# Приложение А. Исходный код программы

## Файл «ErrorDisplayFormUnit.pas»

unit ErrorDisplayFormUnit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, ComCtrls;

type

TErrorDisplayForm = class(TForm)

CheckRichEdit: TRichEdit;

CloseFormButton: TButton;

procedure CloseFormButtonClick(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure FormResize(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

procedure DisplayOutputMessage(messageType: Byte; errorLine: LongInt; messageText: String);

procedure ClearCheckMessages;

var

ErrorDisplayForm : TErrorDisplayForm;

CheckedFileName : String;

implementation

{$R \*.dfm}

procedure TErrorDisplayForm.CloseFormButtonClick(Sender: TObject);

begin

Hide;

end;

procedure TErrorDisplayForm.FormCreate(Sender: TObject);

begin

CheckRichEdit.Lines.Add('[Проверка]');

CheckRichEdit.SelStart := 0;

CheckRichEdit.SelLength := Length(CheckRichEdit.Text);

CheckRichEdit.SelAttributes.Color := clGreen;

CheckRichEdit.SelAttributes.Style := [fsBold];

CheckRichEdit.SelLength := 0;

CheckRichEdit.SelStart := Length(CheckRichEdit.Text);

end;

procedure TErrorDisplayForm.FormResize(Sender: TObject);

begin

CloseFormButton.Left := (ClientWidth - CloseFormButton.Width) div 2;

CloseFormButton.Top := ClientHeight - 40;

CheckRichEdit.Width := ClientWidth - 10;

CheckRichEdit.Height := ClientHeight - 55;

end;

procedure ClearCheckMessages;

begin

with ErrorDisplayForm do

while (CheckRichEdit.Lines.Count > 1) do

begin

CheckRichEdit.Lines.Delete(1);

end;

end;

procedure DisplayOutputMessage(messageType: Byte; errorLine: LongInt; messageText: String);

var i : LongInt;

offset : LongInt;

len : LongInt;

errorMessageText : String;

hintColor : TColor;

begin

with ErrorDisplayForm do

begin

case messageType of

0:

begin

errorMessageText := '[Ошибка] ';

hintColor := clRed;

end;

1:

begin

errorMessageText := '[Подсказка] ';

hintColor := $FF9400;

end;

end;

CheckRichEdit.Lines.Add(errorMessageText + CheckedFileName + ':' + IntToStr(errorLine + 1) + ' - ' + messageText);

offset := 0;

for i := 0 to CheckRichEdit.Lines.Count - 2 do

Inc(offset, Length(CheckRichEdit.Lines[i]) + 2);

CheckRichEdit.SelStart := offset;

CheckRichEdit.SelLength := Length(errorMessageText) - 1;

CheckRichEdit.SelAttributes.Color := hintColor;

CheckRichEdit.SelAttributes.Style := [fsBold];

len := Length(CheckRichEdit.Lines[CheckRichEdit.Lines.Count - 1]);

len := len - Length(messageText) - Length(errorMessageText) - 3;

CheckRichEdit.SelStart := offset + Length(errorMessageText);

CheckRichEdit.SelLength := len;

CheckRichEdit.SelAttributes.Color := clBlue;

CheckRichEdit.SelAttributes.Style := [];

CheckRichEdit.SelLength := 0;

end;

end;

end.

## Файл « StyleConfiguratorUnit.pas»

unit StyleConfiguratorUnit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, ExtCtrls, UnitMain, ComCtrls;

type

TStyleConfiguratorForm = class(TForm)

GroupBox1: TGroupBox;

LangListBox: TListBox;

Label1: TLabel;

GroupBox2: TGroupBox;

Label2: TLabel;

StyleOptionListBox: TListBox;

GroupBox3: TGroupBox;

Label3: TLabel;

Label4: TLabel;

FontColorPreview: TImage;

BackColorPreview: TImage;

Label5: TLabel;

SaveStyleBtn: TButton;

ApplyStyleBtn: TButton;

CancelBtn: TButton;

GroupBox4: TGroupBox;

Label6: TLabel;

Label7: TLabel;

FontPickerBox: TComboBox;

fsBoldCheckBox: TCheckBox;

fsItalicCheckBox: TCheckBox;

fsUnderlineCheckBox: TCheckBox;

Label8: TLabel;

FontSizeEdit: TEdit;

FontSizeUpDown: TUpDown;

textColorPicker: TColorDialog;

procedure FormCreate(Sender: TObject);

procedure CancelBtnClick(Sender: TObject);

procedure StyleOptionListBoxClick(Sender: TObject);

procedure FormActivate(Sender: TObject);

procedure ApplyStyleBtnClick(Sender: TObject);

procedure SaveStyleBtnClick(Sender: TObject);

procedure fsBoldCheckBoxClick(Sender: TObject);

procedure fsItalicCheckBoxClick(Sender: TObject);

procedure fsUnderlineCheckBoxClick(Sender: TObject);

procedure FontColorPreviewClick(Sender: TObject);

procedure BackColorPreviewClick(Sender: TObject);

procedure FontPickerBoxChange(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

var

StyleConfiguratorForm : TStyleConfiguratorForm;

newCodeStyles : array of TFontStyle;

implementation

{$R \*.dfm}

procedure TStyleConfiguratorForm.FormCreate(Sender: TObject);

begin

SetLength(newCodeStyles, DELPHI\_STYLES\_COUNT);

LangListBox.ItemIndex := 5;

FontColorPreview.Canvas.Pen.Width := 2;

BackColorPreview.Canvas.Pen.Width := 2;

end;

procedure TStyleConfiguratorForm.CancelBtnClick(Sender: TObject);

begin

Close;

end;

procedure TStyleConfiguratorForm.StyleOptionListBoxClick(Sender: TObject);

begin

FontColorPreview.Canvas.Brush.Color := newCodeStyles[StyleOptionListBox.ItemIndex].textColor;

FontColorPreview.Canvas.Rectangle(1, 1, 39, 39);

BackColorPreview.Canvas.Brush.Color := newCodeStyles[StyleOptionListBox.ItemIndex].backColor;

BackColorPreview.Canvas.Rectangle(1, 1, 39, 39);

fsBoldCheckBox.Checked := newCodeStyles[StyleOptionListBox.ItemIndex].bold;

fsItalicCheckBox.Checked := newCodeStyles[StyleOptionListBox.ItemIndex].italic;

fsUnderlineCheckBox.Checked := newCodeStyles[StyleOptionListBox.ItemIndex].underlined;

end;

procedure TStyleConfiguratorForm.FormActivate(Sender: TObject);

var i : LongInt;

begin

for i := 0 to High(newCodeStyles) do

newCodeStyles[i] := fontStyles[i];

StyleOptionListBox.ItemIndex := 0;

StyleOptionListBoxClick(nil);

end;

procedure TStyleConfiguratorForm.ApplyStyleBtnClick(Sender: TObject);

var i : LongInt;

begin

for i := 0 to High(newCodeStyles) do

fontStyles[i] := newCodeStyles[i];

CodeField.Refresh;

end;

procedure TStyleConfiguratorForm.SaveStyleBtnClick(Sender: TObject);

var configFile : TextFile;

i : LongInt;

a, b, c : Byte;

begin

ApplyStyleBtnClick(nil);

CancelBtnClick(nil);

// Save configs to file

AssignFile(configFile, CODE\_VIEWER\_STYLE\_FILE);

Rewrite(configFile);

for i := 0 to High(newCodeStyles) do

begin

WriteLn(configFile, fontStyles[i].fontName);

WriteLn(configFile, fontStyles[i].fontSize);

a := 0;

b := 0;

c := 0;

if fontStyles[i].bold then a := 1;

if fontStyles[i].italic then b := 1;

if fontStyles[i].underlined then c := 1;

WriteLn(configFile, a);

WriteLn(configFile, b);

WriteLn(configFile, c);

WriteLn(configFile, fontStyles[i].textColor);

WriteLn(configFile, fontStyles[i].backColor);

WriteLn(configFile);

end;

CloseFile(configFile);

end;

procedure TStyleConfiguratorForm.fsBoldCheckBoxClick(Sender: TObject);

begin

newCodeStyles[StyleOptionListBox.ItemIndex].bold := fsBoldCheckBox.Checked;

end;

procedure TStyleConfiguratorForm.fsItalicCheckBoxClick(Sender: TObject);

begin

newCodeStyles[StyleOptionListBox.ItemIndex].italic := fsItalicCheckBox.Checked;

end;

procedure TStyleConfiguratorForm.fsUnderlineCheckBoxClick(Sender: TObject);

begin

newCodeStyles[StyleOptionListBox.ItemIndex].underlined := fsUnderlineCheckBox.Checked;

end;

procedure TStyleConfiguratorForm.FontColorPreviewClick(Sender: TObject);

begin

// Pick color

textColorPicker.Color := newCodeStyles[StyleOptionListBox.ItemIndex].textColor;

if textColorPicker.Execute then

newCodeStyles[StyleOptionListBox.ItemIndex].textColor := textColorPicker.Color;

StyleOptionListBoxClick(nil);

end;

procedure TStyleConfiguratorForm.BackColorPreviewClick(Sender: TObject);

begin

// Pick color

textColorPicker.Color := newCodeStyles[StyleOptionListBox.ItemIndex].backColor;

if textColorPicker.Execute then

newCodeStyles[StyleOptionListBox.ItemIndex].backColor := textColorPicker.Color;

StyleOptionListBoxClick(nil);

end;

procedure TStyleConfiguratorForm.FontPickerBoxChange(Sender: TObject);

begin

newCodeStyles[StyleOptionListBox.ItemIndex].fontName := FontPickerBox.Text;

end;

end.

Файл «XmlCodeUnit.pas»

unit XmlCodeUnit;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, StdCtrls, XMLDOM, XMLIntf, MSXMLDOM, XMLDoc, UnitMain;

const

ERR\_MSG\_ELSE = #39 + ';' + #39 + ' недопустимо перед ' + #39 + 'ELSE' + #39;

ERR\_MSG\_BRACES\_COUNT = 'Неверное количество скобок';

ERR\_MSG\_BRACE\_CLOSED\_EARLY = 'Закрывающая скобка раньше открывающей';

ERR\_MSG\_PROC\_PARAMS = 'Ошибка в объявлении параметров процедуры';

ERR\_MSG\_INCORRECT\_PROC\_NAME = 'Некорректное название процедуры';

ERR\_MSG\_INCORRECT\_IDENTIFIER\_NAME = 'Некорректное имя идентификатора';

ERR\_MSG\_INCORRECT\_VAR\_NAME = 'Некорректное название переменной';

ERR\_MSG\_VAR\_DECLARATION\_EXPECTED = 'Ожидалось объявление переменной';

ERR\_MSG\_VAR\_DECLARATION\_WRONG = 'Ошибка в объявлении переменной';

ERR\_MSG\_CONST\_DECLARATION\_EXPECTED = 'Ожидалось объявление константы';

ERR\_MSG\_INCORRECT\_TYPE\_NAME = 'Некорректное имя типа';

ERR\_MSG\_INCORRECT\_RANGE = 'Некорректный диапазон';

ERR\_MSG\_RANGE\_BORDERS = 'Правая граница диапазона меньше левой';

ERR\_MSG\_NO\_USES\_MODULE\_DEFINED = 'Ожидалось объявление модуля';

ERR\_MSG\_INVALID\_TYPE\_DECLARATION = 'Ошибка в объявлении типа';

ERR\_MSG\_FOR\_LOOP\_DECLARATION = 'Ошибка в объявлении цикла FOR';

ERR\_MSG\_INCORRECT\_STATEMENT = 'Некорректное выражение';

ERR\_MSG\_THEN\_EXPECTED = 'Ожидалось ' + #39 + 'THEN' + #39 + ' после условного выражения';

ERR\_MSG\_FUNCTION\_DEFINITION\_EXPECTED = 'Ожидалось определение функции';

ERR\_MSG\_PROC\_DEFINITION\_EXPECTED = 'Ожидалось определение процедуры';

ERR\_MSG\_WHILE\_DO\_EXPECTED = 'Ожидалось ' + #39 + 'DO' + #39 + ' после условного выражения';

ERR\_MSG\_CONDITION\_EXPECTED = 'Ожидалось условие';

type

PSingleParameter = ^TSingleParameter;

TSingleParameter = record

parameterNames : TStringList;

attributes : TStringList;

parameterType : String;

end;

type

TXmlCodeForm = class(TForm)

