knoten sind in den Eigenschaften "Parent" und "Children" zu finden.

Die wichtigsten Daten, die dieser Knoten enthält, sind BlockType, d.h. der Verweis auf den Faltblock, den wir angeben, wenn wir StartCodeFoldBlock() verwenden. Für jeden Knoten, d.h. für jeden offenen Block, gibt es einen BlockType.

Wie bei RangeType kann der BlockType-Zeiger auf ein reales Objekt verweisen oder eine Ganzzahl oder Aufzählung sein, die in einem Datenzeigertyp kodiert ist. Dies hängt davon ab, wie der Highlighter, der die Codefaltung nutzt, implementiert ist.

### 2.6.3 Einige Methoden und Eigenschaften

TSynCustomFoldHighlighter verfügt über mehrere Methoden, auf die von innerhalb und außerhalb der Klasse zugegriffen werden kann.

Die Methode FoldEndLine(), mit der Erklärung:

```
function TSynCustomFoldHighlighter.FoldEndLine(ALineIndex, FoldIndex: Integer):  
integer ;
```

ermöglicht die Rückgabe der letzten Zeile des Blocks am Ende der ALineIndex-Zeile (die für die erste Zeile bei 0 beginnt). Mit dem zweiten Parameter können Sie die Ebene des zu verwendenden Blocks angeben. Für den Block der höchsten Ebene sollte er auf Null belassen werden.

Betrachten wir den folgenden Code:

```
1  [ ] procedure primo;  
2  [ ] begin  
3  |   max := 0;  
4  [ ]   if a>b then begin  
5  |       max := a;  
6  [ ]   end else begin  
7  |       max := b;  
8  |   end;  
9  [ ] end;
```

Wenn wir die Funktion FoldEndLine(1,0) anwenden, erhalten wir die Zahl 8, weil der höchste offene Block am Ende von Zeile 2 in Zeile 9 endet.

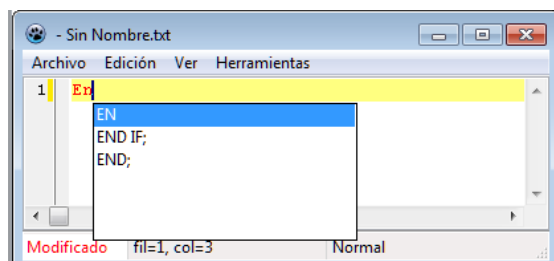
Die Methode FoldLineLength() gibt die Anzahl der Zeilen des Blocks zurück, der am Ende der angegebenen Zeile geöffnet ist. Sie hat die gleichen Parameter wie FoldEndLine():

```
function TSynCustomFoldHighlighter.FoldLineLength(ALineIndex, FoldIndex: Integer):  
integer ;
```

Ebenso beginnt ALineIndex bei Null für Zeile 1.

## 2.7 Autovervollständigen

Autovervollständigen ist die Funktion von SynEdit, die es ermöglicht, während der Eingabe eine Liste von Wörtern anzuzeigen, die wir auswählen können, um den fehlenden Text im aktuellen Wort zu ergänzen. Auf diese Weise ersparen wir uns die Mühe, das ganze Wort zu schreiben.



Autovervollständigung in SynEdit ist nicht so weit entwickelt wie die Folding- oder Syntax-Highlighting-Funktion. Es gibt grundlegende Funktionen, aber sie erfüllt ihre Aufgabe.

Wie einige andere Funktionen von SynEdit ist es möglich, die Autovervollständigung mit Hilfe von Komponenten ("TSynCompletion") aus der Komponentenleiste zu implementieren, oder Sie können Code verwenden, um sie zu erstellen.

Für die Autovervollständigung wurde folgende *Arbeitsweise* festgelegt:

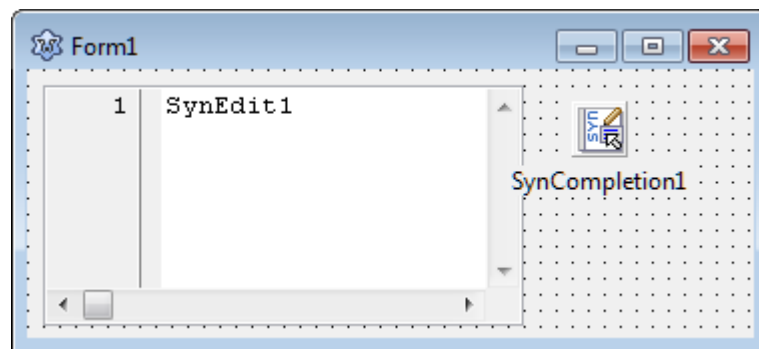
- Er ist immer mit einem Editor des Typs SynEdit verbunden. Diese Zuordnung kann im Entwurf oder durch Code erfolgen.
- Sie wird jedes Mal aktiviert, wenn die mit dem Abschluss verbundene Tastenkombination gedrückt wird. Die am häufigsten verwendete Kombination ist <Strg> + <Leertaste>.
- Nach der Aktivierung sollte an der Cursorposition ein Fenster mit einer Liste von Wörtern (oder Sätzen) erscheinen, von denen eines ausgewählt werden muss.
- Wenn das Auswahlfenster angezeigt wird, wird das aktuelle Wort, das Wort vor dem Cursor, als Arbeitswort verwendet. Dies ist das Wort, das ersetzt werden soll.
- Wenn bei der Anzeige des Auswahlfensters nur eine Option vorhanden ist, wird das Arbeitswort automatisch durch diese einzige Option ersetzt und das Auswahlfenster geschlossen.
- Wenn das Auswahlfenster angezeigt wird, können Sie immer noch im Editor schreiben, aber das Auswahlfenster übernimmt die Kontrolle über einige Tasten, z. B. die Pfeiltasten. Außerdem ist es nicht möglich, das Arbeitswort zu löschen, da es mindestens ein Zeichen enthalten muss.
- In der angezeigten Liste können Sie mit den Pfeiltasten, <Seite hoch>, <Seite runter>, <Home> oder <Ende> durch die Optionen blättern. Solange die Liste der Optionen angezeigt wird, wirken diese Tasten nicht mehr auf den Editor.
- Um eine der Optionen auszuwählen, müssen Sie die <Enter>-Taste, <Leertaste> oder ein Symbol wie '+' oder '-' drücken.
- Wenn Sie eine der Optionen aus der Liste auswählen, wird das aktuelle Wort durch das ausgewählte Wort ersetzt, und das Auswahlfenster wird ausgeblendet.
- WENN Sie keine Option im Optionsfenster auswählen wollen, müssen Sie <escape> drücken, damit das Optionsfenster verschwindet und Sie mit der normalen Bearbeitung fortfahren können.
- Das Optionsfenster kann nur durch Auswahl eines seiner Elemente oder durch Drücken von <escape> ausgeblendet werden.

Dies ist das normale Verhalten der Autovervollständigungsfunktion. Um dieses Verhalten zu ändern, müssen Sie die Eigenschaften der Vervollständigungskomponente (TSynCompletion) ändern, entweder über das Entwurfsfenster oder über Code.

Größere Änderungen müssen vollständig per Code vorgenommen werden.

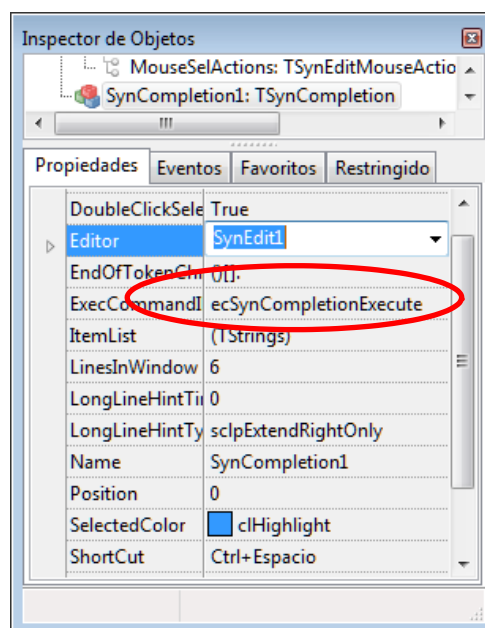
### 2.7.1 Autovervollständigung mit Komponente

Die Vorgehensweise ist einfach. Der TSynEdit-Editor wird natürlich zum Formular hinzugefügt, und dann wird auch die Komponente "TSynCompletion" hinzugefügt:

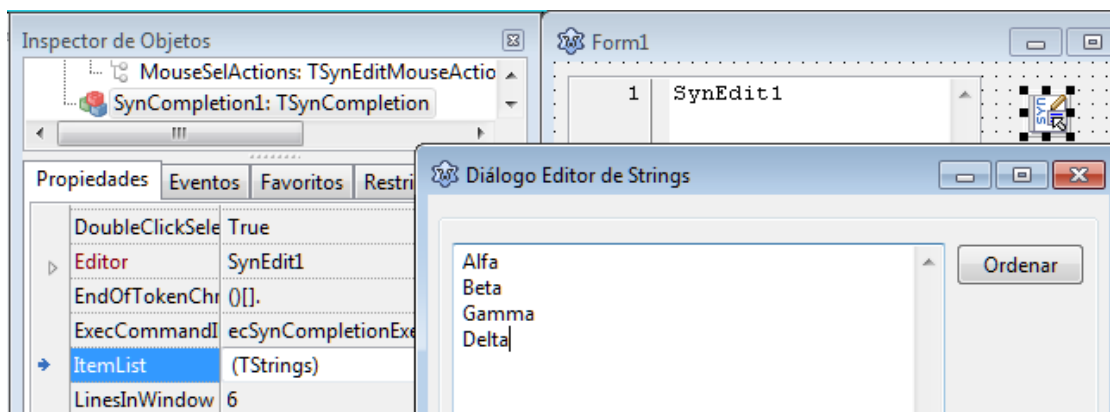


Die Komponente "TSynCompletion" befindet sich in der Komponentenpalette, wo auch "TSynEdit" zu finden ist.

