Slit - Simple Literate Tool

Leonardo Cecchi

Contents

Introduzione

Slit is a software which can write code and documentation into the same file. The relation between code and documentation is inverted: if usually we write the code and into this we put the documentation using Slit we write the documentation and into that the code.

the code can be separed in pieces that compose the complete software too. To have other information to this modality of work we can consult the page

`http://en.wikipedia.org/wiki/Literate_programming`

Capitolo 1. User guide

Slit's files are composed of text's lines of documentation and instructions. Instructions are lines that starts by a @ character and are followed by a letter which determine the directive.

There is a very important thing to remember: the directive is valid only if it is written at the first character of the line. If you want you can make Slit ignore a directive inserting a space before the *at* character.

1.1. The definition directive: @d

The directive @d is the primary instruction of Slit: this instruction is used to memorize a text's macro into the sistem. the macro can be used into others macro or only to write a sourced file.

this is an example of a directive @d:

```
@d function sum

function sum (a,b:Integer) return Integer
begin
  return a+b;
end;
```

How we can see the directive @d is followed to an *scrap*. An *scrap*, in the Slit terminology, is the content of the defined macro.

A scrap starts with @{ and is finished by @}. The text between this two lines is the content of the macro that will be defined.

According to the type of processor selected for the output Slit enounce the name of the macro and its content into the documentation.

To each macro is associate a number that identify it and at the and of the macro are enounce the number of macro that use the just define macro.

A macro can be not used too, in this case Slit advise the user at the end of the process of production of source files.

1.2. The directive @o and the syntax of scraps

The directive @o is very similar, according to the syntactic point of view, to that @d. Infact is used to define a particular type of macro, which name is used to write a file.

For example:

```
@o prova.c
@{
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello world!\n");
    return 0;
}
@}
```

Into a scrap we can include a reference to an other:

```
@d saluta
@{
printf("Hello world!\n");
@}


@o provadue.c
@{
#include <stdio.h>

int main(int argc, char **argv)
{
    @<saluta@>
    return 0;
}
@}
```

Slit remembers the source code indentation level and uses that information to write the target source code.

1.3. @i directive

The @i directive can be used to include a slit file in another slit file.

Often, the software written using Slit, are composed by more Slit files. The @i directive make Slit read another source file and return to the caller file when the included file is readden.

For example:

```
@i provatre_funzioni.s


@i provatre_dichiarazioni.s


@o provatre.c
@{
```

```
@include <stdio.h>

@<dichiarazioni@>
@<funzioni@>

int main(int argc, char **argv)
{
    funzionePrincipale();
    return 0;
}
@}
```

1.4. @+ directive

The `@+` directive append a scrap to the end of an already existent one. This behaviour is useful when you need to reduce the used macro names.

For example:

```
@o provatre.c
@{
@include <stdio.h>

@<dichiarazioni@>

int main(int argc, char **argv)
{
    funzionePrincipale();
    return 0;
}
@}


@d dichiarazioni
@{
void funzioneUno();
@}


@+ dichiarazioni
@{
void funzionePrincipale();
@}
```

1.5. Options (@x directive)

Slit can be configured via options. Options are configurated using the `@x` directive.

This directive must be followed by a string whi is the name of the option being configured. For example:

```
@x output_html
```

This is the list of the supported options:

- `output_html` : select HTML as output format
- `output_txt` : select TXT as output format
- `output_lout` : select LOUT as output format
- `section_markers` : enable start or end section markers in source code
- `no_section_markers` : disable start or end section markers in source code
- `line_markers` : enable line markers in source code
- `no_line_markers` : disable line markers in source code
- `comment_markers` : this options customize the comment markers for programming languages not already supported by Slit or can be used to personalize existing languages support.

The option `comment_markers` is followed by a parameter saying the programming language and the comment start and end marker. This parameter has the syntax `<separator><extension><separator><comment-start><separator><comment-end>`, where the separator can be any character.

For example, the predefined configuration for *Pascal* can be written like this: .

1.6. Documentation

All that is not a directory goes in the generated documentation without any transformation.

Slit must be used with a documentation syntax. For now the following languages can be used:

- HTML language,
- plain old text,
- the Lout typesetting syntax.

The documentation can must be written using the choosen format.

The Lout format will also generate a cross-reference between the scraps: every scrap is numbered and, at the end, the list of all the using scraps is generated automatically.

Capitolo 2. The Slit command

Slit programs can be written in text files which can have every extension. To separate the code from the documentation you can use the `slit` command:

```
slit <nomefile>
```

This command will process the file passed and will process all the directives. The `slit` command generates:

- the documentation files;
- the source code.

The main procedure is like this:

⟨ ***slit procedura principale*** *1* ⟩≡

```
begin
  if ParamCount = 1 then
  begin
    <slit preparazione dell’ambiente 2>
    <slit riempimento del magazzino delle macro 3>
    <slit calcola riferimenti 4>
    <slit controlla macro non utilizzate 5>
    <slit generazione della documentazione 7>
    <slit generazione del codice sorgente 8>
    <slit pulizia 9>
  end
  else
  begin
    writeln(’Use: ’, ParamStr(0), ’ <nomefile>’);
  end;
end.
```

Usata da: 97

The input file is represented by an object of the class `TSlitStream` ant the documentation file is represented by an object of the class `TSlitOutputTxt`. Macros are memorized in a *store* and are retrieved when the source code is generated.

⟨ ***slit preparazione dell’ambiente*** *2* ⟩≡

```
store := TMacroStore.Create;
stream := TSlitStream.CreateForFile(ParamStr(1));
driverMagazzinoMacro := TSlitStreamDriverMagazzino.CreateWithMacroStore( store );
```

Usata da: 1

Documentation is read by a method in the input stream:

⟨ ***slit riempimento del magazzino delle macro*** *3* ⟩≡

```
stream.Driver := driverMagazzinoMacro;
stream.Process();
stream.ResetStream();
```

Usata da: 1

When the macro store is populated whe cross-reference is calculated:

⟨ ***slit calcola riferimenti*** *4* ⟩≡

```
store.CalcolaRiferimenti();
```

Usata da: 1

When the cross-reference is calculated is possible to check unreferenced macros:

⟨ ***slit controlla macro non utilizzate*** *5* ⟩≡

```
ControllaMacroNonUtilizzate();
```

Usata da: 1

This check is done reading all the macro store:

⟨ ***slit ControllaMacroNonUtilizzate*** *6* ⟩≡

```
procedure ControllaMacroNonUtilizzate;
var
  tempMacro : TMacroRecord;
  i : integer;
begin
  for i := 0 to store.MacroCount-1 do
  begin
    tempMacro := store.GetRecord( i );
    if (tempMacro.macroUsersCount = 0) and (tempMacro.macroType <> FileMacro) then
    begin
      LogErrorMessage('The macro ' + tempMacro.macroName + ' is never used.');
    end;
  end;
end;
```