Label1: TLabel;

XMLDocument: TXMLDocument;

DisplayXmlMemo: TMemo;

ShowCheckStatusBtn: TButton;

procedure FormResize(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure Button1Click(Sender: TObject);

private

{ Private declarations }

public

{ Public declarations }

end;

procedure RefreshXmlDocument;

procedure CreateXmlDocument;

procedure ClearXmlDocument;

procedure ParseXmlToMemo;

function ParseNodeText(node: IXMLNode; tabSize: LongInt) : String;

function SplitBySubString(const src, substr: String) : TStringList;

function IsRangeValid(src: String) : Boolean;

procedure ParseCodeIntoExpressionsList;

procedure AddExpressionLine(const line: LongInt);

procedure CheckExpressionsList;

procedure BuildCodeXmlModel;

function GetLineVariable(const src: String; var varNames : TStringList; var varType: String) : Boolean;

function GetLineType(const src: String; var typeName, typeId: String; var recordNode: IXMLNode) : Boolean;

function MatchProcedureDeclarationTemplate(const src: String; var procName, className: String; var parameters: TList) : Boolean;

function MatchFunctionDeclarationTemplate(const src: String; var funcName, className, retType: String; var parameters: TList) : Boolean;

procedure ProcessFORStatement(var node: IXMLNode);

procedure ProcessWHILEStatement(var node: IXMLNode);

procedure ProcessREPEATStatement(var node: IXMLNode);

procedure ProcessIFStatement(var node: IXMLNode);

procedure ProcessOrdinaryStatement(var node: IXMLNode);

procedure ParseProgramSection;

procedure CheckElseErrors;

procedure CheckGlobalVariablesSection;

procedure CheckUsesSection;

procedure CheckTypeSection;

procedure CheckProcedure;

procedure CheckFunction;

procedure ProcessComplexOperator(var operatorNode: IXMLNode);

procedure ProcessSingleOperator(var operatorNode: IXMLNode);

function CheckParametersList(const src: String; var params: TList) : Boolean;

function GetParameter(const src: String; var success: Boolean) : PSingleParameter;

function IsValidArrayDeclaration(const rangePart, typePart: String; var rangeStr, arrayType: String) : Boolean;

var

XmlCodeFrame : TXmlCodeForm;

XmlMainNode : IXMLNode;

CodeToCheck : String;

ExpressionsList : TStringList;

ExpressionLines : TList;

currentNode : IXmlNode;

childNode : IXmlNode;

checkedLine : LongInt;

implementation

uses ErrorDisplayFormUnit, StrUtils;

{$R \*.dfm}

procedure ProcessOrdinaryStatement(var node: IXMLNode);

var assignPos : LongInt;

lvalue : String;

rvalue : String;

innerNode : IXMLNode;

begin

if (checkedLine < ExpressionsList.Count) then

begin

assignPos := Pos(':=', ExpressionsList[checkedLine]);

if (assignPos > 0) then

begin

innerNode := node.AddChild('Assignment', -1);

lvalue := Trim(Copy(ExpressionsList[checkedLine], 1, assignPos - 1));

rvalue := Trim(Copy(ExpressionsList[checkedLine], assignPos + 2, Length(ExpressionsList[checkedLine])));

if (Length(lvalue) > 0) then

innerNode.Attributes['LValue'] := lvalue;

if (Length(rvalue) > 0) then

innerNode.Attributes['RValue'] := rvalue;

if not (IsVariable(lvalue) and IsCorrectStatement(rvalue)) then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_INCORRECT\_STATEMENT)

end

else

begin

if IsCorrectStatement(ExpressionsList[checkedLine]) then

begin

innerNode := node.AddChild('ProcedureCall', -1);

if (Length(ExpressionsList[checkedLine]) > 0) then

innerNode.Attributes['statement'] := ExpressionsList[checkedLine];

end

else

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_INCORRECT\_STATEMENT);

end;

end;

end;

// Process 'IF' statement

procedure ProcessIFStatement(var node: IXMLNode);

var breakerPos : LongInt;

src : String;

branchNode : IXMLNode;

begin

Inc(checkedLine);

if (checkedLine < ExpressionsList.Count) then

begin

src := ExpressionsList[checkedLine];

breakerPos := Pos(' THEN', UpperCase(src));

if (breakerPos = Length(src) - 4) then

begin

src := Trim(Copy(src, 1, Length(src) - 5));

if not IsCorrectStatement(src) then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_INCORRECT\_STATEMENT)

else

begin

if ((Length(src) >= 2) and (src[1] = '(') and (src[Length(src)] = ')')) then

src := Trim(Copy(src, 2, Length(src) - 2));

if (Length(src) > 0) then

node.Attributes['condition'] := src

else

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_CONDITION\_EXPECTED);

branchNode := node.AddChild('TrueBranch', -1);

Inc(checkedLine);

ProcessSingleOperator(branchNode);

if ((checkedLine < ExpressionsList.Count) and (UpperCase(ExpressionsList[checkedLine]) = 'ELSE')) then

begin

branchNode := node.AddChild('ElseBranch', -1);

Inc(checkedLine);

ProcessSingleOperator(branchNode);

end;

end;

end

else

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_THEN\_EXPECTED);

end;

end;

procedure ProcessREPEATStatement(var node: IXMLNode);

var src : String;

endCond : Boolean;

begin

Inc(checkedLine);

endCond := False;

while ((not endCond) and (checkedLine < ExpressionsList.Count)) do

begin

src := ExpressionsList[checkedLine];

if (UpperCase(src) = 'UNTIL') then

begin

endCond := True;

Inc(checkedLine);

if ((checkedLine < ExpressionsList.Count) and IsCorrectStatement(ExpressionsList[checkedLine])) then

node.Attributes['condition'] := ExpressionsList[checkedLine];

Inc(checkedLine);

end

else

ProcessSingleOperator(node);

end;

end;

// Process 'WHILE' statement

procedure ProcessWHILEStatement(var node: IXMLNode);

var breakerPos : LongInt;

src : String;

begin

Inc(checkedLine);

if (checkedLine < ExpressionsList.Count) then

begin

src := ExpressionsList[checkedLine];

breakerPos := Pos(' DO', UpperCase(src));

if (breakerPos = 0) then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_WHILE\_DO\_EXPECTED)

else

begin

src := Trim(Copy(src, 1, breakerPos - 1));

if IsCorrectStatement(src) then

node.Attributes['condition'] := src;

// Get loop operator

if (checkedLine + 1 < ExpressionsList.Count) then

begin

Inc(checkedLine);

ProcessSingleOperator(node);

end;

end;

end;

end;

// Process 'FOR' statement

procedure ProcessFORStatement(var node: IXMLNode);

var breakerPos : LongInt;

breakerLen : LongInt;

src : String;

initPart : String;

varName : String;

assignment : LongInt;

condPart : String;

begin

Inc(checkedLine);

if (checkedLine < ExpressionsList.Count) then

begin

src := ExpressionsList[checkedLine];

breakerPos := Pos(' downto', src);

if (breakerPos > 0) then

begin

breakerLen := Length(' downto');

if (checkedLine + 1 < ExpressionsList.Count) then

begin

Inc(checkedLine);

src := src + ' ' + ExpressionsList[checkedLine];

end;

end

else

begin

breakerPos := Pos(' to', src);

breakerLen := 0;

if (breakerPos > 0) then

breakerLen := Length(' to');

end;

if (breakerPos = 0) then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_FOR\_LOOP\_DECLARATION)

else

begin

initPart := Trim(Copy(src, 1, breakerPos - 1));

condPart := Trim(Copy(src, breakerPos + breakerLen, Length(src)));

if (Pos('DO', UpperCase(condPart)) = Length(condPart) - 1) then

begin

SetLength(condPart, Length(condPart) - 2);

condPart := Trim(condPart);

// Divide parameterName

assignment := Pos(':=', initPart);

if (assignment = 0) then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_FOR\_LOOP\_DECLARATION)

else

begin

varName := Trim(Copy(initPart, 1, assignment - 1));

initPart := Trim(Copy(initPart, assignment + 2, Length(initPart)));

end;

end;

if (IsVariable(varName) and IsCorrectStatement(initPart) and IsCorrectStatement(condPart)) then

begin

node.Attributes['variable'] := varName;

node.Attributes['startValue'] := initPart;

node.Attributes['endCond'] := condPart;

end;

// Get loop operator

if (checkedLine + 1 < ExpressionsList.Count) then

begin

Inc(checkedLine);

ProcessSingleOperator(node);

end;

end;

end;

end;

procedure ProcessComplexOperator(var operatorNode: IXMLNode);

begin

Inc(checkedLine);

while ((checkedLine < ExpressionsList.Count) and (UpperCase(ExpressionsList[checkedLine]) <> 'END')) do

begin

if (UpperCase(ExpressionsList[checkedLine]) = 'BEGIN') then

begin

ProcessSingleOperator(operatorNode);

end

else if (UpperCase(ExpressionsList[checkedLine]) = 'IF') then

begin

ProcessSingleOperator(operatorNode);

end

else if (UpperCase(ExpressionsList[checkedLine]) = 'WHILE') then

begin

ProcessSingleOperator(operatorNode);

end

else if (UpperCase(ExpressionsList[checkedLine]) = 'REPEAT') then

begin

ProcessSingleOperator(operatorNode);

end

else if (UpperCase(ExpressionsList[checkedLine]) = 'FOR') then

begin

ProcessSingleOperator(operatorNode);

end

else

begin

ProcessSingleOperator(operatorNode);

end;

end;

Inc(checkedLine);

end;

procedure ProcessSingleOperator(var operatorNode: IXMLNode);

var innerOperator : IXMLNode;

src : String;

begin

if ((checkedLine < ExpressionsList.Count)) then

begin

src := UpperCase(ExpressionsList[checkedLine]);

if (src = 'BEGIN') then

begin

innerOperator := operatorNode.AddChild('ComplexOperator');

//Inc(checkedLine);

ProcessComplexOperator(innerOperator);

end

else if (src = 'IF') then

begin

// Process 'IF' statement

innerOperator := operatorNode.AddChild('If');

ProcessIFStatement(innerOperator);

end

else if (src = 'WHILE') then

begin

// Process 'WHILE' statement

innerOperator := operatorNode.AddChild('WhileLoop');

ProcessWHILEStatement(innerOperator);

end

else if (src = 'REPEAT') then

begin

// Process 'REPEAT' statement

innerOperator := operatorNode.AddChild('RepeatLoop');

ProcessREPEATStatement(innerOperator);

end

else if (src = 'FOR') then

begin

innerOperator := operatorNode.AddChild('ForLoop');

ProcessFORStatement(innerOperator);

end

else

begin

//innerOperator := operatorNode.AddChild('Operator');

ProcessOrdinaryStatement(operatorNode);

Inc(checkedLine);

end;

end;

end;

function MatchProcedureDeclarationTemplate(const src: String; var procName, className: String; var parameters: TList) : Boolean;

var braceLeft, dotPos : LongInt;

i : LongInt;

procDesc : String;

procParams : String;

checkParams : Boolean;

begin

braceLeft := Pos('(', src);

if (braceLeft > 0) then

begin

procDesc := Copy(src, 1, braceLeft - 1);

procParams := Copy(src, braceLeft, Length(src));

end

else

procDesc := src;

procDesc := Trim(procDesc);

procParams := Trim(procParams);

dotPos := Pos('.', procDesc);

if (dotPos > 0) then

begin

className := Copy(procDesc, 1, dotPos - 1);

procName := Copy(procDesc, dotPos + 1, Length(procDesc));

Result := IsIdentifier(procName) and IsIdentifier(className);

end

else

begin

procName := procDesc;

Result := IsIdentifier(procName);

end;

if not Result then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_INCORRECT\_PROC\_NAME);

Result := IsIdentifier(procName) and ((Length(className) = 0) or (IsIdentifier(className)));

checkParams := False;

for i := 1 to Length(procParams) do

if not (procParams[i] in [' ','(',')']) then

begin

checkParams := True;

break;

end;

if (

Result

and

((Length(procParams) > 2) and checkParams and (procParams[1] = '(') and (procParams[Length(procParams)] = ')'))