Sobald wir die funktionierenden Komponenten haben, müssen wir "SynCompletion1" mit "SynEdit1" verknüpfen. Dazu konfigurieren Sie die Eigenschaft "Editor" von "SynCompletion1" im Objekt-Explorer so, dass sie auf "SynEdit1" verweist:



Beachten Sie, dass die Standardeigenschaft für "ShortCut" "Strg + Leertaste" ist, was bedeutet, dass diese Kombination die Dropdown-Liste für die automatische Vervollständigung aktivieren wird. Sobald diese Eigenschaft zugeordnet ist, ist die Funktionalität im Editor einsatzbereit. Sie müssen jedoch noch die Liste der Wörter hinzufügen, die im Dropdown-Menü angezeigt werden sollen. Zu diesem Zweck können wir die Eigenschaft "ItemList" direkt im Objektinspektor ändern:



Wenn das Programm nun ausgeführt wird und die Tastenkombination <Strg> + <Leertaste> gedrückt wird, erscheint eine Dropdown-Liste mit den angegebenen Wörtern, die es uns ermöglicht, das aktuelle Wort durch eines der Wörter aus der Liste zu ersetzen.

Diese Arbeitsmethode ist einfach und schnell zu implementieren, erlaubt aber keine weitergehenden Arbeitsoptionen wie das Filtern der Liste nach dem geschriebenen Wort. Für diese und andere Funktionalitäten ist es notwendig, mit Code zu arbeiten.

## 2.7.2 Autovervollständigen mit Code

Für die automatische Vervollständigung muss ein Objekt der Klasse "TSynCompletion" erstellt werden:

```
MenuContext : TSynCompletion ; // Kontextmenü
```

Die Initialisierung ist einfach:

```
MenuContext := TSynCompletion.Create ( Self0 ); // Menü erstellen
MenuContext.Editor := ed ; // Editor zuweisen
MenuContext.ShortCut := Menus.ShortCut ( VK_SPACE, [ssCtrl] ); // Tastaturkürzel zuweisen
```

Unter normalen Umständen wird das Vervollständigungsfenster nur aktiviert, wenn die Tastenkombination gedrückt wird. Wenn Sie das Fenster zu einem anderen Zeitpunkt anzeigen möchten, können Sie die Ausführungsmethode verwenden:

Die folgende Prozedur zeigt das Vervollständigungsfenster "TSynCompletion1" für den "ed"-Editor an:

```
procedure Mostrar;  
var p:TPoint;  
begin  
  P := Point(ed.CaretXPix,ed.CaretYPix + ed.LineHeight);  
  P.X:=Max(0,Min(P.X, ed.ClientWidth - TSynCompletion1.Width));  
  P := ed.ClientToScreen(p);  
  // Kontextmenü öffnen. Es wird nur angezeigt, wenn es Einträge enthält.  
  SynCompletion1.Execute(' ', p.x, p.y);  
end;
```

Die Berechnung von px und py erfolgt so, dass das Fenster an der Cursorposition erscheint. Es ist wichtig zu beachten, dass das Fenster nur angezeigt wird, wenn "TSynCompletion1" Elemente in seiner Liste hat.

Um Elemente zu "TSynCompletion1" hinzuzufügen, müssen Sie die Eigenschaft "ItemList" verwenden, die eine "StringList" ist:

```
SynCompletion1.ItemList.add ('Alpha' );  
SynCompletion1.ItemList.add ('Beta' );  
SynCompletion1.ItemList.add ('Gamma' );
```

Die hinzugefügten Zeichenfolgen sind diejenigen, die angezeigt werden, wenn das Fenster "TSynCompletion" angezeigt wird.

Wenn die Wortliste fest sein soll, dann kann sie nur einmal beim Programmstart gefüllt werden.

Wenn die Liste vom aktuellen Wort oder von anderen Informationen abhängt, können Sie eines der "TSynCompletion"-Ereignisse verwenden. Das betreffende Ereignis ist "OnExecute". Dieses Ereignis wird ausgeführt, bevor die Wortliste zur Vervollständigung geöffnet wird. Hier können Sie es verwenden, um die auszufüllende Wortliste auszuwählen. Als zusätzliche Hilfe kann die Zeichenkette "SynCompletion1.CurrentString" verwendet werden. Hier wird der Bezeichner gespeichert, der sich vor dem Cursor befindet, also das Arbeitswort.