Usata da: 97

Now the documentation can be generated:

⟨ ***slit generazione della documentazione*** *7* ⟩≡

```
streamOutputDocumentazione := CreaStreamOutputDaOpzioni(ParamStr(1), store);
driverScriviDocumentazione :=
  TSlitStreamDriverGenerazioneDoc.CreateWithOutputStream( streamOutputDocumentazione );

stream.Driver := driverScriviDocumentazione;
stream.Process();
stream.ResetStream();
writeln(store.MacroCount, ’ macro processate’);
```

Usata da: 1

Then the source files are generated:

⟨ ***slit generazione del codice sorgente*** *8* ⟩≡

```
ProcessaFiles();
```

Usata da: 1

And now a bit of cleaning:

⟨ ***slit pulizia*** *9* ⟩≡

```
FreeAndNil(driverMagazzinoMacro);
FreeAndNil(driverScriviDocumentazione);
FreeAndNil(streamOutputDocumentazione);
FreeAndNil(stream);
FreeAndNil(store);
```

Usata da: 1

Capitolo 3. The macro store

Macro are memorized in the macro store.

⟨ ***TMacroRecord*** *10* ⟩≡

```
EMacroType = ( FileMacro, ScrapMacro );

RScrapLine = record
  Content:String;
  FileName:String;
  LineNumber:Integer;
end;

TMacroRecord = class
private
  FMacroName:String;
  FMacroContent:array of RScrapLine;
  FMacroLinesCount:Integer;
  FMacroProgr:Integer;
  FMacroType:EMacroType;

  procedure AddLine (Content:String; FileCorrente:String; LineaCorrente:Integer);
  function ReadMacroContent:String;
  function GetMacroLine(Idx:Integer):RScrapLine;
public
  macroUsers: array of Integer;
  macroUsersCount: Integer;

  constructor CreateWithData(Name:String; Progressivo:Integer;
    Tipo:EMacroType);

  property MacroName:String read FMacroName;
  property MacroContent:String read ReadMacroContent;
  property MacroProgr:Integer read FMacroProgr;
  property MacroType:EMacroType read FMacroType;
  procedure AddContent (Content:String; FileCorrente:String; LineaCorrente:Integer);
  property MacroLinesCount:integer read FMacroLinesCount;
  property MacroLine[idx:Integer]:RScrapLine read GetMacroLine;
end;
```

Usata da: 27

For every macro the following attribute are memorized:

- the name;
- a progressive number;
- the content;
- the type (this permits to remember if it's a file generating macro or not);
- the macros where this macro is used (`macroUsers` vector and `macroUsersCount`)

Every macro is populated with scraps and every scrap is created by a set of lines. Slit must memorize, for every row in a scrap, the name of the input file and the number of the row. In this way we can build, afterwords, a map between the row in the documentation file and the row in the source code.

This map is really useful for developer to demangle the error messages given by compilers.

Macro are always created with a name:

⟨ ***TMacroRecord.CreateWithData*** *11* ⟩≡

```
constructor TMacroRecord.CreateWithData(Name:String;
  Progressivo:Integer; Tipo:EMacroType);
begin
  FMacroName := Name;
  FMacroProgr := Progressivo;
  FMacroType := Tipo;
  FMacroLinesCount := 0;
  SetLength(FMacroContent, 50);
end;
```

Usata da: 27

You can add rows to a macro, and this is managed by hold the current filename and the row number. The row number passed to this procedure is that of the first row of the content passed.

⟨ ***TMacroRecord.AddContent*** *12* ⟩≡

```
procedure TMacroRecord.AddContent (Content:String; FileCorrente:String;
  LineaCorrente:Integer);
var
  divisioneRighe:TStringList;
  i:Integer;
begin
  divisioneRighe := TStringList.Create;
  divisioneRighe.Text := Content;

  for i:=0 to divisioneRighe.Count-1 do
  begin
    AddLine (divisioneRighe.Strings[i], FileCorrente, LineaCorrente+i);
  end;

  FreeAndNil (divisioneRighe);
  end;
```
Usata da: 27

The following function instead add the content line by line:

⟨ ***TMacroRecord.AddLine*** *13* ⟩≡
```
  procedure TMacroRecord.AddLine (Content:String; FileCorrente:String;
    LineaCorrente:Integer);
  begin
    if Length(FMacroContent)>=FMacroLinesCount then
    begin
      SetLength(FMacroContent, Length(FMacroContent)+50);
    end;

    FMacroContent[FMacroLinesCount].Content := Content;
    FMacroContent[FMacroLinesCount].FileName := FileCorrente;
    FMacroContent[FMacroLinesCount].LineNumber := LineaCorrente;
    FMacroLinesCount := FMacroLinesCount+1;
  end;
```
Usata da: 27

This function gives the content of the macro:

⟨ ***TMacroRecord.ReadMacroContent*** *14* ⟩≡
```
  function TMacroRecord.ReadMacroContent:String;
  var
    i:Integer;

  begin
    Result:='';
    for i:=0 to FMacroLinesCount-1 do
    begin
      Result:=Result+FMacroContent[i].Content+LineEnding;
    end;
  end;
```
Usata da: 27

⟨ ***TMacroRecord.GetMacroLine*** *15* ⟩≡
```
  function TMacroRecord.GetMacroLine(idx:Integer):RScrapLine;
  begin
    Result := FMacroContent[idx];
  end;
```
Usata da: 27

Macros are memorized in a dynamic vector whose size is initially 50.

⟨ ***TMacroStore.Create*** *16* ⟩≡
```
  constructor TMacroStore.Create;
```

```
var
  i:Integer;
begin
  count:=0;
  SetLength(store, 50);

  for i:=0 to Length(Store)-1 do
  begin
    Store[i] := Nil;
  end;
end;
```

Usata da: 27

Per memorizzare una macro viene controllato lo spazio disponibile nel vettore (che è memorizzato nella variabile count). Se c'è spazio a sufficienza allora la macro viene memorizzata altrimenti prima di essere memorizzata il vettore viene ampliato per far posto ad altre 50 macro.

Una macro non può essere ripetuta all'interno dello stesso file. Per questo motivo, prima di memorizzare la macro, viene controllata l'esistenza di una macro con lo stesso nome e, caso mai, viene segnalato un errore all'utente.