) then

Result := CheckParametersList(Copy(procParams, 2, Length(procParams) - 2), parameters);

end;

function MatchFunctionDeclarationTemplate(const src: String; var funcName, className, retType: String; var parameters: TList) : Boolean;

var braceLeft : LongInt;

braceRight : LongInt;

dotPos : LongInt;

i : LongInt;

semicolon : LongInt;

procDesc : String;

procParams : String;

begin

Result := False;

for i := Length(src) downto 1 do

if (src[i] = ':') then

begin

retType := Trim(retType);

InvertString(retType);

Result := IsCorrectTypeName(retType);

semicolon := i;

procDesc := Trim(Copy(src, 1, semicolon - 1));

break;

end

else

retType := retType + src[i];

if Result then

begin

braceLeft := Pos('(', procDesc);

braceRight := Pos(')', procDesc);

if ((braceRight = Length(procDesc)) and (braceLeft > 0)) then

begin

procParams := Trim(Copy(procDesc, braceLeft + 1, braceRight - braceLeft - 1));

procDesc := Trim(Copy(procDesc, 1, braceLeft - 1));

end

else if not ((braceRight = 0) and (braceLeft = 0)) then

Result := False;

dotPos := Pos('.', procDesc);

if (dotPos > 0) then

begin

className := Copy(procDesc, 1, dotPos - 1);

funcName := Copy(procDesc, dotPos + 1, Length(procDesc));

Result := IsIdentifier(funcName) and IsIdentifier(className);

end

else

begin

funcName := procDesc;

Result := IsIdentifier(funcName);

end;

if not Result then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_INCORRECT\_PROC\_NAME);

if (Length(procParams) > 0) then

Result := CheckParametersList(procParams, parameters);

end;

end;

procedure CheckTypeSection;

var typeName : String;

typeId : String;

typesRead : Boolean;

recordNode : IXMLNode;

begin

childNode := currentNode.AddChild('TypeSection', -1);

typesRead := False;

while (

(checkedLine + 1 < ExpressionsList.Count)

and

(GetLineType(ExpressionsList[checkedLine + 1], typeName, typeId, recordNode))

) do

begin

childNode := childNode.AddChild('Type', -1);

childNode.Attributes['name'] := typeName;

childNode.Attributes['id'] := typeId;

if (typeId = 'record') then

begin

recordNode.ParentNode.ChildNodes.Remove(recordNode);

childNode.ChildNodes.Add(recordNode);

end;

childNode := childNode.ParentNode;

typesRead := True;

Inc(checkedLine);

end;

Inc(checkedLine);

if not typesRead then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_VAR\_DECLARATION\_EXPECTED);

end;

procedure CheckProcedure;

var procName, className : String;

parameters : TList;

i, j : LongInt;

paramListElement : PSingleParameter;

varNames : TStringList;

varType : String;

operatorNode : IXMLNode;

expectDefinition : Boolean;

begin

Inc(checkedLine);

if (checkedLine < ExpressionsList.Count) then

if MatchProcedureDeclarationTemplate(ExpressionsList[checkedLine], procName, className, parameters) then

begin

childNode := currentNode.AddChild('Procedure', -1);

if (Length(className) > 0) then

childNode.Attributes['namespace'] := className;

childNode.Attributes['name'] := procName;

// Add parameters info

if Assigned(parameters) then

begin

childNode := childNode.AddChild('ParametersList', -1);

for i := 0 to parameters.Count - 1 do

begin

paramListElement := parameters[i];

for j := 0 to paramListElement^.parameterNames.Count - 1 do

begin

childNode := childNode.AddChild('Parameter', -1);

childNode.Attributes['name'] := paramListElement^.parameterNames[j];

childNode.Attributes['type'] := paramListElement^.parameterType;

if (Length(paramListElement^.attributes[j]) > 0) then

childNode.Attributes['modifier'] := paramListElement^.attributes[j];

childNode := childNode.ParentNode;

end;

end;

childNode := childNode.ParentNode;

end;

expectDefinition := False;

// Check variables

if (checkedLine + 1 < ExpressionsList.Count) and (UpperCase(ExpressionsList[checkedLine + 1]) = 'VAR') then

begin

expectDefinition := True;

Inc(checkedLine);

childNode := childNode.AddChild('VarSection', -1);

while (

(checkedLine + 1 < ExpressionsList.Count)

and

(GetLineVariable(ExpressionsList[checkedLine + 1], varNames, varType))

) do

begin

for i := 0 to varNames.Count - 1 do

begin

childNode := childNode.AddChild('Variable', -1);

childNode.Attributes['name'] := varNames[i];

childNode.Attributes['type'] := varType;

childNode := childNode.ParentNode;

end;

Inc(checkedLine);

end;

childNode := childNode.ParentNode;

end;

// Check operators list

if (checkedLine + 1 < ExpressionsList.Count) and (UpperCase(ExpressionsList[checkedLine + 1]) = 'BEGIN') then

begin

Inc(checkedLine);

operatorNode := childNode.AddChild('ProcedureBody');

ProcessSingleOperator(operatorNode);

end

else if expectDefinition then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_PROC\_DEFINITION\_EXPECTED);

end;

end;

procedure CheckFunction;

var funcName, className : String;

retType : String;

parameters : TList;

i, j : LongInt;

paramListElement : PSingleParameter;

varNames : TStringList;

varType : String;

operatorNode : IXMLNode;

expectDefinition : Boolean;

begin

Inc(checkedLine);

if (checkedLine < ExpressionsList.Count) then

if MatchFunctionDeclarationTemplate(ExpressionsList[checkedLine], funcName, className, retType, parameters) then

begin

childNode := currentNode.AddChild('Function', -1);

if (Length(className) > 0) then

childNode.Attributes['namespace'] := className;

childNode.Attributes['name'] := funcName;

childNode.Attributes['returnType'] := retType;

// Add parameters info

if Assigned(parameters) then

begin

childNode := childNode.AddChild('ParametersList', -1);

for i := 0 to parameters.Count - 1 do

begin

paramListElement := parameters[i];

for j := 0 to paramListElement^.parameterNames.Count - 1 do

begin

childNode := childNode.AddChild('Parameter', -1);

childNode.Attributes['name'] := paramListElement^.parameterNames[j];

childNode.Attributes['type'] := paramListElement^.parameterType;

if (Length(paramListElement^.attributes[j]) > 0) then

childNode.Attributes['modifier'] := paramListElement^.attributes[j];

childNode := childNode.ParentNode;

end;

end;

childNode := childNode.ParentNode;

end;

expectDefinition := False;

// Check variables

if (checkedLine + 1 < ExpressionsList.Count) and (UpperCase(ExpressionsList[checkedLine + 1]) = 'VAR') then

begin

expectDefinition := True;

Inc(checkedLine);

childNode := childNode.AddChild('VarSection', -1);

while (

(checkedLine + 1 < ExpressionsList.Count)

and

(GetLineVariable(ExpressionsList[checkedLine + 1], varNames, varType))

) do

begin

for i := 0 to varNames.Count - 1 do

begin

childNode := childNode.AddChild('Variable', -1);

childNode.Attributes['name'] := varNames[i];

childNode.Attributes['type'] := varType;

childNode := childNode.ParentNode;

end;

Inc(checkedLine);

end;

childNode := childNode.ParentNode;

end;

// Check operators list

if (checkedLine + 1 < ExpressionsList.Count) and (UpperCase(ExpressionsList[checkedLine + 1]) = 'BEGIN') then

begin

Inc(checkedLine);

operatorNode := childNode.AddChild('FunctionBody');

ProcessSingleOperator(operatorNode);

end

else if expectDefinition then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_FUNCTION\_DEFINITION\_EXPECTED);

end;

end;

procedure CheckGlobalVariablesSection;

var varNames : TStringList;

varType : String;

i : LongInt;

varsRead : Boolean;

begin

childNode := currentNode.AddChild('VarSection', -1);

varsRead := False;

while (

(checkedLine + 1 < ExpressionsList.Count)

and

(GetLineVariable(ExpressionsList[checkedLine + 1], varNames, varType))

) do

begin

for i := 0 to varNames.Count - 1 do

begin

childNode := childNode.AddChild('Variable', -1);

childNode.Attributes['name'] := varNames[i];

childNode.Attributes['type'] := varType;

childNode := childNode.ParentNode;

end;

varsRead := True;

Inc(checkedLine);

end;

if not varsRead then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_VAR\_DECLARATION\_EXPECTED);

end;

function GetLineType(const src: String; var typeName, typeId: String; var recordNode: IXMLNode) : Boolean;

var dividerPos : LongInt;

i : LongInt;

isRecord : Boolean;

varsRead : Boolean;

fieldNode : IXMLNode;

varNames : TStringList;

varType : String;

begin

dividerPos := Pos('=', src);

Result := (dividerPos > 0);

isRecord := False;

if Result then

begin

typeName := Trim(Copy(src, 1, dividerPos - 1));

typeId := Trim(Copy(src, dividerPos + 1, Length(src)));

isRecord := typeId = 'record';

Result := IsIdentifier(typeName) and (IsCorrectTypeName(typeId) or isRecord);

end;

if (not Result) and ((not IsKeyword(1, src)) or (UpperCase(src) = 'END')) then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine + 1]^), ERR\_MSG\_INVALID\_TYPE\_DECLARATION)

else if isRecord then

begin

// Process as record

varsRead := False;

recordNode := childNode.AddChild('Record', -1);

Inc(checkedLine);

while (

(checkedLine + 1 < ExpressionsList.Count)

and

(

GetLineVariable(ExpressionsList[checkedLine + 1], varNames, varType)

or

(UpperCase(ExpressionsList[checkedLine + 1]) = 'END')

)

) do

begin

Inc(checkedLine);

if not (UpperCase(ExpressionsList[checkedLine]) = 'END') then

begin

for i := 0 to varNames.Count - 1 do

begin

fieldNode := recordNode.AddChild('Field', -1);

fieldNode.Attributes['name'] := varNames[i];

fieldNode.Attributes['type'] := varType;

end;

varsRead := True;

end

else

begin

Dec(checkedLine);

break;

end;

end;

if not varsRead then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_VAR\_DECLARATION\_EXPECTED);

end;

end;

function GetLineVariable(const src: String; var varNames : TStringList; var varType: String) : Boolean;

var dividerPos : LongInt;

i : LongInt;

namePart : String;

typePart : String;

rangeStr : String;

arrayType : String;

begin

dividerPos := Pos(':', src);

Result := (dividerPos > 0);

if Result then

begin

namePart := Trim(Copy(src, 1, dividerPos - 1));

typePart := Trim(Copy(src, dividerPos + 1, Length(src)));

end;

if Result then

begin

varNames := SplitBySubString(namePart, ',');

for i := 0 to varNames.Count - 1 do

if not IsIdentifier(varNames[i]) then

begin

Result := False;

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine + 1]^), ERR\_MSG\_INCORRECT\_VAR\_NAME);

break;

end;

end;

if Result then

begin

if IsCorrectTypeName(typePart) then

varType := typePart

else if ((UpperCase(typePart) = 'ARRAY') and (checkedLine + 3 < ExpressionsList.Count)) then

begin

Result := IsValidArrayDeclaration(

ExpressionsList[checkedLine + 2], ExpressionsList[checkedLine + 3],

rangeStr, arrayType

);

if Result then

begin

varType := typePart;

Inc(checkedLine, 2);

end;

end

else

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine + 1]^), ERR\_MSG\_INCORRECT\_TYPE\_NAME);

end;

end;

function MatchConstTemplate(src: String; var constName, constType, constValue: String) : Boolean;

var divPos, typePos : LongInt;

begin

divPos := Pos('=', src);

typePos := Pos(':', src);

Result := (divPos > 0);

if Result then

begin

if (typePos > 0) then

begin

constType := Copy(src, typePos + 1, divPos - typePos - 1);

constType := Trim(constType);

constName := Copy(src, 1, typePos - 1);

constName := Trim(constName);

Result := IsIdentifier(constType) and IsIdentifier(constName);

end

else

begin

constName := Copy(src, 1, divPos - 1);

constName := Trim(constName);

end;

constValue := Copy(src, divPos + 1, Length(src));

constValue := Trim(constValue);

end;

end;

procedure ParseCodeIntoExpressionsList;

var i, j : LongInt;

currentExpression : String;

braceState : LongInt;

braceError : Boolean;

begin

ExpressionsList.Clear;

ExpressionLines.Clear;

currentExpression := EMPTY\_STRING;

braceState := 0;

braceError := False;