Das Ereignis "OnExecute" wird unabhängig davon ausgelöst, ob die Liste durch eine Tastenkombination oder mit der Methode "Execute" aktiviert wurde. Aber nur bei der Aktivierung durch eine Tastenkombination wird die Zeichenkette "CurrentString" automatisch aktualisiert. Wenn "Execute" verwendet wird, ist der erste Parameter der Anfangswert, der "CurrentString" zugewiesen wird.

Bei jedem Tastendruck, während das Auswahlfenster aktiv ist, wird die Zeichenkette "CurrentString" aktualisiert.

Ein weiteres wichtiges Ereignis ist "OnSearchPosition". Dieses Ereignis wird jedes Mal aufgerufen, wenn eine Taste gedrückt wird, während das Fenster "TSynCompletion" geöffnet ist. Als Parameter wird eine ganze Zahl übergeben, die das ausgewählte Element angibt. Das erste Element ist 0. Dieser Wert kann innerhalb des Ereignisses geändert werden, um ein anderes Element zu selektieren. Wenn Sie dieses Ereignis nicht verwenden, wird standardmäßig das Element ausgewählt, das mit "CurrentString" übereinstimmt.

Die Eigenschaft "CurrentString" gibt das aktuelle Wort an, das sich hinter dem Cursor befindet. Es spielt keine Rolle, ob sich Zeichen vor dem Cursor befinden, "CurrentString" berücksichtigt nur die vorherigen Zeichen, bis es ein Leerzeichen findet. Diese Eigenschaft wird nur beim Aufruf von "TSynCompletion" über die Tastatur aktualisiert. Wird sie mit "Execute()" geöffnet, beginnt "CurrentString" leer.

## 3 Anhang

### 3.1 Algorithmen für die Implementierung von Highlightern.

Sie können auf verschiedene Weise implementiert werden. Das Hauptziel besteht darin, eine schnelle Antwort zu erhalten, und sie konzentrieren sich hauptsächlich auf den schnellen Vergleich von Zeichenketten. Die Algorithmen, die am häufigsten getestet wurden, sind:

#### 3.1.1 Algorithmus, der auf der Verwendung von Hash-Funktionen basiert

Dies ist der Algorithmus, der verwendet wurde, um die meisten der vordefinierten Highlighter zu implementieren, die mit Lazarus geliefert werden. Er ist schnell zu verarbeiten, hat aber eine komplizierte Implementierung.

Es arbeitet mit einer Funktionstabelle "fProcTable[]", die dazu dient, jedes Zeichen je nach Typ an eine bestimmte Funktion zu leiten, die für die Identifizierung des analysierten Tokens zuständig ist. Das Ausfüllen dieser Tabelle erfolgt vor der Verwendung des Highlighters mit der Funktion MakeMethodTables():

```
procedure TSynPerlSyn.MakeMethodTables ;
var
  I : Char ;
begin
  for I := #0 to #255 do
    case I of
      #0 : fProcTable[I] := @NullProc ;
      #1..#9,#11,#12,#14..#32 : fProcTable[I] := @SpaceProc ;
      #10 : fProcTable[I] := @LFProc ;
      #13 : fProcTable[I] := @CRProc ;
      ' ' : fProcTable[I] := @ColonProc ;
      '#' : fProcTable[I] := @CommentProc ;
      '=' : fProcTable[I] := @EqualProc ;
      '>' : fProcTable[I] := @GreaterProc ;
      '<' : fProcTable[I] := @LowerProc ;
      '0'..'9', '.' : fProcTable[I] := @NumberProc ;
      '$', 'A'..'Z', 'a'..'z', '_' : fProcTable[I] := @IdentProc ;
      '-' : fProcTable[I] := @MinusProc ;
      '+' : fProcTable[I] := @PlusProc ;
      '/' : fProcTable[I] := @SlashProc ;
      '*' : fProcTable[I] := @StarProc ;
      #34 : fProcTable[I] := @StringInterpProc ;
      #39 : fProcTable[I] := @StringLiteralProc ;
    else
      fProcTable[I] := @UnknownProc ;
    end;
  end;
end;
```



Hier werden alphabetische Zeichen immer als Anfang von Bezeichnern betrachtet und an die Funktion IdentProc() weitergeleitet, die für die Extraktion des Bezeichners und die Überprüfung, ob es sich um ein Schlüsselwort handelt, zuständig ist.

Um Schlüsselidentifikatoren aufzuspüren, gehen Sie wie folgt vor:

Das dem Bezeichner entsprechende Token wird extrahiert und das Ergebnis seiner Hash-Funktion berechnet<sup>13</sup>. Mit diesem Wert wird eine Funktionstabelle mit der Bezeichnung fIdentFuncTable[] adressiert, deren Einträge nur in den Hash-Funktionswerten aktiviert sein müssen, die den Schlüsselwörtern entsprechen.