⟨ ***TMacroStore.StoreMacro*** *17* ⟩≡

```
procedure TMacroStore.StoreMacro(macroName:String; macroContent:String;
  macroType:EMacroType; FileName:String; CurrentLine:Integer);
var
  i:Integer;
begin
  if GetMacro(macroName) <> Nil then
  begin
    writeln('Attenzione: macro ', macroName, ' duplicata.');
  end
  else
  begin
    if count>=Length(store) then
    begin
      SetLength(store, Length(Store)+50);
      for i:=count to Length(Store)-1 do
      begin
        Store[i] := Nil;
      end;
    end;
    store[count] := TMacroRecord.CreateWithData (macroName, count+1, macroType);
    store[count].AddContent (macroContent, FileName, CurrentLine);
    count := count + 1;
  end;
end;
```

Usata da: 27

To find a macro by name every element in the vector is read:

⟨ ***TMacroStore.GetMacro*** *18* ⟩≡

```
function TMacroStore.GetMacro(macroName:String):TMacroRecord;
var
  i:integer;
begin
  Result := Nil;
  for i:=0 to length(Store)-1 do
  begin
    if (store[i]<>Nil) and (store[i].MacroName=macroName) then
    begin
      Result:=store[i];
      exit;
    end;
  end;
end;
```

Usata da: 27

This procedures gives the count of the macro in the store:

⟨ ***TMacroStore.MacroCount*** *19* ⟩≡

```
function TMacroStore.MacroCount:Integer;
begin
  Result := count;
end;
```

Usata da: 27

This procedure instead find a macro given it's progressive number:

⟨ ***TMacroStore.GetRecord*** *20* ⟩≡

```
function TMacroStore.GetRecord(i:integer):TMacroRecord;
begin
  Result := store[i];
end;
```

Usata da: 27

3.1. Cross-reference

The store computes also the cross-reference database. For example if the macro *one* includes the macro *two*, in the record of the macro *two* is inserted the ID of the macro *one*. The cross-reference database if computed by the procedure *CalcolaRiferimenti*.

⟨ ***TMacroStore.CalcolaRiferimenti*** *21* ⟩≡

```
procedure TMacroStore.CalcolaRiferimenti;
var
  i, j, k: Integer;
  listaStringhe: TStringList;
  stringaPulita: String;
  tempRecord: TMacroRecord;
begin
  listaStringhe := TStringList.Create;

  <TMacroStore.CalcolaRiferimenti pulizia record 22>
  <TMacroStore.CalcolaRiferimenti calcolo 23>

  FreeAndNil( listaStringhe );
end;
```

Usata da: 27

The cross-reference database is initially cleared:

⟨ ***TMacroStore.CalcolaRiferimenti pulizia record*** *22* ⟩≡

```
for i:=0 to count-1 do
begin
  store[i].macroUsersCount:=0;
  SetLength(store[i].macroUsers, 10);
end;
```

Usata da: 21

Now all the rows are read:

⟨ ***TMacroStore.CalcolaRiferimenti calcolo*** *23* ⟩≡

```
for i:=0 to count-1 do
begin
  listaStringhe.Text := store[i].macroContent;

  for j := 0 to listaStringhe.Count-1 do
  begin
    <TMacroStore.CalcolaRiferimenti processa riga 24>
  end;
end;
```

Usata da: 21

If the row is a reference the ID of the corresponding macro is retrieved and inserted in the reference database.

⟨ ***TMacroStore.CalcolaRiferimenti processa riga*** *24* ⟩≡

```
stringaPulita := Trim( listaStringhe.Strings[j] );
if AnsiStartsStr('@<', stringaPulita) and
   AnsiEndsStr('@>', stringaPulita) then
begin
  stringaPulita := MidStr(stringaPulita, 3, Length(stringaPulita)-4);
  tempRecord := GetMacro( stringaPulita );
  if (tempRecord<>Nil) and (tempRecord.macroProgr<>0) then
  begin
```

```
    <TMacroStore.CalcolaRiferimenti inserisci riferimento 25>
  end;
end;
```

Usata da: 23

To insert a now reference the dynamic vector must be checked and resized if necessary:

⟨ ***TMacroStore.CalcolaRiferimenti inserisci riferimento*** *25* ⟩≡

```
k := tempRecord.macroProgr-1;

if store[k].macroUsersCount = Length(store[k].macroUsers) then
begin
  SetLength(store[k].macroUsers, Length(store[k].macroUsers)+10);
end;
store[k].macroUsers[store[k].macroUsersCount] := i + 1;
store[k].macroUsersCount := store[k].macroUsersCount + 1;
```

Usata da: 24

3.2. Definition of the macrostore unit

To sum up, the definition of the `TMacroStore` class is the following:

⟨ ***TMacroStore*** *26* ⟩≡

```
TMacroStore = class
private
  count:integer;
  store:array of TMacroRecord;
public
  constructor Create;
  function MacroCount:Integer;
  procedure StoreMacro(macroName:String; macroContent:String;
    macroType:EMacroType; FileName:String; CurrentLine:Integer);
  function GetMacro(macroName:String):TMacroRecord;
  function GetRecord(i:integer):TMacroRecord;
  procedure CalcolaRiferimenti;
end;
```

Usata da: 27

And this is the definition of the `macrostore` unit:

⟨ ***FILE macrostore.pas*** *27* ⟩≡

```
{$MODE OBJFPC}
{$H+}
unit macrostore;

interface
```

```
type
  <TMacroRecord 10>
  <TMacroStore 26>

implementation
  uses SysUtils, Classes, StrUtils, slitstatus;

  <TMacroStore.Create 16>
  <TMacroStore.StoreMacro 17>
  <TMacroStore.GetMacro 18>
  <TMacroStore.MacroCount 19>
  <TMacroStore.GetRecord 20>
  <TMacroStore.CalcolaRiferimenti 21>

  <TMacroRecord.ReadMacroContent 14>
  <TMacroRecord.CreateWithData 11>
  <TMacroRecord.AddContent 12>
  <TMacroRecord.AddLine 13>
  <TMacroRecord.GetMacroLine 15>
end.
```

Capitolo 4. The translation status

Slit has a unit to manage the translation process and the translation status.

The informations in this unit and the data structured are documented in this chapter.

4.1. Options management

The Slit parameters system makes the translation process configurable and adaptable to the user preferences.

This parameters set, called *options,* is managed by a unit, who provides high-level operation managing that data.

4.1.1. The output format

Slit must write, in the documentation, the content of the macros. So he need to know the documentation format.

This parameters rembember the format used and this information is used to create the right documentation driver.