// Parse code into expressions

for i := 0 to CodeField.Lines.Count - 1 do

begin

j := 1;

while (j <= Length(CodeField.Lines[i])) do

begin

if (Ord(CodeField.Lines[i][j]) >= KEY\_SPACE) then

begin

if ((CodeField.Lines[i][j] = ';') and (CodeField.LineStyles[i][j] = TextStyle(TEXT\_STYLE\_NONE))) then

begin

if (Length(Trim(currentExpression)) > 0) then

if (braceState = 0) then

begin

ExpressionsList.Add(Trim(currentExpression));

AddExpressionLine(i);

currentExpression := EMPTY\_STRING;

end

else

currentExpression := currentExpression + ';';

end

else if (

(CodeField.LineStyles[i][j] <> TextStyle(TEXT\_STYLE\_COMMENT))

and

(CodeField.LineStyles[i][j] <> TextStyle(TEXT\_STYLE\_MULTILINE\_COMMENT))

) then

begin

if (

(

(CodeField.LineStyles[i][j] = TextStyle(TEXT\_STYLE\_KEYWORD))

and

(

((j < Length(CodeField.LineStyles[i])) and (CodeField.LineStyles[i][j + 1] <> TextStyle(TEXT\_STYLE\_KEYWORD)))

or

(j = Length(CodeField.LineStyles[i]))

)

)

) then

begin

if (

(Pos('t', currentExpression) <> Length(currentExpression)) and

(Pos('xo', currentExpression) <> Length(currentExpression) - 1) and

(Pos('sh', currentExpression) <> Length(currentExpression) - 1) and

(Pos('mo', currentExpression) <> Length(currentExpression) - 1) and

(Pos('di', currentExpression) <> Length(currentExpression) - 1) and

(Pos('no', currentExpression) <> Length(currentExpression) - 1)

) then

begin

if (Length(Trim(currentExpression + CodeField.Lines[i][j])) > 0) then

begin

if (braceState = 0) then

begin

ExpressionsList.Add(Trim(currentExpression + CodeField.Lines[i][j]));

AddExpressionLine(i);

currentExpression := EMPTY\_STRING;

end

else

currentExpression := currentExpression + CodeField.Lines[i][j];

end;

end

else

currentExpression := currentExpression + CodeField.Lines[i][j];

end

else

begin

if not (Ord(CodeField.LineStyles[i][j]) in [TEXT\_STYLE\_COMMENT, TEXT\_STYLE\_MULTILINE\_COMMENT, TEXT\_STYLE\_STRING]) then

begin

if (CodeField.Lines[i][j] = '(') then

Inc(braceState)

else if (CodeField.Lines[i][j] = ')') then

Dec(braceState);

if (braceState < 0) then

braceError := True;

end;

currentExpression := currentExpression + CodeField.Lines[i][j];

end;

end;

end;

Inc(j);

end;

end;

ExpressionsList.Add(currentExpression);

AddExpressionLine(CodeField.Lines.Count);

i := 0;

while (i < ExpressionsList.Count) do

begin

if (Pos('ELSE', UpperCase(ExpressionsList[i])) = Length(ExpressionsList[i]) - 3) then

begin

currentExpression := Trim(Copy(ExpressionsList[i], 1, Length(ExpressionsList[i]) - 4));

if (Length(currentExpression) > 0) then

begin

ExpressionsList[i] := currentExpression;

ExpressionsList.Insert(i + 1, 'else');

ExpressionLines.Insert(i + 1, ExpressionLines[i]);

Inc(i);

end;

end;

Inc(i);

end;

if (braceError or (braceState <> 0)) then

DisplayOutputMessage(0, CodeField.Lines.Count, ERR\_MSG\_BRACES\_COUNT);

end;

procedure CheckExpressionsList;

var

variablesDefined : LongInt;

varName, varType : String;

varValue : String;

begin

checkedLine := 0;

currentNode := XmlMainNode;

while (checkedLine < ExpressionsList.Count) do

begin

if (Length(ExpressionsList[checkedLine]) > 0) then

begin

if (IsKeyword(1, ExpressionsList[checkedLine])) then

begin

if (UpperCase(ExpressionsList[checkedLine]) = 'PROGRAM') then

ParseProgramSection

else if (UpperCase(ExpressionsList[checkedLine]) = 'USES') then

CheckUsesSection

else if (UpperCase(ExpressionsList[checkedLine]) = 'TYPE') then

CheckTypeSection

else if (UpperCase(ExpressionsList[checkedLine]) = 'VAR') then

CheckGlobalVariablesSection

else if (UpperCase(ExpressionsList[checkedLine]) = 'CONST') then

begin

variablesDefined := 0;

currentNode := currentNode.AddChild('ConstSection', -1);

varType := EMPTY\_STRING;

Inc(checkedLine);

while (

(checkedLine < ExpressionsList.Count)

and

(MatchConstTemplate(ExpressionsList[checkedLine], varName, varType, varValue))

) do

begin

Inc(variablesDefined);

childNode := currentNode.AddChild('Const', -1);

childNode.Attributes['name'] := varName;

if (Length(varType) > 0) then

childNode.Attributes['type'] := varType;

childNode.Attributes['value'] := varValue;

varType := EMPTY\_STRING;

Inc(checkedLine);

end;

// Exit from variable section

currentNode := currentNode.ParentNode;

if (variablesDefined = 0) then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_CONST\_DECLARATION\_EXPECTED);

end

else if (UpperCase(ExpressionsList[checkedLine]) = 'PROCEDURE') then

CheckProcedure

else if (UpperCase(ExpressionsList[checkedLine]) = 'FUNCTION') then

CheckFunction

else

begin

childNode := currentNode.AddChild('UnhandledKeyword', -1);

childNode.Attributes['id'] := ExpressionsList[checkedLine];

Inc(checkedLine);

end;

end

else

begin

childNode := currentNode.AddChild('UnhandledExpression', -1);

childNode.Attributes['text'] := ExpressionsList[checkedLine];

Inc(checkedLine);

end;

end

else

break;

end;

end;

procedure BuildCodeXmlModel;

begin

CheckElseErrors;

ParseCodeIntoExpressionsList;

CheckExpressionsList;

end;

procedure ParseProgramSection;

var programNode : IXMLNOde;

begin

if (checkedLine + 1 < ExpressionsList.Count) then

begin

Inc(checkedLine);

if not IsIdentifier(ExpressionsList[checkedLine]) then

DisplayOutputMessage(

0, LongInt(ExpressionLines[checkedLine]^),

ExpressionsList[checkedLine] + ERR\_MSG\_INCORRECT\_IDENTIFIER\_NAME

)

else

begin

// Add program description to DOM

programNode := XmlMainNode.AddChild('ProgramSection', -1);

programNode.Attributes['name'] := ExpressionsList[checkedLine];

Inc(checkedLine);

end;

end

else

DisplayOutputMessage(

0, 0,

'Ожидалось описание секции ' + #39 + 'PROGRAM' + #39 + ', конец файла достигнут'

);

end;

procedure TXmlCodeForm.FormResize(Sender: TObject);

begin

DisplayXmlMemo.Left := 5;

DisplayXmlMemo.Top := 35;

DisplayXmlMemo.Width := ClientWidth - 10;

DisplayXmlMemo.Height := ClientHeight - 80;

ShowCheckStatusBtn.Top := ClientHeight - ShowCheckStatusBtn.Height - 5;

end;

procedure TXmlCodeForm.FormCreate(Sender: TObject);

begin

ExpressionsList := TStringList.Create;

ExpressionLines := TList.Create;

CreateXmlDocument;

end;

procedure TXmlCodeForm.Button1Click(Sender: TObject);

begin

ErrorDisplayForm.Show;

end;

function IsRangeValid(src: String) : Boolean;

var rangeParts : TStringList;

left : LongInt;

right : LongInt;

begin

Result := (Length(src) > 2) and (src[1] = '[') and (src[Length(src)] = ']');

if Result then

begin

src := Copy(src, 2, Length(src) - 2);

rangeParts := SplitBySubString(src, '..');

if (rangeParts.Count = 2) then

begin

Result := IsNumber(rangeParts[0], left) and IsNumber(rangeParts[1], right);

if Result then

begin

Result := left <= right;

if not Result then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_RANGE\_BORDERS);

end;

end;

end;

if not Result then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), 'Неверно задан диапазон');

end;

procedure CheckElseErrors;

var i, j, k, offset, pos : LongInt;

found : Boolean;

begin

for i := 0 to CodeField.Lines.Count - 1 do

begin

offset := 1;

repeat

pos := PosEx('ELSE', UpperCase(CodeField.Lines[i]), offset);

if (pos > 0) then

begin

if (CodeField.LineStyles[i][pos] = TextStyle(TEXT\_STYLE\_KEYWORD)) then

begin

found := False;

k := i;

Dec(pos);

while ((not found) and (k >= 0)) do

begin

for j := pos downto 1 do

if (Ord(CodeField.Lines[k][j]) > KEY\_SPACE) then

begin

if (CodeField.Lines[k][j] = ';') then

DisplayOutputMessage(0, k, ERR\_MSG\_ELSE);

found := True;

break;

end;

Dec(k);

if (k >= 0) then

pos := Length(CodeField.Lines[k]);

end;

end;

Inc(offset, 4);

end;

until (pos = 0);

end;

end;

procedure AddExpressionLine(const line: LongInt);

var expressionLine : ^LongInt;

begin

New(expressionLine);

expressionLine^ := line;

ExpressionLines.Add(expressionLine);

end;

procedure ParseXmlToMemo;

//var i : LongInt;

begin

with XmlCodeFrame do

begin

XMLDocument.Active := True;

if Assigned(XmlMainNode) then

DisplayXmlMemo.Lines.Add(ParseNodeText(XmlMainNode, 0));

XMLDocument.Active := False;

end;

end;

{FUNCTIONS, THAT ARE OBVIOUSLY CORRECT}

function GetParameter(const src: String; var success: Boolean) : PSingleParameter;

var paramParts : String;

breaker : LongInt;

i : LongInt;

paramsList : TStringList;

keyword : String;

begin

New(Result);

Result^.parameterNames := TStringList.Create;

Result^.attributes := TStringList.Create;

breaker := Pos(':', src);

if (breaker > 0) then

begin

Result^.parameterType := Trim(Copy(src, breaker + 1, Length(src)));

paramParts := Trim(Copy(src, 1, breaker - 1));

success := IsCorrectTypeName(Result^.parameterType);

end

else

success := False;

if success then

begin

paramsList := SplitBySubString(paramParts, ',');

for i := 0 to paramsList.Count - 1 do

begin

if not IsIdentifier(paramsList[i]) then

begin

// Check keyword prefix

keyword := EMPTY\_STRING;

if (Pos('CONST ', UpperCase(paramsList[i])) = 1) then

keyword := 'const'

else if (Pos('VAR ', UpperCase(paramsList[i])) = 1) then

keyword := 'var';

if (Length(keyword) > 0) then

begin

paramsList[i] := Trim(Copy(paramsList[i], Length(keyword) + 1, Length(paramsList[i])));

success := IsIdentifier(paramsList[i]);

end

else

begin

success := False;

break;

end;

end;

if success then

begin

Result^.parameterNames.Add(paramsList[i]);

if (Length(keyword) > 0) then

Result^.attributes.Add(keyword)

else

Result^.attributes.Add(EMPTY\_STRING);

end;

end;

end;

end;

procedure CheckUsesSection;

var modules : TStringList;

i : LongInt;

begin

childNode := currentNode.AddChild('UsesSection', -1);

if (checkedLine + 1 < ExpressionsList.Count) then

begin

Inc(checkedLine);

modules := SplitBySubString(ExpressionsList[checkedLine], ',');

if (modules.Count > 0) then

begin

for i := 0 to modules.Count - 1 do

if (IsModuleName(modules[i])) then

begin

childNode := childNode.AddChild('Module', -1);

childNode.Attributes['name'] := modules[i];

childNode := childNode.ParentNode;

end

else

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_INCORRECT\_IDENTIFIER\_NAME);

Inc(checkedLine);

end

else

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_NO\_USES\_MODULE\_DEFINED);

end;

end;

function CheckParametersList(const src: String; var params: TList) : Boolean;

var temp : String;

i : LongInt;

parameter : PSingleParameter;

success : Boolean;

paramsList : TStringList;

begin

Result := True;

params := TList.Create;

temp := EMPTY\_STRING;