Die Tabelle fIdentFuncTable[] wird in der InitIdent-Methode ausgefüllt und muss vor der Verwendung des Highlighters erstellt werden:

```
procedure TSynPerlSyn.InitIdent ;
var
  I : integer;
begin
  for I := 0 to 2167 do
    case I of
      109 : fIdentFuncTable[I] := @Func109 ;
      113 : fIdentFuncTable[I] := @func113 ;
      196 : fIdentFuncTable[I] := @func196 ;
      201 : fIdentFuncTable[I] := @func201 ;
      204 : fIdentFuncTable[I] := @func204 ;
      207 : fIdentFuncTable[I] := @func207 ;
      209 : fIdentFuncTable[I] := @func209 ;
      211 : fIdentFuncTable[I] := @func211 ;
      230 : fIdentFuncTable[I] := @func230 ;
      ...
    else
      fIdentFuncTable[I] := @AltFunc ;
    end;
  end;
end;
```

Die angesprochenen Funktionen haben die Form "funcXXX", wobei XXX der entsprechende "Hash"-Wert ist. Für eine Gruppe von Schlüsselwörtern gibt es nur eine bestimmte Anzahl von Hash-Werten in diesem Bereich. Die anderen Einträge geben nur den Wert "tkIdentifier" (über AltFunc) zurück, was anzeigt, dass es sich um einen einfachen Bezeichner handelt. Gibt es mehrere Schlüsselwörter, die den gleichen Hash-Wert haben, wird ein zusätzlicher Vergleich innerhalb der entsprechenden Funktion durchgeführt:

```
function TSynPerlSyn.Func230 : TtkTokenKind ;
begin
  if KeyComp ( 'tr' ) then Result := tkKey else
    if KeyComp ( 'my' ) then Result := tkKey else Result := tkIdentifier ;
  end;
```

<sup>13</sup> Diesen Wert erhält man, indem man eine Tabelle erstellt, die jedem Buchstaben des englischen Alphabets einen Wert zuordnet (normalerweise in der mHashTable[], und sie mit der Funktion "MakeIdentTable" füllt). Anhand dieser Tabelle wird der jedem Schlüsselwort entsprechende Wert berechnet, indem der Wert jedes Buchstabens im Bezeichner addiert wird. Der so erhaltene Wert hängt in der Regel von den Buchstaben des Bezeichners und seiner Größe ab, überschreitet aber bei normaler Syntax in der Regel nicht 200. Dieser Wert ist nicht für jedes Wort eindeutig, da mehrere verschiedene Wörter denselben Wert haben können, aber er ermöglicht es Ihnen, die Bezeichner effektiv zu kategorisieren, um die Suche auf eine viel kleinere Gruppe zu beschränken.

### 3.1.2 Algorithmus auf der Grundlage des ersten Zeichens als Präfix.

Dieser Algorithmus ist derjenige, den wir in diesem Dokument beschreiben, und basiert auf der Verwendung des ersten Zeichens eines Bezeichners als Präfix, um die Erkennung von Schlüsselwörtern zu beschleunigen. Es verwendet auch eine Funktionstabelle "fProcTable[]", die dazu dient, jedes Zeichen je nach seinem Typ einer bestimmten Funktion zuzuweisen, die für die Identifizierung des analysierten Tokens zuständig ist. Der Unterschied besteht darin, dass diese Tabelle auf eine spezifische Funktion für jedes alphabetische Zeichen verweist:

```
procedure TSynMiColorSF.MakeMethodTables ;
var
  I : Char ;
begin
  for I := #0 to #255 do
    case I of
      ' - ' : fProcTable[I] := @ProcMinus ;
      ' / ' : fProcTable[I] := @ProcSlash ;
      #39 : fProcTable[I] := @ProcString ;
      '"' : fProcTable[I] := @ProcString2 ;
      '0' .. '9' : fProcTable[I] := @ProcNumber ;
      'A' , 'a' : fProcTable[I] := @ProcA ;
      'B' , 'b' : fProcTable[I] := @ProcB ;
      'C' , 'c' : fProcTable[I] := @ProcC ;
      'D' , 'd' : fProcTable[I] := @ProcD ;
      ...
      'Z' , 'z' : fProcTable[I] := @ProcZ ;
      '_' : fProcTable[I] := @ProcUnder ;
      '$' : fProcTable[I] := @ProcMacro ;
      #13 : fProcTable[I] := @ProcCR ;
      #10 : fProcTable[I] := @ProcLF ;
      #0 : fProcTable[I] := @ProcNull ;
      #1..#9, #11, #12, #14..#32 : fProcTable[I] := @ProcSpace ;
      else fProcTable[I] := @ProcUnknown ;
    end;
  end;
end;
```

Dann führt jede Funktion vom Typ ProcA() oder ProcB() direkt die Vergleiche durch, um die Schlüsselwörter zu identifizieren:

```
Procedure TSynMiColorSF.ProcA ;
begin
  while Identifizier[fLine[Lauf]] do inc ( Lauf );
  fStringLen := Run - fTokenPos - 1 ; // Größe berechnen - 1
  fToIdent := Fline + fTokenPos + 1 ; // Zeiger auf Bezeichner + 1
  if KeyComp ( 'nd' ) then fTokenID := tkKey else
  if KeyComp ( 'rray' ) then fTokenID := tkKey else
    fTokenID := tkIdentifizier ; // gemeinsamer Bezeichner
end;
```

Da das erste Zeichen der Kennung bereits bekannt ist, wird der Vergleich mit einem Zeichen weniger durchgeführt, was den Vergleich beschleunigt.