⟨ ***slitstatus, gestore del processore di documentazione*** *28* ⟩ ≡

```
function GetNomeProcessoreInformazioni():String;
begin
  Result := NomeProcessoreInformazioni;
end;

procedure SetNomeProcessoreInformazioni(value:String);
begin
  if value='html' then
  begin
    NomeProcessoreInformazioni := 'html';
  end
  else if value='lout' then
  begin
    NomeProcessoreInformazioni := 'lout';
  end
  else if value='txt' then
  begin
    NomeProcessoreInformazioni := 'txt';
  end
  else
  begin
    raise Exception.Create('Nome processore informazioni non conosciuto: ' +
      value);
  end;
end;
```

Usata da: 42

Now that we know the documentation format we can write a function to generated the right instance of the documentation driver.

⟨ ***slitstatus, crea lo stream di output*** *29* ⟩≡

```
function CreaStreamOutputDaOpzioni(NomeFile:String; store:TMacroStore):TSlitOutput;
begin
  if NomeProcessoreInformazioni='lout' then
  begin
    Result := TSlitOutputLout.CreateForFileAndStore (NomeFile, store);
  end
  else if NomeProcessoreInformazioni='txt' then
  begin
    Result := TSlitOutputTxt.CreateForFile (NomeFile);
  end
  else if NomeProcessoreInformazioni='html' then
  begin
    Result := TSlitOutputHtml.CreateForFile (NomeFile);
  end
  else
  begin
    Result := Nil;
  end;
end;
```

Usata da: 42

4.1.2. Comment start and end markers

In the source code generated Slit can insert a reference to the original documentation file.

This information is enclosed in a comment block that must be valid for the generated language. A software can be composed of source code written in different languages so we must distuish between the source code languages to choose the right comment marker. We do this using the extension of the generated source code file.

At every extension corresponds the comment start and end markers. Slit has a database with predefines associations for the popular programming languages.

Slit insert comments only at the end of line so works also for languages that doesn't admit comment between the source code lines.

Slit is ready for the following programming languages:

- Pascal (.pas, .pp),
- C, C++ (.c, .cpp, .h),
- D (.d),
- Java (.java),
- C# (.cs),
- Ada (.ada, .adb, .ads),
- Haskell (.hs),
- Python (.py),
- Lout (.lout),
- Ruby (.rb),
- Assembler (.s, .asm),
- Basic (.bas),
- NSIS (.NSI),
- Zinc (.zc).

These languages are configured directly in the source code of slit:

⟨ ***slitstatus, configurazione delle estensioni predefinite****30* ⟩ ≡

```
AggiungiLinguaggio( ’.pas’, ’{’, ’}’ );
AggiungiLinguaggio( ’.pp’, ’{’, ’}’ );
AggiungiLinguaggio( ’.c’, ’/*’, ’*/’ );
AggiungiLinguaggio( ’.d’, ’/*’, ’*/’ );
AggiungiLinguaggio( ’.cpp’, ’/*’, ’*/’ );
AggiungiLinguaggio( ’.java’, ’/*’, ’*/’ );
AggiungiLinguaggio( ’.cs’, ’/*’, ’*/’ );
AggiungiLinguaggio( ’.ada’, ’--’, ’’ );
AggiungiLinguaggio( ’.adb’, ’--’, ’’ );
AggiungiLinguaggio( ’.ads’, ’--’, ’’ );
AggiungiLinguaggio( ’.hs’, ’--’, ’’ );
AggiungiLinguaggio( ’.py’, ’#’, ’’ );
AggiungiLinguaggio( ’.rb’, ’#’, ’’ );
AggiungiLinguaggio( ’.lout’, ’#’, ’’ );
AggiungiLinguaggio( ’.s’, ’; ’, ’’ );
AggiungiLinguaggio( ’.asm’, ’; ’, ’’ );
AggiungiLinguaggio( ’.bas’, ’’’’’’, ’’ );
AggiungiLinguaggio( ’.nsi’, ’; ’, ’’ );
AggiungiLinguaggio( ’.zc’, ’// ’, ’’ );
```

Usata da: 42

This informations are hold in records:

⟨ ***slitstatus, RInformazioniLinguaggi*** *31* ⟩≡

```
RInformazioniLinguaggi = record
  Estensione:String;
  Inizio:String;
  Fine:String;
end;
```

Usata da: 42

The records are held in a table inside the Slit global configuration status:

⟨ ***slitstatus, AggiungiLinguaggio*** *32* ⟩≡

```
procedure AggiungiLinguaggio (Estensione, Inizio, Fine:String);
begin
  if Length(TabellaLinguaggi)>=TabellaLinguaggi_Count then
  begin
    SetLength (TabellaLinguaggi, TabellaLinguaggi_Count+50);
  end;

  TabellaLinguaggi[TabellaLinguaggi_Count].Estensione := Estensione;
  TabellaLinguaggi[TabellaLinguaggi_Count].Inizio := Inizio;
  TabellaLinguaggi[TabellaLinguaggi_Count].Fine := Fine;
  TabellaLinguaggi_Count := TabellaLinguaggi_Count + 1;
end;
```

Usata da: 42

To use these informations slit use the following procedure:

⟨ ***slitstatus, PrendiMarcatori*** *33* ⟩≡

```
procedure PrendiMarcatori (NomeFile:String; var Inizio:String; var Fine:String);
var
  i:Integer;
  fatto:Boolean;

begin
  fatto := false;

  for i:=TabellaLinguaggi_Count-1 downto 0 do
  begin
    if AnsiEndsText (TabellaLinguaggi[i].Estensione, NomeFile) then
    begin
      Inizio := TabellaLinguaggi[i].Inizio;
      Fine := TabellaLinguaggi[i].Fine;
      Fatto := True;
    end;
  end;

  if not fatto then
  begin
    Inizio := ’’;
    Fine := ’’;
  end;
end;
```

Usata da: 42

Inside the table the instructions are searched in backward manner because the configuration made by the user must take precedence over the default configurations.

4.1.3. Generation of the section start and end marker

The generated source code can also contain the name of the Slit macro that has been read. This parameter enable or disable this functionality:

⟨ ***slitstatus, Get/Set GenerazioneMarcatoriAbilitata*** *34* ⟩≡

```
function GetGenerazioneMarcatoriAbilitata:Boolean;
begin
  Result := GenerazioneMarcatoriAbilitata;
end;

procedure SetGenerazioneMarcatoriAbilitata(value:Boolean);
begin
  GenerazioneMarcatoriAbilitata := value;
end;
```

Usata da: 42

4.1.4. Generation of line markers

Sometimes it’s important to know, in the generated source code, the name of the documentation file and the row number. This is useful to decode the error messages given by the compiler.