// Parse by ';'

paramsList := SplitBySubString(src, ';');

for i := 0 to paramsList.Count - 1 do

begin

parameter := GetParameter(paramsList[i], success);

if not success then

begin

Result := False;

break;

end

else

params.Add(parameter);

end;

if not Result then

DisplayOutputMessage(0, LongInt(ExpressionLines[checkedLine]^), ERR\_MSG\_PROC\_PARAMS);

end;

function ParseNodeText(node: IXMLNode; tabSize: LongInt) : String;

var hasChildren: Boolean;

childrenString, attributes, tabs: String;

nodeList : IXMLNodeList;

i: LongInt;

tempText : String;

begin

Result := '';

if Assigned(node) then

begin

nodeList := node.ChildNodes;

hasChildren := nodeList.Count > 0;

tabs := '';

for i := 1 to tabSize do

tabs := tabs + #32#32#32;

attributes := '';

for i := 0 to node.AttributeNodes.Count - 1 do

attributes := attributes + #32 +

node.AttributeNodes[i].NodeName + '="' +

node.AttributeNodes[i].NodeValue + '"';

if not hasChildren then

Result := tabs + '<' + node.NodeName + attributes + '/>'

else

begin

childrenString := '';

nodeList := node.ChildNodes;

for i := 0 to node.ChildNodes.Count - 1 do

childrenString := childrenString + ParseNodeText(node.ChildNodes[i], tabSize + 1) + #13#10;

Result := tabs + '<' + node.NodeName + attributes + '>' + #13#10 + childrenString + tabs + '</' + node.NodeName + '>';

end;

end;

end;

procedure RefreshXmlDocument;

begin

XmlCodeFrame.DisplayXmlMemo.Clear;

ClearXmlDocument;

BuildCodeXmlModel;

ParseXmlToMemo;

end;

procedure ClearXmlDocument;

var i : LongInt;

begin

if Assigned(XmlMainNode) then

for i := 0 to XmlMainNode.ChildNodes.Count - 1 do

XmlMainNode.ChildNodes.Remove(XmlMainNode.ChildNodes[0]);

end;

procedure CreateXmlDocument;

begin

with XmlCodeFrame do

begin

XMLDocument.Active := True;

XmlMainNode := XMLDocument.CreateNode('DelphiDocument', ntElement, '');

XMLDocument.DocumentElement := XmlMainNode;

XMLDocument.Active := False;

end;

end;

function IsValidArrayDeclaration(const rangePart, typePart: String; var rangeStr, arrayType: String) : Boolean;

var ofPos : LongInt;

begin

arrayType := typePart;

Result := IsCorrectTypeName(arrayType);

if Result then

begin

ofPos := Pos('of', rangePart);

if (ofPos > 0) and (ofPos = (Length(rangePart) - 1)) then

begin

rangeStr := Trim(Copy(rangePart, 1, ofPos - 1));

Result := IsRangeValid(rangeStr);

end

else

Result := False;

end;

end;

function SplitBySubString(const src, substr: String) : TStringList;

var breakerPos : LongInt;

searchOffset : LongInt;

begin

Result := TStringList.Create;

searchOffset := 1;

repeat

breakerPos := PosEx(substr, src, searchOffset);

if (breakerPos > 0) then

Result.Add(Trim(Copy(src, searchOffset, breakerPos - searchOffset)))

else

Result.Add(Trim(Copy(src, searchOffset, Length(src))));

searchOffset := breakerPos + Length(substr);

until (breakerPos = 0);

end;

end.

## Файл «UnitMain.pas»

unit UnitMain;

interface

uses

Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,

Dialogs, Menus, StdCtrls, Tabs, ComCtrls, ExtCtrls, RichEdit, ClipBrd,

Buttons, StrUtils;

{$Include Unit\TextFieldClass}

{$Include TCodeTextField}

type

TProgramForm = class(TForm)

MainMenu: TMainMenu;

FileMenu: TMenuItem;

HelpBtn: TMenuItem;

NewMenuBtn: TMenuItem;

Open: TMenuItem;

Save: TMenuItem;

SaveAs: TMenuItem;

CloseBtn: TMenuItem;

AboutMenuBtn: TMenuItem;

EditMenu: TMenuItem;

CopyBtn: TMenuItem;

CutBtn: TMenuItem;

Paste: TMenuItem;

DeleteBtn: TMenuItem;

SelectAll: TMenuItem;

Settings: TMenuItem;

Style: TMenuItem;

ExitBtn: TMenuItem;

FilenamesTab: TTabSet;

FontDialog1: TFontDialog;

ColorDialog: TColorDialog;

BlinkTimer: TTimer;

SaveFileDialog: TSaveDialog;

ToolsPanel: TPanel;

OpenFileBitBtn: TSpeedButton;

NewFileBitBtn: TSpeedButton;

CloseFileBitBtn: TSpeedButton;

PrintBitBtn: TSpeedButton;

CutBitBtn: TSpeedButton;

CopyBitBtn: TSpeedButton;

PasteBitBtn: TSpeedButton;

ImageSeparator1: TImage;

ImageSeparator2: TImage;

UndoBitBtn: TSpeedButton;

RedoBitBtn: TSpeedButton;

Separator1: TMenuItem;

Separator2: TMenuItem;

Print: TMenuItem;

Separator3: TMenuItem;

Undo: TMenuItem;

Redo: TMenuItem;

Separator4: TMenuItem;

SaveBitBtn: TSpeedButton;

ImageSeparator3: TImage;

SpellCheckBitBtn: TSpeedButton;

CodeScrollVertical: TScrollBar;

CodeScrollHorizontal: TScrollBar;

procedure FormResize(Sender: TObject);

procedure StyleClick(Sender: TObject);

procedure FormCreate(Sender: TObject);

procedure FilenamesTabChange(Sender: TObject; NewTab: Integer;

var AllowChange: Boolean);

procedure OpenClick(Sender: TObject);

procedure FormDestroy(Sender: TObject);

procedure CloseFileClick(Sender: TObject);

procedure ExitClick(Sender: TObject);

procedure SaveClick(Sender: TObject);

procedure FormKeyPress(Sender: TObject; var Key: Char);

procedure FormKeyDown(Sender: TObject; var Key: Word;

Shift: TShiftState);

procedure BlinkTimerTimer(Sender: TObject);

procedure SelectAllClick(Sender: TObject);

procedure DeleteClick(Sender: TObject);

procedure CopyClick(Sender: TObject);

procedure CutClick(Sender: TObject);

procedure PasteClick(Sender: TObject);

procedure NewMenuBtnClick(Sender: TObject);

procedure SaveAsClick(Sender: TObject);

procedure OpenFileBitBtnClick(Sender: TObject);

procedure NewFileBitBtnClick(Sender: TObject);

procedure CloseFileBitBtnClick(Sender: TObject);

procedure AboutMenuBtnClick(Sender: TObject);

procedure CutBitBtnClick(Sender: TObject);

procedure CopyBitBtnClick(Sender: TObject);

procedure PasteBitBtnClick(Sender: TObject);

procedure SpellCheckBitBtnClick(Sender: TObject);

procedure UndoClick(Sender: TObject);

procedure RedoClick(Sender: TObject);

procedure SaveBitBtnClick(Sender: TObject);

procedure CodeScrollVerticalChange(Sender: TObject);

procedure FormMouseWheelDown(Sender: TObject; Shift: TShiftState;

MousePos: TPoint; var Handled: Boolean);

procedure FormMouseWheelUp(Sender: TObject; Shift: TShiftState;

MousePos: TPoint; var Handled: Boolean);

procedure FormClose(Sender: TObject; var Action: TCloseAction);

private

public

end;

var

ProgramForm : TProgramForm;

TextLinesCounter : TLinesCounter;

FilesManager : TNexonFileManager;

OpenFileDlg : TOpenDialog;

CodeField : TCodeTextField;

CodeFont : TFont;

TextCaret : TTextCaret;

BlinkTimerValue : LongWord;

BlinkFrequency : Word;

FilenamesTabCanChange : Boolean;

VerticalScrollChange : Boolean;

{$Include TextFieldProcedures}

implementation

uses XmlCodeUnit, StyleConfiguratorUnit, ErrorDisplayFormUnit;

{$R \*.dfm}

{$Include TCodeTextField\_implement}

{$Include Unit\TextFieldClassImplement}

{$Include TextFieldProceduresSrc}

procedure TProgramForm.FormResize(Sender: TObject);

begin

// Resize by screen

FitTextFieldIntoWindow;

end;

procedure TProgramForm.StyleClick(Sender: TObject);

begin

// Open style window

StyleConfiguratorForm.ShowModal;

end;

procedure TProgramForm.FormCreate(Sender: TObject);

begin

DoubleBuffered := True;

BlinkTimerValue := 0;

BlinkFrequency := DEFAULT\_BLINK\_RATE;

LoadKeywords;

CodeFont := TFont.Create;

CodeFont.Name := DEFAULT\_FONT\_NAME;

CodeFont.Size := DEFAULT\_FONT\_SIZE;

CodeField := TCodeTextField.Create(Self);

CodeField.Parent := Self;

CodeField.Visible := True;

CodeField.Width := FIELD\_WIDTH\_MAX;

CodeField.Height := FIELD\_HEIGHT\_MAX;

CodeField.Font := CodeFont;

TextCaret := TTextCaret.Create(Self);

TextCaret.Parent := Self;

TextCaret.Visible := True;

CodeField.CaretObj := TextCaret;

InitCodeStyles;

InitFileManager;

InitOpenFileDialog;

InitLinesCounter;

FilenamesTabCanChange := True;

VerticalScrollChange := True;

//IsArrayElementCall('sukabitch [a][b][c, d]');

//IsCorrectStatement('--Trim(Copy(src+2, 72, 7) --2) \* 67 / 2 and 4 mod 9 div 10');

end;

procedure TProgramForm.FilenamesTabChange(Sender: TObject; NewTab: Integer; var AllowChange: Boolean);

begin

// On tab change handler

if (FilenamesTabCanChange) then

begin

// Save current view to buffer

FilesManager.UpdateActiveFileContents;

FilesManager.ActiveView := NewTab;

UpdateActiveTextField;

end;

end;

procedure TProgramForm.OpenClick(Sender: TObject);

var path : String;

begin

FilenamesTabCanChange := False;

// On File Open Dialog

if OpenFileDlg.Execute then

begin

path := OpenFileDlg.FileName;

if FileExists(path) then

begin

FilesManager.UpdateActiveFileContents;

FilesManager.AddFile(ExtractFileName(path), LoadFileToStr(path));

FilesManager.ActiveView := FilesManager.GetFilesCount - 1;

UpdateActiveTextField;

UpdateFilenamesTab;

end;

end;

FilenamesTabCanChange := True;

end;

procedure TProgramForm.FormDestroy(Sender: TObject);

begin

Cleanup;

end;

procedure TProgramForm.CloseFileClick(Sender: TObject);

begin

CloseCurrentFile;

end;

procedure TProgramForm.ExitClick(Sender: TObject);

begin

Close;

end;

procedure TProgramForm.SaveClick(Sender: TObject);

begin

FilesManager.UpdateActiveFileContents;

FilesManager.SaveActiveFile;

end;

procedure TProgramForm.FormKeyPress(Sender: TObject; var Key: Char);

begin

if not (GetKeyState(VK\_LCONTROL) and $8000 = $8000) then

CodeField.TypeLetter(Key);

end;

procedure TProgramForm.FormKeyDown(Sender: TObject; var Key: Word; Shift: TShiftState);

var flags: Byte;

begin

flags := 0;

if (GetKeyState(VK\_LCONTROL) and $8000 = $8000) then

flags := (flags or KEY\_FLAG\_LCTRL);

if (GetKeyState(VK\_LSHIFT) and $8000 = $8000) then

flags := (flags or KEY\_FLAG\_LSHIFT);

case (Key) of

VK\_LEFT:

CodeField.MoveCaret(DIR\_LEFT, flags);

VK\_RIGHT:

CodeField.MoveCaret(DIR\_RIGHT, flags);

VK\_UP:

CodeField.MoveCaret(DIR\_UP, flags);

VK\_DOWN:

CodeField.MoveCaret(DIR\_DOWN, flags);

VK\_DELETE:

CodeField.OnDeletePressed;

VK\_HOME:

CodeField.OnHomePressed((flags and KEY\_FLAG\_LSHIFT) <> 0);

VK\_END:

CodeField.OnEndPressed((flags and KEY\_FLAG\_LSHIFT) <> 0);

end;

end;

procedure TProgramForm.BlinkTimerTimer(Sender: TObject);

begin

// Make blinking

Inc(BlinkTimerValue);

if (BlinkTimerValue mod BlinkFrequency = 0) then

CodeField.CaretObj.Visible := not TextCaret.Visible;

end;

procedure TProgramForm.SelectAllClick(Sender: TObject);

begin

CodeField.SelectAll;

end;

procedure TProgramForm.DeleteClick(Sender: TObject);

begin

CodeField.OnDeletePressed;

end;

procedure TProgramForm.CopyClick(Sender: TObject);

begin

CodeField.CopySelection;

end;

procedure TProgramForm.CutClick(Sender: TObject);

begin

CodeField.CutToClipboard;

end;

procedure TProgramForm.PasteClick(Sender: TObject);

begin

CodeField.PasteFromClipboard;

end;

procedure TProgramForm.NewMenuBtnClick(Sender: TObject);

begin

FilenamesTabCanChange := False;

// Save current view to buffer

FilesManager.UpdateActiveFileContents;

FilesManager.AddFile(DEFAULT\_NEWFILE\_NAME, EMPTY\_STRING);

FilesManager.ActiveView := FilesManager.GetFilesCount - 1;

UpdateFilenamesTab;

UpdateActiveTextField;

FilenamesTabCanChange := True;

end;

procedure TProgramForm.SaveAsClick(Sender: TObject);

var path : String;

begin

// On File Save Dialog

if SaveFileDialog.Execute then

begin

path := SaveFileDialog.FileName;

FilesManager.UpdateActiveFileContents;

FilesManager.SaveActiveFileTo(path);

end;

end;

procedure TProgramForm.OpenFileBitBtnClick(Sender: TObject);

begin

OpenClick(nil);

end;

procedure TProgramForm.NewFileBitBtnClick(Sender: TObject);

begin

NewMenuBtnClick(nil);

end;

procedure TProgramForm.CloseFileBitBtnClick(Sender: TObject);

begin

CloseFileClick(nil);

end;

procedure TProgramForm.AboutMenuBtnClick(Sender: TObject);

begin

// Display program info

MessageBox(

handle,

ABOUT\_TEXT,

PChar('About'),

MB\_ICONINFORMATION + MB\_OK

);

end;

procedure TProgramForm.CutBitBtnClick(Sender: TObject);

begin

CutClick(nil);

end;

procedure TProgramForm.CopyBitBtnClick(Sender: TObject);

begin

CopyClick(nil);

end;

procedure TProgramForm.PasteBitBtnClick(Sender: TObject);

begin

PasteClick(nil);

end;

procedure TProgramForm.SpellCheckBitBtnClick(Sender: TObject);

begin

// Spell-check

CodeToCheck := CodeField.FormattedCode;

CheckedFileName := FilesManager.GetActiveFileName;

ClearCheckMessages;

RefreshXmlDocument;

XmlCodeFrame.ShowModal;

end;

procedure TProgramForm.UndoClick(Sender: TObject);

begin

CodeField.Undo;

end;

procedure TProgramForm.RedoClick(Sender: TObject);

begin

CodeField.Redo;

end;

procedure TProgramForm.SaveBitBtnClick(Sender: TObject);

begin

SaveClick(nil);

end;

procedure TProgramForm.CodeScrollVerticalChange(Sender: TObject);

begin

if VerticalScrollChange then

begin

CodeField.StartLine := CodeScrollVertical.Position;

TextLinesCounter.StartLine := CodeScrollVertical.Position;

end;

end;

procedure TProgramForm.FormMouseWheelDown(Sender: TObject;

Shift: TShiftState; MousePos: TPoint; var Handled: Boolean);

begin

if VerticalScrollChange then

begin

CodeScrollVertical.Position := CodeScrollVertical.Position + 3;

CodeField.StartLine := CodeScrollVertical.Position;

TextLinesCounter.StartLine := CodeScrollVertical.Position;

end;

end;

procedure TProgramForm.FormMouseWheelUp(Sender: TObject;

Shift: TShiftState; MousePos: TPoint; var Handled: Boolean);

begin

if VerticalScrollChange then

begin

CodeScrollVertical.Position := CodeScrollVertical.Position - 3;

CodeField.StartLine := CodeScrollVertical.Position;

TextLinesCounter.StartLine := CodeScrollVertical.Position;

end;

end;

procedure TProgramForm.FormClose(Sender: TObject;

var Action: TCloseAction);

var messageBoxResult : Word;

begin

messageBoxResult := MessageBox(Handle, EXIT\_MESSAGE, CAUTION\_TEXT, MB\_YESNO + MB\_ICONINFORMATION);

if (messageBoxResult = IDNo) then

Action := caNone;

end;

end.

## Файл «TCodeTextField.pas»

type

// Stack type

TEditHistoryElementPtr = ^TEditHistoryElement;

TEditHistoryElement = record

editType : Byte;

text : String;

point : TPoint;

end;

TStackPtr = ^TStack;

TStackElementPtr = ^TStackElement;

TStack = record

top: TStackElementPtr;

end;

TStackElement = record

data: TEditHistoryElementPtr;

above, below: TStackElementPtr;

end;

function CreateStack : TStackPtr;

procedure PushElementToStack(data: TEditHistoryElementPtr; stack: TStackPtr);

function PopElementFromStack(stack: TStackPtr) : TEditHistoryElementPtr;

procedure MoveToStackTop(stack: TStackPtr);

type

TTextCaret = class (TImage)

private

fRealHeight: Word;

procedure InitCaret;

public

constructor Create(Control: TControl); reintroduce; overload;

procedure SetHeight(height: Word);

protected

published

property RealHeight: Word read fRealHeight write fRealHeight;

end;

TCodeTextField = class(TImage)

private

fLines : TStringList;

fLineStyles : TStringList;

fWidgetFont : TFont;

fLineDisplayStart : LongInt;

fDisplayLength : LongInt;

fSize : TPoint;

fBackColor : TColor;

fSelectionColor : TColor;

fCaretPos : TPoint;

fCaretObject : TTextCaret;

fSelStart : TPoint;

fSelEnd : TPoint;

fFontSize : TPoint;

fEditStack : TStackPtr;

procedure InitTextField;

procedure SetSize(size: TPoint);

procedure RedrawLines;

procedure RedrawLinesFromCaretPos;

procedure RedrawLine(line: LongInt);

procedure SetTextStyle(style: Byte);

function ComparePositions(p1, p2: TPoint) : Integer;

function GetCharStyle(pos: TPoint) : Byte;

procedure PushToEditStack(editType: Byte; stringInfo: String; pointInfo: TPoint);

procedure ResetLineStyle(line: LongInt);

procedure SetLineStyle(line, \_from, \_to: LongInt; style: Byte); overload;

procedure SetLineStyle(line: LongInt; style: String); overload;

procedure OnLineTextChanged(line: LongInt);

procedure RefreshLineStyles(\_from, \_to: LongInt);

function IsPreviousLineMultilineComment(line: LongInt) : Boolean;

procedure ProcessStrings(line: LongInt);

procedure ProcessKeywords(line: LongInt);

procedure ProcessDigits(line: LongInt);

// Input methods

procedure ResetSelection;

procedure AppendChar(c: Char);

procedure ProcessBackSpace;

procedure ProcessEnter;

function IsCharSelected(pos: TPoint) : Boolean;

function GetFormattedCode : String;

protected

{ Protected declarations }

public

constructor Create(Control: TControl); reintroduce; overload;

procedure Refresh;

procedure RefreshCaretPos;

function GetLineCount : Integer;

procedure TypeLetter(c: Char);

procedure MoveCaret(dir, flags: Byte);

procedure SetCaretPos(pos: TPoint);

procedure SetStartLine(line: LongInt);

procedure SetDisplayLength(len: LongInt);

function GetSelectionLength : LongInt;

function GetSelectionText : String;

procedure SelectAll;

procedure OnDeletePressed;

procedure OnHomePressed(shiftPressed: Boolean);

procedure OnEndPressed(shiftPressed: Boolean);

procedure RemoveSelection;

procedure CopySelection;

procedure CutToClipboard;

procedure PasteFromClipboard;

procedure LoadText(text: String);

function GetText : String;

procedure SetWidgetFont(font: TFont);

procedure OnTextFieldPress(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

procedure Undo;

procedure Redo;

function GetLastDisplayedLine : LongInt;

published

property Lines: TStringList read fLines;

property LineStyles: TStringList read fLineStyles;

property LinesCount: Integer read GetLineCount;

property Font: TFont read fWidgetFont write SetWidgetFont;

property WidgetSize: TPoint read fSize write SetSize;

property CaretPos: TPoint read fCaretPos write SetCaretPos;

property CaretObj: TTextCaret read fCaretObject write fCaretObject;

property SelectionColor: TColor read fSelectionColor write fSelectionColor;

property StartLine: LongInt read fLineDisplayStart write SetStartLine;

property DisplayLength: LongInt read fDisplayLength write SetDisplayLength;

property FormattedCode : String read GetFormattedCode;

end;

## Файл «TCodeTextField\_implement.pas»

constructor TCodeTextField.Create(Control: TControl);

begin

inherited Create(Control);

InitTextField;

end;

procedure TCodeTextField.InitTextField;

begin

fLines := TStringList.Create;

fLineStyles := TStringList.Create;

fLineDisplayStart := 0;

fDisplayLength := 1;

Cursor := crIBeam;

fFontSize := Point(20, 10);

fSize := Point(0, 0);

fCaretPos := Point(0, 0);

fSelStart := fCaretPos;

fSelEnd := fCaretPos;

fBackColor := clWhite;

fEditStack := CreateStack;

OnMouseDown := OnTextFieldPress;

end;

procedure TCodeTextField.SetSize(size: TPoint);

begin

fSize := size;

Refresh;

end;

procedure TCodeTextField.TypeLetter(c: Char);

begin

if (Ord(c) = KEY\_BACKSPACE) then

ProcessBackSpace

else if (Ord(c) = KEY\_ENTER) then

ProcessEnter

else if (Ord(c) >= KEY\_SPACE) then

AppendChar(c);

RefreshCaretPos;

end;

procedure TCodeTextField.SelectAll;

begin

fCaretPos.y := fLines.Count - 1;

fCaretPos.x := Length(fLines[fCaretPos.y]);

fSelStart := Point(0, 0);

fSelEnd := fCaretPos;

Refresh;

RefreshCaretPos;

end;

procedure TCodeTextField.OnDeletePressed;

var tempString: String;

begin

if (GetSelectionLength > 0) then

RemoveSelection

else if (fCaretPos.x < Length(fLines[fCaretPos.y])) then

begin

tempString := Copy(fLines[fCaretPos.y], 0, fCaretPos.x) + Copy(fLines[fCaretPos.y], fCaretPos.x + 2, Length(fLines[fCaretPos.y]));

fLines[fCaretPos.y] := tempString;

tempString := Copy(fLineStyles[fCaretPos.y], 0, fCaretPos.x) + Copy(fLineStyles[fCaretPos.y], fCaretPos.x + 2, Length(fLineStyles[fCaretPos.y]));

SetLineStyle(fCaretPos.y, tempString);

OnLineTextChanged(fCaretPos.y);

RedrawLine(fCaretPos.y);

end

else if (fCaretPos.y < fLines.Count - 1) then

begin

tempString := fLines[fCaretPos.y] + fLines[fCaretPos.y + 1];

fLines[fCaretPos.y] := tempString;

fLines.Delete(fCaretPos.y + 1);

fLineStyles.Delete(fCaretPos.y + 1);

OnLineTextChanged(fCaretPos.y);

RedrawLinesFromCaretPos;

end;

RefreshCaretPos;

UpdateCodeVerticalScrollRange(fLines.Count - 1);

end;

// Procedure to process caret (arrow keys movements)

procedure TCodeTextField.MoveCaret(dir, flags: Byte);

var isSelecting : Boolean;

caretMoveDir : Integer;

posOld : TPoint;

begin

posOld := fCaretPos;

isSelecting := (flags and KEY\_FLAG\_LSHIFT <> 0);

// Process caret movement

case dir of

DIR\_LEFT:

begin

if (fCaretPos.x > 0) then

Dec(fCaretPos.x)

else if (fCaretPos.y > 0) then

begin

Dec(fCaretPos.y);

fCaretPos.x := Length(fLines[fCaretPos.y]);

end;

end;