### 3.1.3 Vergleich zwischen Algorithmen

Um die Leistung beider Algorithmen zu überprüfen, habe ich zwei Vergleichsreihen durchgeführt. Einer davon mit dem Lazarus 1.0.12 PHP-Highlighter und der andere mit dem Perl-Highlighter. Beide Highlighter (implementiert mit dem Hash-Algorithmus) wurden hinsichtlich ihrer Geschwindigkeit mit einem Highlighter verglichen, der mit dem First Character as Prefix-Algorithmus implementiert wurde.

Dies waren die Ergebnisse:

|  |   |
|--|---|
| PHP-Highlighter (mit Hashing-Algorithmus): | 1,1 Sekunden, 1,1 Sekunden, 1,1 Sekunden. |
| PHP Highlighter (mit Präfix-Algorithmus):  | 1,0 sec, 1,0 sec, 1,0 sec.                |

|   |                               |
|---|-------------------------------|
| Resaltador Perl (mit Hash-Algorithmus):     | 1.84 seg, 1.82 seg, 1.83 seg  |
| Resaltador Perl (mit Präferenzalgorithmus): | 1,68 seg, 1,68 seg, 1,68 seg. |

Die Tests zeigen, wie lange es dauert, eine ganze Datei eine bestimmte Anzahl von Malen zu verarbeiten. Der Test mit Perl führte 5000 einfache Überprüfungen einer Datei durch. Die Scan-Routine sah wie folgt aus:

```
procedure BrowseFile ( lines : TStringList ; hlt : TSynCustomHighlighter );
// Erkunden Sie eine Datei mit dem angegebenen Highlighter.
var
  p : PChar ;
  tam : ganze Zahl ;
  Lin : string ;
begin
  for lin in lines do begin
    hlt.SetLine ( lin,1 );
    while not hlt.GetEol do begin
      hlt.Next ;
      hlt.GetTokenEx ( p,tam );
      hlt.GetTokenAttribute ;
    end;
  end;
end;
```

Beide Tests wurden am Beispiel einer PHP- und einer Perl-Quelldatei durchgeführt. Die von Perl hatte 427 Zeilen. Der Vergleich wurde auf einem 32-Bit-Windows-System mit Intel-Architektur durchgeführt.

Die Tests haben ergeben, dass der Präfix-Algorithmus im Allgemeinen schneller ist als der Hash-Funktions-Algorithmus. In bestimmten Fällen mag dies nicht der Fall sein, aber die Implementierung des Präfix-Algorithmus kann immer verbessert werden, um ihn vergleichsweise schneller zu machen.

Unter Berücksichtigung anderer Vergleichskriterien kann die folgende Tabelle erstellt werden:

| KRITERIUM  | HASH-FUNKTIONS-ALGORITHMUS  | ALGORITHMUS DES ERSTEN ZEICHENS ALS PRÄFIX  |
|--|---|---|
| Geschwindigkeit                                    | In den meisten Fällen langsamer   | Schneller in den meisten Fällen   |
| Code Größe   | Größer, weil es eine zusätzliche Funktionstabelle und eine große Anzahl von Funktionen erfordert.                     | Geringfügig, da es nur eine Tabelle mit Funktionen und einfachen Vergleichen verwendet.   |
| Optimierbar  | Schwierig zu optimieren. Es erfordert die Verbesserung der Hash-Funktion, kann aber die Verarbeitungszeit erschweren. | Leichter zu optimieren. Es können weitere Ebenen in Form eines Präfixbaums hinzugefügt oder Vergleiche verbessert werden.           |
| Lesbarkeit   | Nicht sehr lesbar. Es ist schwierig, dem Code zu folgen.  | Sie ist besser lesbar.  |
| Wartung.   | Schwierig zu pflegen. Beim Hinzufügen neuer Schlüsselwörter muss die Hash-Funktion neu berechnet werden.              | Einfach zu pflegen. Wenn Sie neue Schlüsselwörter hinzufügen, müssen Sie sie nur in die entsprechende Funktion einfügen.            |
| Möglichkeit der Verwendung externer Syntaxdateien. | Sehr wenige. Seine inhärente Struktur erschwert seine Dynamisierung.  | Besser handhabbar. Durch die Anordnung der Bezeichner kann es besser an die Verwendung von externen Syntaxdateien angepasst werden. |

Nach dem durchgeführten Vergleich ist es NICHT EMPFOHLEN, Syntax-Highlighter mit dem Hash-Funktions-Algorithmus zu implementieren, der in SynEdit verwendet wurde.