⟨ ***slitstatus, Get/Set GenerazioneNumeriRigaAbilitata*** *35* ⟩≡

```
function GetGenerazioneNumeriRigaAbilitata:Boolean;
begin
  Result := GenerazioneNumeriRigaAbilitata;
end;

procedure SetGenerazioneNumeriRigaAbilitata(value:Boolean);
begin
  GenerazioneNumeriRigaAbilitata := value;
```

```
end;
```

Usata da: 42

The markers, for aestetics reasons, are put at a precise column or at the end of the source code line. This column number is configurable:

⟨ ***slitstatus, Get/Set ColonnaNumeriRiga*** *36* ⟩≡

```
function GetColonnaNumeriRiga:Integer;
begin
  Result := ColonnaNumeriRiga;
end;

procedure SetColonnaNumeriRiga(value:Integer);
begin
  ColonnaNumeriRiga := value;
end;
```

Usata da: 42

4.2. Current file name management

The current filename must be hold to make error messages as precise as possible. The parser, after opening a file and before closing it, signal an event to this module. These events are received and used to held the current filename.

Streams are held in a dynamic stack:

⟨ ***slitstatus, SegnalaInizioElaborazioneStream*** *37* ⟩≡

```
procedure SegnalaInizioElaborazioneStream (stream:TSlitStream);
begin
  if Length(StreamStack)>=StreamStackCount then
  begin
    SetLength(StreamStack, Length(StreamStack)+10);
  end;

  StreamStack[StreamStackCount] := stream;
  StreamStackCount := StreamStackCount+1;
end;
```

Usata da: 42

⟨ ***slitstatus, SegnalaFineElaborazioneStream*** *38* ⟩≡

```
procedure SegnalaFineElaborazioneStream;
begin
  if StreamStackCount>0 then
  begin
    StreamStack[StreamStackCount] := Nil;
    StreamStackCount := StreamStackCount-1;
    end;
  end;
```

Usata da: 42

In this way we can write functions to get the current filename and the current row:

⟨ ***slitstatus, GetCurrentParsingFile*** *39* ⟩≡

```
  function GetCurrentParsingFile:String;
  begin
    if StreamStackCount>0 then
    begin
      Result := StreamStack[StreamStackCount-1].CurrentFile;
    end
    else
    begin
      Result := ’’;
    end;
  end;
```

Usata da: 42

Se non c’è la linea corrente allora la funzione ritorna -1:

⟨ ***slitstatus, GetCurrentParsingLine*** *40* ⟩≡

```
  function GetCurrentParsingLine:Integer;
  begin
    if StreamStackCount>0 then
    begin
      Result := StreamStack[StreamStackCount-1].CurrentLine;
    end
    else
    begin
      Result := -1;
    end;
  end;
```

Usata da: 42

Thanks to these procedures we can create a procedure to emit error messages referring to che current processed file:

⟨ ***slitstatus, LogErrorMessage*** *41* ⟩≡

```
  procedure LogErrorMessage(message:String);
  begin
    if StreamStackCount>0 then
    begin
      write (StdErr, StreamStack[StreamStackCount-1].CurrentFile);
      write (StdErr, ’:’);
      write (StdErr, StreamStack[StreamStackCount-1].CurrentLine);
      write (StdErr, ’ ’);
    end;

    writeln (StdErr, message);
  end;
```
Usata da: 42

4.3. slitstatus unit definition

⟨ ***FILE slitstatus.pas*** *42* ⟩≡

```
  {$MODE OBJFPC}
  {$H+}
  unit slitstatus;

  interface

  uses slitoutput, macrostore, slitstream;

  function GetNomeProcessoreInformazioni():String;
  procedure SetNomeProcessoreInformazioni(value:String);
  function CreaStreamOutputDaOpzioni(NomeFile:String; store:TMacroStore):TSlitOutput;
  procedure SegnalaInizioElaborazioneStream (stream:TSlitStream);
  procedure SegnalaFineElaborazioneStream;
  procedure LogErrorMessage(message:String);
  function GetCurrentParsingFile:String;
  function GetCurrentParsingLine:Integer;
  function GetGenerazioneMarcatoriAbilitata:Boolean;
  procedure SetGenerazioneMarcatoriAbilitata(value:Boolean);
  function GetGenerazioneNumeriRigaAbilitata:Boolean;
  procedure SetGenerazioneNumeriRigaAbilitata(value:Boolean);
  function GetColonnaNumeriRiga:Integer;
  procedure SetColonnaNumeriRiga(value:Integer);
  procedure AggiungiLinguaggio (Estensione, Inizio, Fine:String);
  procedure PrendiMarcatori (NomeFile:String; var Inizio:String; var Fine:String);

  implementation

  uses sysutils, slithtml, slitlout, slittxt, strutils;

  type
    <slitstatus, RInformazioniLinguaggi 31>

  var
    NomeProcessoreInformazioni:String;
    GenerazioneMarcatoriAbilitata:Boolean;
    GenerazioneNumeriRigaAbilitata:Boolean;
    StreamStack: array of TSlitStream;
    StreamStackCount: Integer;
    ColonnaNumeriRiga:Integer;
    TabellaLinguaggi:array of RInformazioniLinguaggi;
    TabellaLinguaggi_Count:Integer;

  <slitstatus, gestore del processore di documentazione 28>
  <slitstatus, crea lo stream di output 29>
  <slitstatus, SegnalaInizioElaborazioneStream 37>
  <slitstatus, SegnalaFineElaborazioneStream 38>
  <slitstatus, LogErrorMessage 41>
  <slitstatus, GetCurrentParsingFile 39>
  <slitstatus, GetCurrentParsingLine 40>
  <slitstatus, Get/Set GenerazioneMarcatoriAbilitata 34>
  <slitstatus, Get/Set GenerazioneNumeriRigaAbilitata 35>
  <slitstatus, Get/Set ColonnaNumeriRiga 36>
  <slitstatus, AggiungiLinguaggio 32>
  <slitstatus, PrendiMarcatori 33>

  initialization

    NomeProcessoreInformazioni := ’lout’;
    StreamStackCount := 0;
    GenerazioneMarcatoriAbilitata := True;
    GenerazioneNumeriRigaAbilitata := True;
    ColonnaNumeriRiga := 100;
    TabellaLinguaggi_Count := 0;
    SetLength (TabellaLinguaggi, 0);

    <slitstatus, configurazione delle estensioni predefinite 30>

  end.
```

Capitolo 5. Parser

Slit read text files made be text rows and directives. Directives are rows starting with the prefix @ and optionally followed by *scraps*.

5.1. Drivers

The Slit file parter reads the source files and uses a driver to process directives. In this war, we can use the parser to drive different phases of the translation process.