DIR\_RIGHT:

begin

if (fCaretPos.x < Length(fLines[fCaretPos.y])) then

Inc(fCaretPos.x)

else if (fCaretPos.y < fLines.Count - 1) then

begin

Inc(fCaretPos.y);

fCaretPos.x := 0;

end;

end;

DIR\_UP:

if (fCaretPos.y > 0) then

begin

Dec(fCaretPos.y);

if (Length(fLines[fCaretPos.y]) < fCaretPos.x) then

fCaretPos.x := Length(fLines[fCaretPos.y]);

end;

DIR\_DOWN:

if (fCaretPos.y < fLines.Count - 1) then

begin

Inc(fCaretPos.y);

if (Length(fLines[fCaretPos.y]) < fCaretPos.x) then

fCaretPos.x := Length(fLines[fCaretPos.y]);

end;

end;

// Refresh selection borders

if not isSelecting then

begin

fSelStart := fCaretPos;

fSelEnd := fCaretPos;

end

else

begin

// Define caret movement direction

caretMoveDir := comparePositions(fCaretPos, posOld);

if (caretMoveDir = 1) then

begin

if (comparePositions(fCaretPos, fSelEnd) = 1) then

fSelEnd := fCaretPos

else

fSelStart := fCaretPos;

end

else if (caretMoveDir = -1) then

begin

if (comparePositions(fCaretPos, fSelStart) = -1) then

fSelStart := fCaretPos

else

fSelEnd := fCaretPos;

end;

end;

RefreshCaretPos;

Refresh;

UpdateCodeVerticalScrollPos(fLineDisplayStart);

end;

function TCodeTextField.GetLineCount : Integer;

begin

Result := fLines.Count;

end;

procedure TCodeTextField.Refresh;

begin

Canvas.Brush.Color := ProgramForm.Color;

Canvas.Pen.Color := ProgramForm.Color;

Canvas.Rectangle(0, 0, Width, Height);

// Refill

Canvas.Pen.Color := clBlack;

Canvas.Brush.Color := fBackColor;

Canvas.Rectangle(0, 0, fSize.x, fSize.y);

RedrawLines;

end;

procedure TCodeTextField.RedrawLines;

var i: LongInt;

begin

for i := fLineDisplayStart to GetLastDisplayedLine do

RedrawLine(i);

end;

procedure TCodeTextField.RedrawLinesFromCaretPos;

var i: LongInt;

begin

for i := fCaretPos.y to GetLastDisplayedLine do

RedrawLine(i);

end;

procedure TCodeTextField.RedrawLine(line: LongInt);

var i : LongInt;

begin

// Draw only visible lines

if (line in [fLineDisplayStart..GetLastDisplayedLine]) then

begin

// Redraw background

Canvas.Brush.Color := clWhite;

Canvas.Pen.Color := clWhite;

Canvas.Rectangle(1, 4 + fFontSize.y \* line, fSize.x - 1, 4 + fFontSize.y \* (line + 1));

// Draw characters

for i := 1 to Length(fLines[line]) do

begin

SetTextStyle( GetCharStyle(Point(i, line)) );

if IsCharSelected(Point(i, line)) then

begin

Canvas.Brush.Color := fontStyles[TEXT\_STYLE\_SELECTED].backColor;

Canvas.Pen.Color := fontStyles[TEXT\_STYLE\_SELECTED].backColor;

Canvas.Rectangle(3 + fFontSize.x \* (i - 1), 4 + fFontSize.y \* line, 4 + fFontSize.x \* i, 4 + fFontSize.y \* (line + 1));

Canvas.Font.Color := fontStyles[TEXT\_STYLE\_SELECTED].textColor;

end;

if (fLines[line][i] > Chr(KEY\_SPACE)) then

Canvas.TextOut(5 + fFontSize.x \* (i - 1), 4 + fFontSize.y \* (line - fLineDisplayStart), fLines[line][i]);

end;

end;

end;

procedure TCodeTextField.RefreshCaretPos;

begin

if (fCaretPos.y > fLineDisplayStart + fDisplayLength - 2) then

fLineDisplayStart := fCaretPos.y - fDisplayLength + 1

else if (fCaretPos.y < fLineDisplayStart) then

fLineDisplayStart := fCaretPos.y;

fCaretObject.Left := Left + 5 + fCaretPos.x \* fFontSize.x;

fCaretObject.Top := Top + 4 + (fCaretPos.y - fLineDisplayStart) \* fFontSize.y;

// Force show caret

CodeField.CaretObj.Visible := True;

BlinkTimerValue := 0;

end;

function TCodeTextField.GetLastDisplayedLine : LongInt;

begin

Result := fLineDisplayStart + fDisplayLength - 1;

if (Result > fLines.Count - 1) then

Result := fLines.Count - 1;

end;

// Procedure to apply different font styles

procedure TCodeTextField.SetTextStyle(style: Byte);

var currentStyle : TFontStyle;

styleSet : set of fsBold..fsUnderline;

begin

if (style in [0..DELPHI\_STYLES\_COUNT - 1]) then

currentStyle := fontStyles[style]

else

currentStyle := fontStyles[TEXT\_STYLE\_NONE];

Canvas.Font.Name := currentStyle.fontName;

Canvas.Font.Size := currentStyle.fontSize;

Canvas.Font.Color := currentStyle.textColor;

Canvas.Brush.Color := currentStyle.backColor;

if (currentStyle.bold) then Include(styleSet, fsBold);

if (currentStyle.italic) then Include(styleSet, fsItalic);

if (currentStyle.underlined) then Include(styleSet, fsUnderline);

Canvas.Font.Style := styleSet;

end;

// Function to get selection length from selStart - selEnd points

function TCodeTextField.GetSelectionLength : LongInt;

var i: LongInt;

begin

Result := 0;

for i := fSelStart.y to fSelEnd.y do

begin

Inc(Result, Length(fLines[i]));

if (i = fSelStart.y) then

Dec(Result, fSelStart.x);

if (i = fSelEnd.y) then

Dec(Result, Length(fLines[i]) - fSelEnd.x);

end;

end;

function TCodeTextField.GetSelectionText : String;

var i, a, b: LongInt;

t: String;

begin

Result := '';

for i := fSelStart.y to fSelEnd.y do

begin

t := fLines[i];

a := 1;

b := Length(t);

if (i = fSelStart.y) then

a := fSelStart.x + 1; // Crop left side

if (i = fSelEnd.y) then

b := fSelEnd.x; // Crop left side

Result := Result + Copy(t, a, b - a + 1);

if (i <> fSelEnd.y) then

Result := Result + #13#10;

end;

end;

function TCodeTextField.ComparePositions(p1, p2: TPoint) : Integer;

begin

Result := 0;

if (p1.y = p2.y) then

begin

if (p1.x > p2.x) then

Result := 1

else if (p1.x < p2.x) then

Result := -1;

end

else if (p1.y > p2.y) then

Result := 1

else

Result := -1;

end;

procedure TCodeTextField.RemoveSelection;

var i : LongInt;

t : String;

begin

t := Copy(fLines[fSelStart.y], 0, fSelStart.x) + Copy(fLines[fSelEnd.y], fSelEnd.x + 1, Length(fLines[fSelEnd.y]));

fLines[fSelStart.y] := t;

for i := 1 to (fSelEnd.y - fSelStart.y) do

begin

fLines.Delete(fSelStart.y + 1);

fLineStyles.Delete(fSelStart.y + 1);

end;

fSelEnd := fSelStart;

fCaretPos := fSelStart;

RefreshCaretPos;

OnLineTextChanged(fCaretPos.y);

RedrawLinesFromCaretPos;

UpdateCodeVerticalScrollRange(fLines.Count - 1);

end;

procedure TCodeTextField.CopySelection;

begin

Clipboard.AsText := GetSelectionText;

end;

procedure TCodeTextField.CutToClipboard;

begin

CopySelection;

RemoveSelection;

end;

procedure TCodeTextField.PasteFromClipboard;

var i, n: LongInt;

s, t: String;

list: TStringList;

begin

RemoveSelection;

s := Clipboard.AsText;

list := TStringList.Create;

repeat

n := Pos(#13#10, s);

if (n <> 0) then

begin

list.Add(Copy(s, 0, n - 1));

s := Copy(s, n + 2, Length(s));

end

else

begin

list.Add(s);

s := '';

end;

until (n = 0);

if (list.Count > 0) then

begin

t := Copy(fLines[fCaretPos.y], fCaretPos.x + 1, Length(fLines[fCaretPos.y]));

s := Copy(fLines[fCaretPos.y], 0, fCaretPos.x) + list[0];

fLines[fCaretPos.y] := s;

for i := 1 to list.Count - 1 do

begin

fLines.Insert(fCaretPos.y + i, list[i]);

end;

s := fLines[fCaretPos.y + (list.Count - 1)] + t;

fLines[fCaretPos.y + (list.Count - 1)] := s;

end;

RefreshCaretPos;

Refresh;

end;

procedure TCodeTextField.OnHomePressed(shiftPressed: Boolean);

begin

fCaretPos.x := 0;

fSelStart := fCaretPos;

if not shiftPressed then

fSelEnd := fCaretPos;

RefreshCaretPos;

end;

procedure TCodeTextField.OnEndPressed(shiftPressed: Boolean);

begin

fCaretPos.x := Length(fLines[fCaretPos.y]);

fSelEnd := fCaretPos;

if not shiftPressed then

fSelStart := fCaretPos;

RefreshCaretPos;

end;

procedure TCodeTextField.SetCaretPos(pos: TPoint);

begin

if (fLines.Count = 0) then

pos := Point(0, 0)

else if (pos.y >= fLines.Count) then

begin

pos.y := fLines.Count - 1;

pos.x := Length(fLines[pos.y]);

end;

if ((fLines.Count > 0) and (pos.x > Length(fLines[pos.y]))) then

pos.x := Length(fLines[pos.y]);

fCaretPos := pos;

fSelStart := pos;

fSelEnd := pos;

RefreshCaretPos;

end;

procedure TCodeTextField.OnTextFieldPress(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer);

var i, j: LongInt;

begin

i := (Y - 4) div fFontSize.y;

j := X div fFontSize.x;

if (i >= fDisplayLength) then

i := fDisplayLength - 1;

Inc(i, fLineDisplayStart);

SetCaretPos(Point(j, i));

end;

procedure TCodeTextField.LoadText(text: String);

var i, n : LongInt;

tempString : String;

begin

fSelStart := Point(0, 0);

fSelEnd := fSelStart;

fLines.Clear;

fLineStyles.Clear;

if (Length(text) > 0) then

begin

repeat

n := Pos(#13#10, text);

if (n <> 0) then

begin

tempString := Copy(text, 1, n - 1);

fLines.Add(tempString);

for i := 1 to Length(tempString) do

tempString[i] := TextStyle(TEXT\_STYLE\_NONE);

fLineStyles.Add(tempString);

text := Copy(text, n + 2, Length(text));

end

else

begin

fLines.Add(text);

tempString := text;

for i := 1 to Length(tempString) do

tempString[i] := TextStyle(TEXT\_STYLE\_NONE);

fLineStyles.Add(tempString);

text := '';

end;

until (n = 0);

end;

if (fLines.Count = 0) then

begin

fLines.Add(EMPTY\_STRING);

fLineStyles.Add(EMPTY\_STRING);

end;

fCaretPos := Point(0, 0);

fSelStart := fCaretPos;

fSelEnd := fCaretPos;

// Rebuild code highlight

for i := 0 to fLines.Count - 1 do

OnLineTextChanged(i);

Refresh;

RefreshCaretPos;

UpdateCodeVerticalScrollRange(fLines.Count - 1);

end;

function TCodeTextField.GetText : String;

var i: LongInt;

begin

Result := '';

for i := 0 to fLines.Count - 2 do

Result := Result + fLines[i] + #13#10;

if (fLines.Count > 0) then

Result := Result + fLines[fLines.Count - 1];

end;

procedure TCodeTextField.SetStartLine(line: LongInt);

begin

fLineDisplayStart := line;

if ((fCaretPos.y < fLineDisplayStart) or (fCaretPos.y > fLineDisplayStart + fDisplayLength)) then

fCaretObject.Left := -10

else

RefreshCaretPos;

Refresh;

end;

procedure TCodeTextField.SetDisplayLength(len: LongInt);

begin

fDisplayLength := len;

RefreshCaretPos;