### 3.1.4 Optimierungskriterien

Ich habe wiederholt betont, dass bei der Implementierung von Highlightern schnelle Reaktionsalgorithmen verwendet werden müssen.

Hier möchte ich einige Punkte aufzeigen, die bei der Implementierung von Algorithmen für Highlighter zu berücksichtigen sind, wenn man bedenkt, dass wir den Free Pascal Compiler in seiner Version 2.6.2 verwenden. Einige dieser Kriterien gelten möglicherweise nicht für andere Compiler oder sogar für andere Versionen von Free Pascal.

Der erste Punkt, den wir berücksichtigen müssen, ist, dass Zeichenketten vom Typ PChar im Allgemeinen schneller sind als Zeichenketten vom Typ String, da sie als Zeiger behandelt werden. Eine weitere Überlegung in Bezug auf Zeiger ist, dass, wenn Sie eine Zeichenkette vom Typ PChar haben, die wie folgt deklariert ist:

```
fLine: PChar;
```

Der schnellste Weg, auf ein Zeichen zuzugreifen, ist: `fLine[i]`

Die Form: `(fLine+i)^` ist merklich langsamer (schätzungsweise 7 % langsamer).

In Highlightern ist es üblich, einen Zeiger vorzuschieben, solange das angezeigte Zeichen zu einem gültigen Zeichensatz gehört. Um schnell zu vergleichen, ob ein Zeichen in einem Zeichensatz enthalten ist, werden drei Methoden diskutiert:

#### Array mit booleschen Werten

Ein typischer Algorithmus würde so aussehen:

```
var CharsIdentif : array [#0..#255] of ByteBool ;  
...  
while CharsIdentif[fLine[posEnd]] do  
    inc ( posEnd );
```

Hier wird davon ausgegangen, dass das Array CharsIdentif[] für die Zeichen initialisiert wurde, die Sie als gültig betrachten möchten.

#### Vergleich mit Sets

Ein typischer Algorithmus würde so aussehen:

```
var letters : array of char ;  
...  
while fLine[posFin] in Buchstaben do  
    inc ( posEnd );
```

Dabei wird davon ausgegangen, dass der Buchstabensatz mit den Zeichen begonnen hat, die Sie als gültig betrachten wollen.

#### Vergleich mit Case...Of

Eine schnelle Vergleichsmethode kann auch mit der folgenden Struktur implementiert werden:

```
while true do begin  
    case fLine[posFin] von  
        '$' , '_' , '0'..'9' , 'a'..'z' , 'A'..'Z' : inc ( posEnd ) ;  
    else break ; // exit  
    end;  
end;
```

Eine Vergleichstabelle zeigt die Reaktionsgeschwindigkeiten dieser drei Methoden, die unter Windows mit 32-Bit-x86-Architektur getestet wurden:

| METHODE                     | ZEIT          |
|-----------------------------|---------------|
| Array mit booleschen Werten | 39            |
| Vergleich mit Sets          | 48            |
| Vergleich mit Case...Of     | Vier.<br>Fünf |

Es ist zu erkennen, dass der Vergleich mit einem Array von Booleschen Werten etwa 20% schneller ist als der Vergleich mit Mengen und etwa 13% schneller als der Vergleich mit Case ..Of.

Es wurde auch ein Vergleich mit einem Array von Ganzzahlen anstelle von booleschen Werten versucht, wobei eine längere Verzögerung erzielt wurde. Die Verwendung ganzer Zahlen ist etwa 1 bis 2 % langsamer.