Drivers share this structure:

⟨ ***slitstream definizione TSlitStreamDriver*** *43* ⟩ ≡

```
TSlitStream = class;

TSlitStreamDriver = class
private
  FParser:TSlitStream;

public
  procedure ProcessaDefinizioneMacro(nomeMacro:String; scrap:String;
    scrapStartLine:Integer); virtual; abstract;
  procedure ProcessaAggiungiNellaMacro(nomeMacro:String; scrap:String;
    scrapStartLine:Integer); virtual; abstract;
  procedure ProcessaDefinizioneFile(nomeMacro:String; scrap:String;
    scrapStartLine:Integer); virtual; abstract;
  procedure ProcessaRigaDocumentazione(riga:String);
    virtual; abstract;
  procedure ProcessaOpzione(opzione:String);
    virtual; abstract;

  property Parser:TSlitStream read FParser write FParser;
end;
```

Usata da: 55

5.2. Scrap reading

A scrap represent a part of the body of a macro and starts with the row `@{` and ends with the row `@}` .

A scrap can be read with this code:

⟨ ***TSlitStream.ReadScrap*** *44* ⟩≡

```
function TSlitStream.ReadScrap():String;
var
  buffer:String;
  bufferLine:String;
begin
  bufferLine := NextLine;
  if Trim(bufferLine)<>'@{' then
  begin
    LogErrorMessage('Mi aspettavo l''inizio di una macro');
  end;

  buffer := '';
  while (not EOF) do
  begin
    bufferLine := NextLine();
    if Trim(bufferLine)='@}' then
    begin
      break;
    end;
    buffer := buffer + bufferLine + Chr(13) + Chr(10);
  end;
  Result := buffer;
end;
```

Usata da: 55

5.3. Directives

The macro definition directive `@d` permit to create a new macro. The directive is followed by the name of the macro and the content of the macro is the content of the following scrap.

The `@d` is readden with this code:

⟨ ***processa direttiva d*** *45* ⟩≡

```
scrapStartLine := CurrentLine+2;
scrapBuffer := ReadScrap();
macroName := Trim(MidStr(lineBuffer, 3, Length(lineBuffer)-2));

if FDriver <> Nil then
begin
  FDriver.ProcessaDefinizioneMacro(macroName, scrapBuffer, scrapStartLine);
end;
```

Usata da: 50

Inside every macro definition you can call another macro with the sintax `@<nomemacro@>`.

The directive `@o` è is like the definition one but is used to write a file whose name is the name of the macro.

The filename can be enclosed by quotation marks `""`. When this is true the quotation marks must be removed from the filename.

⟨ ***processa direttiva o*** *46* ⟩ ≡

```
scrapStartLine := CurrentLine+2;
scrapBuffer := ReadScrap();
macroName := Trim(MidStr(lineBuffer, 3, Length(lineBuffer)-2));

if AnsiStartsStr(’"’, macroName) and AnsiEndsStr(’"’, macroName) then
begin
  macroName := MidStr(macroName, 2, Length(macroName)-2);
end;

if FDriver <> Nil then
begin
  FDriver.ProcessaDefinizioneFile(macroName, scrapBuffer, scrapStartLine);
end;
```

Usata da: 50

The directive `@+` è is somewhat similiar: is’t used to add content to an existing macro. To process an *add* directive we can use a process similiar to the processing of the definition directive: from the parser point of view the only difference is that he must call a different function of the driver.

⟨ ***processa direttiva +*** *47* ⟩ ≡

```
scrapStartLine := CurrentLine+1;
scrapBuffer := ReadScrap();
macroName := Trim(MidStr(lineBuffer, 3, Length(lineBuffer)-2));

if FDriver <> Nil then
begin
  FDriver.ProcessaAggiungiNellaMacro(macroName, scrapBuffer, scrapStartLine);
end;
```

Usata da: 50

The directive `@i` is used to include a file in the main file. This code will make the parser read another file using the same macro store and the same output:

⟨ ***processa direttiva i*** *48* ⟩ ≡

```
macroName := Trim(MidStr(lineBuffer, 3, Length(lineBuffer)-2));
temporaryStream := TSlitStream.CreateForFile(
  ExtractFilePath(FNomeFile) + macroName);
temporaryStream.Driver := FDriver;
temporaryStream.Process();
FreeAndNil(temporaryStream);
```

Usata da: 50

The filename is interpresed as relative to the current file.

The directive `@#` is the comment directive. All the content following this directive is ignored.

The directive `@x` is used to configure a parameter and is managed by the current driver:

⟨ ***processa direttiva x*** *49* ⟩≡

```
FDriver.ProcessaOpzione (MidStr(lineBuffer,3,Length(lineBuffer)-2));
```

Usata da: 50

If the read row is not a directive then is interpreted as a documentation line:

⟨ ***TSlitStream.Process*** *50* ⟩≡

```
procedure TSlitStream.Process();
var
  lineBuffer:String;
  scrapBuffer:String;
  scrapStartLine:Integer;
  macroName:String;
  temporaryStream:TSlitStream;
begin
  SegnalaInizioElaborazioneStream(Self);
  FDriver.Parser := Self;

  while (not Eof) do
  begin
    lineBuffer := NextLine();

    if AnsiStartsStr('@d ',lineBuffer) then
    begin
      <processa direttiva d 45>
    end
    else if AnsiStartsStr('@o ', lineBuffer) then
    begin
      <processa direttiva o 46>
    end
    else if AnsiStartsStr('@i ', lineBuffer) then
    begin
      <processa direttiva i 48>
    end
    else if AnsiStartsStr('@# ', lineBuffer) then
    begin
      { no-op, si tratta di un commento }
    end
    else if AnsiStartsStr('@x ', lineBuffer) then
    begin
     <processa direttiva x 49>
    end
    else if AnsiStartsStr('@+ ', lineBuffer) then
    begin
     <processa direttiva + 47>
    end
    else
    begin
      if FDriver <> Nil then
      begin
        FDriver.ProcessaRigaDocumentazione(lineBuffer);
      end;
    end;
  end;

  SegnalaFineElaborazioneStream;
end;
```

Usata da: 55

5.4. Streams

Every file is opened then the stream get created:

⟨ ***TSlitStream.CreateForFile*** *51* ⟩≡

```
constructor TSlitStream.CreateForFile(fileName:String);
begin
  if not FileExists(fileName) then
  begin
    writeln('Il file ', fileName, ' non esiste');
    Abort;
  end;

  FCurrentLine := 0;
  FNomeFile := fileName;
  FDriver := Nil;
  Assign(handle, fileName);
  Reset(handle);
end;
```

Usata da: 55

and closed when the stream is destroyed:

⟨ ***TSlitStream.Destroy*** *52* ⟩≡

```
destructor TSlitStream.Destroy;
```

```
begin
  Close(Handle);
  inherited Destroy;
end;
```