Refresh;

UpdateCodeVerticalScrollRange(fLines.Count - 1);

end;

function TCodeTextField.GetCharStyle(pos: TPoint) : Byte;

begin

if ((pos.y >= fLineStyles.Count) or (pos.x > Length(fLineStyles[pos.y]))) then

Result := TEXT\_STYLE\_NONE

else

Result := Ord(fLineStyles[pos.y][pos.x]);

end;

procedure TCodeTextField.SetWidgetFont(font: TFont);

begin

fWidgetFont := font;

Canvas.Font := fWidgetFont;

fFontSize := Point(

Canvas.TextWidth('#') + CHAR\_SPACING,

-fWidgetFont.Height + LINE\_SPACING

);

end;

procedure TCodeTextField.Undo;

begin

end;

procedure TCodeTextField.Redo;

begin

end;

procedure TCodeTextField.PushToEditStack(editType: Byte; stringInfo: String; pointInfo: TPoint);

var historyElement: TEditHistoryElementPtr;

begin

New(historyElement);

historyElement^.editType := editType;

historyElement^.text := stringInfo;

historyElement^.point := pointInfo;

PushElementToStack(historyElement, fEditStack);

end;

procedure TCodeTextField.ResetLineStyle(line: LongInt);

var i : LongInt;

tempString : String;

begin

if ((line < fLineStyles.Count) and (line < fLines.Count)) then

begin

SetLength(tempString, Length(fLines[line]));

for i := 1 to Length(tempString) do

tempString[i] := TextStyle(TEXT\_STYLE\_NONE);

fLineStyles[line] := tempString;

end;

end;

procedure TCodeTextField.SetLineStyle(line, \_from, \_to: LongInt; style: Byte);

var i, len : LongInt;

tempString : String;

begin

len := Length(fLines[line]);

if (\_from < 1) then

\_from := 1;

if (\_to > len) then

\_to := len;

tempString := fLineStyles[line];

for i := \_from to \_to do

tempString[i] := TextStyle(style);

fLineStyles[line] := tempString;

end;

procedure TCodeTextField.SetLineStyle(line: LongInt; style: String);

begin

fLineStyles[line] := style;

end;

procedure TCodeTextField.OnLineTextChanged(line: LongInt);

var lastChar : Char;

begin

lastChar := #0;

if ((line in [0..fLineStyles.Count - 1]) and (Length(fLineStyles[line]) > 0)) then

lastChar := fLineStyles[line][Length(fLineStyles[line])];

ResetLineStyle(line);

if (line in [0..fLineStyles.Count - 1]) then

begin

ProcessStrings(line);

ProcessKeywords(line);

ProcessDigits(line);

if (

(Length(fLineStyles[line]) = 0)

or

(fLineStyles[line][Length(fLineStyles[line])] = TextStyle(TEXT\_STYLE\_MULTILINE\_COMMENT))

or

(

(lastChar = TextStyle(TEXT\_STYLE\_MULTILINE\_COMMENT))

and

(lastChar <> fLineStyles[line][Length(fLineStyles[line])])

)

) then

OnLineTextChanged(line + 1);

end;

RedrawLine(line);

end;

procedure TCodeTextField.RefreshLineStyles(\_from, \_to: LongInt);

var i : LongInt;

begin

for i := \_from to \_to do

OnLineTextChanged(i);

end;

function TCodeTextField.IsPreviousLineMultilineComment(line: LongInt) : Boolean;

begin

Result := False;

if (line in [1..fLines.Count - 1]) then

begin

Dec(line);

if (Length(fLineStyles[line]) > 0) then

Result := (

(fLineStyles[line][Length(fLineStyles[line])] = TextStyle(TEXT\_STYLE\_MULTILINE\_COMMENT))

and

(fLines[line][Length(fLines[line])] <> DELPHI\_MULTILINE\_CLOSER)

)

else

Result := IsPreviousLineMultilineComment(line);

end;

end;

procedure TCodeTextField.AppendChar(c: Char);

var tempString : String;

len, i : LongInt;

begin

if (fCaretPos.y in [0..fLines.Count - 1]) then

begin

// Expand line

tempString := fLines[fCaretPos.y];

SetLength(tempString, Length(tempString) + 1);

// Process 'out of range' situation

len := Length(fLines[fCaretPos.y]);

if (fCaretPos.x > len) then

fCaretPos.x := len;

// Shift text to the right

for i := Length(tempString) downto fCaretPos.x + 1 do

tempString[i] := tempString[i - 1];

Inc(fCaretPos.x);

tempString[fCaretPos.x] := c;

fLines[fCaretPos.y] := tempString;

// Call line style rebuild

OnLineTextChanged(fCaretPos.y);

Refresh;

end;

end;

procedure TCodeTextField.ProcessBackSpace;

var tempString : String;

i : LongInt;

begin

if (GetSelectionLength > 0) then

RemoveSelection

else if (fCaretPos.x > 0) then

begin

Dec(fCaretPos.x);

tempString := fLines[fCaretPos.y];

// Shift characters to the left

for i := fCaretPos.x + 1 to Length(tempString) - 1 do

tempString[i] := tempString[i + 1];

// Update string

SetLength(tempString, Length(tempString) - 1);

fLines[fCaretPos.y] := tempString;

// Rebuild line style

OnLineTextChanged(fCaretPos.y);

end

else if (fCaretPos.y > 0) then

begin

fCaretPos.x := Length(fLines[fCaretPos.y - 1]);

tempString := fLines[fCaretPos.y - 1] + fLines[fCaretPos.y];

fLines[fCaretPos.y - 1] := tempString;

// Remove line from lists

fLines.Delete(fCaretPos.y);

fLineStyles.Delete(fCaretPos.y);

Dec(fCaretPos.y);

// Rebuild line style

OnLineTextChanged(fCaretPos.y);

RedrawLinesFromCaretPos;

end;

ResetSelection;

Refresh;

UpdateCodeVerticalScrollRange(fLines.Count - 1);

end;

procedure TCodeTextField.ResetSelection;

begin

fSelStart := fCaretPos;

fSelEnd := fCaretPos;

end;

procedure TCodeTextField.ProcessStrings(line: LongInt);

var currentState : Byte;

i : LongInt;

tempString : String;

begin

currentState := TEXT\_STYLE\_NONE;

if IsPreviousLineMultilineComment(line) then

currentState := TEXT\_STYLE\_MULTILINE\_COMMENT;

tempString := fLineStyles[line];

for i := 1 to Length(fLines[line]) do

begin

if (currentState = TEXT\_STYLE\_NONE) then

begin

if (fLines[line][i] = DELPHI\_MULTILINE\_OPENER) then

currentState := TEXT\_STYLE\_MULTILINE\_COMMENT

else if (fLines[line][i] = DELPHI\_STRING\_BORDER) then

currentState := TEXT\_STYLE\_STRING

else if ((i < Length(fLines[line])) and (Copy(fLines[line], i, 2) = DELPHI\_COMMENT\_BEGINNER)) then

currentState := TEXT\_STYLE\_COMMENT;

tempString[i] := TextStyle(currentState);

end

else

begin

tempString[i] := TextStyle(currentState);

if (currentState = TEXT\_STYLE\_STRING) then

if (fLines[line][i] = DELPHI\_STRING\_BORDER) then

currentState := TEXT\_STYLE\_NONE;

if (currentState = TEXT\_STYLE\_MULTILINE\_COMMENT) then

if (fLines[line][i] = DELPHI\_MULTILINE\_CLOSER) then

currentState := TEXT\_STYLE\_NONE;

end;

end;

SetLineStyle(line, tempString);

end;

procedure TCodeTextField.ProcessKeywords(line: LongInt);

var keywordAllowed : Boolean;

c : Char;

currentList : TUserListPtr;

currentElement : TUserListElementPtr;

i, j : LongInt;

breakerElement : TUserListElementPtr;

begin

i := 1;

while (i <= Length(fLines[line])) do

begin

keywordAllowed := (i = 1) or ((not IsSymbol(fLines[line][i - 1]) and (not IsDigit(fLines[line][i - 1]))));

keywordAllowed := keywordAllowed and (fLineStyles[line][i] = TextStyle(TEXT\_STYLE\_NONE));

if (IsSymbol(fLines[line][i]) and keywordAllowed) then

begin

c := UpCase(fLines[line][i]);

currentList := keywordsList;

currentElement := FindElement(c, currentList);

if Assigned(currentElement) then

begin

j := 1;

while Assigned(currentElement) do

begin

currentList := currentElement^.childList;

breakerElement := FindElement(KEYWORD\_BREAKER, currentList);

if (

Assigned(breakerElement) and

(not IsSymbol(fLines[line][i + j])) and

(not IsDigit(fLines[line][i + j]))

) then

begin

SetLineStyle(line, i, i + j - 1, TEXT\_STYLE\_KEYWORD);

Inc(i, j - 1);

break;

end

else

begin

c := UpCase(fLines[line][i + j]);

currentElement := FindElement(c, currentList);

if Assigned(currentElement) then

Inc(j)

else

continue;

end;

end;

end;

end;

Inc(i);

end;

end;

procedure TCodeTextField.ProcessDigits(line: LongInt);

var i : LongInt;

tempString : String;

digitState : Boolean;

begin

tempString := fLineStyles[line];

digitState := False;

for i := 1 to Length(fLines[line]) do

begin

if (tempString[i] = TextStyle(TEXT\_STYLE\_NONE)) then

if IsDigit(fLines[line][i]) then

begin

if (

(not digitState)

and

(

(i = 1)

or

((not IsSymbol(fLines[line][i - 1])) and (not IsDigit(fLines[line][i - 1])))

)

) then

begin

digitState := True;

tempString[i] := TextStyle(TEXT\_STYLE\_NUMBER);

end

else if digitState then

tempString[i] := TextStyle(TEXT\_STYLE\_NUMBER);

end

else

begin

digitState := False;

end;

end;

SetLineStyle(line, tempString);

end;

procedure TCodeTextField.ProcessEnter;

var lineNew : String;

lineOld : String;

lineStyle : String;

begin

lineNew := Copy(fLines[fCaretPos.y], fCaretPos.x + 1, Length(fLines[fCaretPos.y]));

lineOld := Copy(fLines[fCaretPos.y], 0, fCaretPos.x);

fLines[fCaretPos.y] := lineOld;

SetLength(lineStyle, Length(lineNew));

OnLineTextChanged(fCaretPos.y);

if (fCaretPos.y = fLines.Count - 1) then

begin

// Add to the end

fLines.Add(lineNew);

fLineStyles.Add(lineStyle);

end

else

begin

// Insert string into the list

fLines.Insert(fCaretPos.y + 1, lineNew);

fLineStyles.Insert(fCaretPos.y + 1, lineStyle);

end;

OnLineTextChanged(fCaretPos.y + 1);

// Update caret pos

fCaretPos.x := 0;

Inc(fCaretPos.y);

// Refresh line styles

Refresh;

end;

function TCodeTextField.IsCharSelected(pos: TPoint) : Boolean;

var posStart, posEnd : TPoint;

begin

posStart := Point(fSelStart.x, fSelStart.y);

posEnd := Point(fSelEnd.x + 1, fSelEnd.y);

Result := ((comparePositions(pos, posStart) > 0) and (comparePositions(pos, posEnd) < 0));

end;

function TCodeTextField.GetFormattedCode : String;

var i, j : LongInt;

isSpace : Boolean;

begin

Result := EMPTY\_STRING;

for i := 0 to fLines.Count - 1 do

begin

isSpace := True;

for j := 1 to Length(fLines[i]) do

begin

if (

(

(Ord(fLines[i][j]) = KEY\_SPACE)

and

(not isSpace)

)

or

(Ord(fLines[i][j]) >= KEY\_SPACE)

) then

Result := Result + fLines[i][j];

isSpace := Ord(fLines[i][j]) = KEY\_SPACE;

end;

if (Length(fLines[i]) > 0) then

Result := Result + #13#10;

end;

end;

constructor TTextCaret.Create(Control: TControl);

begin

inherited Create(Control);

InitCaret;

end;

procedure TTextCaret.InitCaret;

begin

Width := 1;

Height := 22;

SetHeight(Height);

Cursor := crIBeam;

end;

procedure TTextCaret.SetHeight(height: Word);

begin

fRealHeight := height;

Canvas.Brush.Color := ProgramForm.Color;

Canvas.Rectangle(0, 0, Width, Height);

end;