## Inhaltsverzeichnis

|   |    |
|---|----|
| 1 Syntax-bewusster Editor: SynEdit.....                           | 4  |
| 1.1 Was ist SynEdit?.....   | 4  |
| 1.2 SynEdit-Funktionen.....                                       | 5  |
| 1.3 Erscheinungsbild.....   | 5  |
| 1.3.1 Rechter Rand.....   | 7  |
| 1.3.2 Typografie.....   | 8  |
| 1.4 Funktionsweise.....   | 11 |
| 1.4.1 Editor Koordinaten.....                                     | 11 |
| 1.4.2 Handhabung des Cursors.....                                 | 14 |
| 1.4.3 Zeilen Begrenzungszeichen.....                              | 15 |
| 1.5 Ändern Sie den Inhalt.....                                    | 17 |
| 1.5.1 Befehle ausführen.....                                      | 18 |
| 1.5.2 Zugriff auf Lines[].....                                    | 20 |
| 1.5.3 Die Zwischenablage.....                                     | 22 |
| 1.5.4 Wiederherstellen und Rückgängig machen.....                 | 23 |
| 1.6 Verwaltung der Auswahl.....                                   | 26 |
| 1.6.1 Auswahl im Spaltenmodus.....                                | 29 |
| 1.6.2 BlockBegin und BlockEnd.....                                | 31 |
| 1.6.3 BlockBegin und BlockEnd in der normalen Auswahl.....        | 33 |
| 1.6.4 BlockBegin und BlockEnd in der Auswahl im Spaltenmodus..... | 34 |
| 1.7 Suchen und Ersetzen.....                                      | 35 |
| 1.7.1 Suchen.....   | 37 |
| 1.7.2 Suche mit TFindDialog.....                                  | 38 |
| 1.7.3 Ersetzen.....   | 40 |
| 1.7.4 Ersetzen mit TReplaceDialog.....                            | 42 |
| 1.8 Hervorhebungsoptionen.....                                    | 44 |
| 1.8.1 Hervorheben eines Textes.....                               | 44 |
| 1.8.2 Hervorhebung des aktuellen Wortes.....                      | 46 |
| 1.8.3 Hervorhebung der aktuellen Zeile.....                       | 48 |
| 1.8.4 Hervorheben einer beliebigen Zeile.....                     | 49 |
| 1.8.5 Textmarkierungen.....                                       | 50 |
| 1.8.6 Zugriff auf Lesezeichen.....                                | 53 |
| 1.8.7 Mehr über Lesezeichen.....                                  | 55 |
| 1.9 Verwendung von Plugins.....                                   | 60 |
| 1.9.1 Synchrone Bearbeitung.....                                  | 60 |
| 1.9.2 Mehrere Cursor.....   | 61 |
| 1.10 Zusammenfassung der Eigenschaften und Methoden.....          | 62 |
| 1.10.1 Optionen und Optionen2 Eigenschaft.....                    | 65 |
| 2 Syntaxhervorhebung und Autovervollständigung mit SynEdit.....   | 67 |
| 2.1 Einführung.....   | 67 |
| 2.1.1 Wichtige Konzepte.....                                      | 68 |
| 2.2 Syntaxfärbung mit vordefinierten Komponenten.....             | 70 |
| 2.2.1 Verwendung einer vordefinierten Sprache.....                | 70 |
| 2.2.2 Verwendung einer benutzerdefinierten Sprache.....           | 70 |
| 2.3 Syntaxfärbung mit Code.....                                   | 71 |
| 2.3.1 Fälle von Syntaxhervorhebung.....                           | 72 |

|  |     |
|--|-----|
| 2.3.2 Zeilenabtastung.....   | 73  |
| 2.3.3 Erste Schritte.....  | 78  |
| 2.3.4 Hinzufügen von Funktionen zur Syntax.....                            | 82  |
| 2.3.5 GetDefaultAttribute-Eigenschaft.....                                 | 96  |
| 2.3.6 Erkennen von Schlüsselwörtern.....                                   | 97  |
| 2.3.7 Optimierung der Syntax.....  | 98  |
| 2.3.8 Einzeiliger Kommentar Farbgebung.....                                | 103 |
| 2.3.9 Färbung der Kette.....   | 104 |
| 2.3.10 Verwaltung eines Bereiches.....                                     | 106 |
| 2.3.11 Bereich oder Kontextfärbung.....                                    | 110 |
| 2.4 Die Klasse TSynCustomHighlighter.....                                  | 115 |
| 2.4.1 CurrentLines[] und CurrentRanges[].....                              | 115 |
| 2.4.2 Einige Methoden und Eigenschaften.....                               | 117 |
| 2.4.3 Attribute.....   | 117 |
| 2.5 Code-Faltfunktionalität.....   | 122 |
| 2.5.1 Basic Folding.....   | 122 |
| 2.5.2 Beispiel-Code.....   | 125 |
| 2.5.3 Verbesserte Faltung.....   | 133 |
| 2.5.4 Mehr über Folding.....   | 135 |
| 2.5.5 Aktivieren und Deaktivieren von Faltblöcken.....                     | 137 |
| 2.6 Die Klasse TSynCustomFoldHighlighter.....                              | 140 |
| 2.6.1 Low Level Folding.....   | 141 |
| 2.6.2 CurrentLines[] und CurrentRanges[].....                              | 143 |
| 2.6.3 Einige Methoden und Eigenschaften.....                               | 147 |
| 2.7 Autovervollständigen.....  | 148 |
| 2.7.1 Autovervollständigung mit Komponente.....                            | 149 |
| 2.7.2 Autovervollständigen mit Code.....                                   | 150 |
| 3 Anhang.....  | 152 |
| 3.1 Algorithmen für die Implementierung von Highlightern.....              | 152 |
| 3.1.1 Algorithmus, der auf der Verwendung von Hash-Funktionen basiert..... | 152 |
| 3.1.2 Algorithmus auf der Grundlage des ersten Zeichens als Präfix.....    | 154 |
| 3.1.3 Vergleich zwischen Algorithmen.....                                  | 155 |
| 3.1.4 Optimierungskriterien.....   | 156 |