Usata da: 55

⟨ ***slitstream definizione TSlitStream*** *53* ⟩≡

```
TSlitStream = class
private
  FCurrentLine:integer;
  FNomeFile:String;
  FDriver:TSlitStreamDriver;
  handle:Text;

  function IsEof:Boolean;
public
  constructor CreateForFile(fileName:String);
  destructor Destroy; override;
  function NextLine:String;
  function ReadScrap():String;
  procedure Process();
  procedure ResetStream();

  property EOF:Boolean read IsEof;
  property Driver:TSlitStreamDriver read FDriver write FDriver;
  property CurrentFile:String read FNomeFile;
  property CurrentLine:Integer read FCurrentLine;
end;
```

Usata da: 55

The other operations call only the primitives of the stream:

⟨ ***TSlitStream altre*** *54* ⟩≡

```
function TSlitStream.NextLine:String;
var
  bufLine:String;
begin
  readln(Handle, bufLine);
  FCurrentLine := currentLine + 1;
  Result := bufLine;
end;

function TSlitStream.IsEof:Boolean;
begin
  Result := system.EOF(handle);
end;
```

```
procedure TSlitStream.ResetStream();
begin
  Reset(handle);
end;
```

Usata da: 55

5.5. slitstream definition

⟨ ***FILE slitstream.pas*** *55* ⟩≡

```
{$MODE OBJFPC}
{$H+}
unit slitstream;

interface
  uses macrostore, slitoutput;

type
  <slitstream definizione TSlitStreamDriver 43>
  <slitstream definizione TSlitStream 53>

implementation
  uses sysutils, strutils, slitstatus;

  <TSlitStream.CreateForFile 51>
  <TSlitStream.Destroy 52>
  <TSlitStream.ReadScrap 44>
  <TSlitStream.Process 50>
  <TSlitStream altre 54>
end.
```

Capitolo 6. Macro store and parameters

With this driver the macro store get populated.

This drives is called with a macro store.

⟨ ***TSlitStreamDriverMagazzino.CreateWithMacroStore*** *56* ⟩≡

```
constructor TSlitStreamDriverMagazzino.CreateWithMacroStore(ms:TMacroStore);
begin
  FMacroStore := ms;
  FTipoOutput := ’’;
end;
```

Usata da: 63

6.1. Adding a new macro

When a macro get read it’s checked and, if a macro with the same name already exists, a warning is emitted and the new macro is not considerated.

If it’s all ok the macro is added to the macro store:

⟨ ***TSlitStreamDriverMagazzino.ProcessaDefinizioneMacro*** *57* ⟩≡

```
procedure TSlitStreamDriverMagazzino.ProcessaDefinizioneMacro(
  nomeMacro:String; scrap:String; scrapStartLine:Integer);
var
  tempMacro : TMacroRecord;
begin
  tempMacro := FMacroStore.GetMacro(nomeMacro);
  if tempMacro<>Nil then
  begin
    LogErrorMessage(’Macro ’ + nomeMacro + ’ definita piu’’ volte’);
  end
  else
  begin
    FMacroStore.StoreMacro(nomeMacro, scrap, ScrapMacro, Parser.CurrentFile,
      scrapStartLine);
  end;
end;
```

Usata da: 63

The same happens for file definition macros:

⟨ ***TSlitStreamDriverMagazzino.ProcessaDefinizioneFile*** *58* ⟩≡

```
procedure TSlitStreamDriverMagazzino.ProcessaDefinizioneFile(
  nomeMacro:String; scrap:String; scrapStartLine:Integer);
var
  tempMacro : TMacroRecord;
begin
  tempMacro := FMacroStore.GetMacro(nomeMacro);
  if tempMacro<>Nil then
  begin
    LogErrorMessage(’Macro ’ + nomeMacro + ’ definita piu’’ volte’);
  end
  else
  begin
    FMacroStore.StoreMacro(nomeMacro , scrap, FileMacro,
      Parser.CurrentFile, scrapStartLine);
  end;
end;
```

Usata da: 63

When a scrap is added to an already existing macro we must do a different sequence of checks: in this case the macro must already exists. If the macro is already existing, the content of the scrap get added.

⟨ ***TSlitStreamDriverMagazzino.ProcessaAggiungiNellaMacro*** *59* ⟩≡

```
procedure TSlitStreamDriverMagazzino.ProcessaAggiungiNellaMacro(
  nomeMacro:String; scrap:String; scrapStartLine:Integer);
var
  tempMacro : TMacroRecord;
begin
  tempMacro := FMacroStore.GetMacro(nomeMacro);
  if tempMacro<>Nil then
  begin
    tempMacro.AddContent (scrap, Parser.CurrentFile, scrapStartLine);
  end
  else
  begin
    LogErrorMessage(’Richiesta aggiunta di uno scrap alla macro ’ + nomeMacro +
      ’ che non e’’ stata ancora definita’);
  end;
end;
```

Usata da: 63

The rows of documentation, in this implementation, are discarded:

⟨ ***TSlitStreamDriverMagazzino.ProcessaRigaDocumentazione*** *60* ⟩≡

```
procedure TSlitStreamDriverMagazzino.ProcessaRigaDocumentazione(riga:String);
begin
  { no op }
  end;
```
Usata da: 63

6.2. Parameters

The parameters are treated here and managed calling the functions of the Slit global state:

⟨ ***TSlitStreamDriverMagazzino.ProcessaOpzione*** *61* ⟩≡

```
  procedure TSlitStreamDriverMagazzino.ProcessaOpzione(opzione:String);
  begin
    opzione := Trim(Opzione);
    if opzione='output_lout' then
    begin
      SetNomeProcessoreInformazioni ('lout');
    end
    else if opzione='output_html' then
    begin
      SetNomeProcessoreInformazioni ('html');
    end
    else if opzione='output_txt' then
    begin
      SetNomeProcessoreInformazioni ('txt');
    end
    else if opzione='section_markers' then
    begin
      SetGenerazioneMarcatoriAbilitata(true);
    end
    else if opzione='no_section_markers' then
    begin
      SetGenerazioneMarcatoriAbilitata(false);
    end
    else if opzione='line_markers' then
    begin
      SetGenerazioneNumeriRigaAbilitata(true);
    end
    else if opzione='no_line_markers' then
    begin
      SetGenerazioneNumeriRigaAbilitata(false);
    end
    else if AnsiStartsStr('comment_markers', opzione) then
    begin
      GestioneOpzioneCommenti (opzione);
    end
    else
    begin
      LogErrorMessage('Opzione non conosciuta: ' + opzione);
    end;
  end;
```

Usata da: 63

The parameters needed to add comment to the generated source code is a little different because the markers are to be extracted from the option parameter.

⟨ ***TSlitStreamDriverMagazzino.GestioneOpzioneCommenti*** *62* ⟩≡

```
  procedure TSlitStreamDriverMagazzino.GestioneOpzioneCommenti (opzione:String);
  var
    estensione, inizio, fine:String;
    delimitatori: TSysCharSet;

  begin
    opzione := MidStr (opzione, Length('comment_markers')+1,
      Length(opzione)-Length('comment_markers')-1);
    opzione := Trim (opzione);

    if Length(opzione)<5 then
    begin
      LogErrorMessage ('opzione comment_markers con valore non valido');
    end
    else
    begin
      delimitatori := [opzione[1]];

      estensione := ExtractDelimited (2, opzione, delimitatori);
      inizio := ExtractDelimited (3, opzione, delimitatori);
      fine := ExtractDelimited (4, opzione, delimitatori);

      AggiungiLinguaggio (estensione, inizio, fine);
    end;
  end;
```

Usata da: 63

6.3. Driver definition

The definition of the class `TSlitStreamDriverMagazzino` and of the relative unit is following:

⟨ ***FILE drivermagazzino.pas*** *63* ⟩≡

```
  {$MODE OBJFPC}
  {$H+}
```

```
unit drivermagazzino;

interface
  uses macrostore, slitstream;

type
  TSlitStreamDriverMagazzino = class(TSlitStreamDriver)
    FMacroStore: TMacroStore;
    FTipoOutput: String;
  public
    constructor CreateWithMacroStore(ms:TMacroStore);
    procedure ProcessaDefinizioneMacro(nomeMacro:String; scrap:String;
      scrapStartLine:Integer); override;
    procedure ProcessaAggiungiNellaMacro(nomeMacro:String; scrap:String;
      scrapStartLine:Integer); override;
    procedure ProcessaDefinizioneFile(nomeMacro:String; scrap:String;
      scrapStartLine:Integer); override;
    procedure ProcessaRigaDocumentazione(riga:String);
      override;
    procedure ProcessaOpzione(opzione:String);
      override;
    procedure GestioneOpzioneCommenti (opzione:String);
  end;

implementation
  uses strutils, sysutils, slitstatus;

  <TSlitStreamDriverMagazzino.CreateWithMacroStore 56>
  <TSlitStreamDriverMagazzino.ProcessaDefinizioneMacro 57>
  <TSlitStreamDriverMagazzino.ProcessaDefinizioneFile 58>
  <TSlitStreamDriverMagazzino.ProcessaRigaDocumentazione 60>
  <TSlitStreamDriverMagazzino.ProcessaOpzione 61>
  <TSlitStreamDriverMagazzino.ProcessaAggiungiNellaMacro 59>
  <TSlitStreamDriverMagazzino.GestioneOpzioneCommenti 62>
end.
```

Capitolo 7. Documentation generator

This driver get bound to the parser to generate documentation using one of the predefined backend.

The driver is created with a macro store:

⟨ ***TSlitStreamDriverGenerazioneDoc.CreateWithOutputStream*** *64* ⟩≡

```
constructor TSlitStreamDriverGenerazioneDoc.CreateWithOutputStream(output:TSlitOutput);
  begin
    FOutputStream := output;
  end;
```

Usata da: 70

When a macro definition is received the macro definition must be written in the documentation using the backend:

⟨ ***TSlitStreamDriverGenerazioneDoc.ProcessaDefinizioneMacro*** *65* ⟩≡

```
  procedure TSlitStreamDriverGenerazioneDoc.ProcessaDefinizioneMacro(
    nomeMacro:String; scrap:String; scrapStartLine:Integer);
  begin
    FOutputStream.ScriviScrap(DefinitionScrap, nomeMacro, scrap);
  end;
```

Usata da: 70

The same happens whene a file definition is received:

⟨ ***TSlitStreamDriverGenerazioneDoc.ProcessaDefinizioneFile*** *66* ⟩≡

```
  procedure TSlitStreamDriverGenerazioneDoc.ProcessaDefinizioneFile(
    nomeMacro:String; scrap:String; scrapStartLine:Integer);
  begin
    FOutputStream.ScriviScrap(FileScrap, nomeMacro, scrap);
  end;
```

Usata da: 70

When a macro get added to the end of another:

⟨ ***TSlitStreamDriverGenerazioneDoc.ProcessaAggiungiNellaMacro*** *67* ⟩≡

```
  procedure TSlitStreamDriverGenerazioneDoc.ProcessaAggiungiNellaMacro(
    nomeMacro:String; scrap:String; scrapStartLine:Integer);
  begin
    FOutputStream.ScriviScrap(AppendScrap, nomeMacro, scrap);
  end;
```

Usata da: 70

The documentation lines are directly wrote:

⟨ ***TSlitStreamDriverGenerazioneDoc.ProcessaRigaDocumentazione*** *68* ⟩≡

```
procedure TSlitStreamDriverGenerazioneDoc.ProcessaRigaDocumentazione(
  riga:String);
begin
  FOutputStream.PutLine(riga);
end;
```

Usata da: 70

Parameters are simply ignored because are handled by the macro store driver:

⟨ ***TSlitStreamDriverGenerazioneDoc.ProcessaOpzione*** *69* ⟩≡

```
procedure TSlitStreamDriverGenerazioneDoc.ProcessaOpzione(opzione:String);
begin
  {* nop() *}
end;
```

Usata da: 70

The definition of the class `TSlitStreamDriverGenerazioneDoc` and of the containing file is the following:

⟨ ***FILE driverdoc.pas*** *70* ⟩≡

```
{$MODE OBJFPC}
{$H+}
unit driverdoc;

interface
  uses slitoutput, slitstream, macrostore;

type
  TSlitStreamDriverGenerazioneDoc = class(TSlitStreamDriver)
    FOutputStream: TSlitOutput;
  public
    constructor CreateWithOutputStream(output:TSlitOutput);
    procedure ProcessaDefinizioneMacro(nomeMacro:String; scrap:String;
      scrapStartLine:Integer); override;
    procedure ProcessaDefinizioneFile(nomeMacro:String; scrap:String;
      scrapStartLine:Integer); override;
    procedure ProcessaAggiungiNellaMacro(nomeMacro:String; scrap:String;
      scrapStartLine:Integer); override;
    procedure ProcessaRigaDocumentazione(riga:String);
      override;
    procedure ProcessaOpzione(opzione:String);
      override;
  end;
```

```
implementation
  <TSlitStreamDriverGenerazioneDoc.CreateWithOutputStream 64>
  <TSlitStreamDriverGenerazioneDoc.ProcessaDefinizioneMacro 65>
  <TSlitStreamDriverGenerazioneDoc.ProcessaDefinizioneFile 66>
  <TSlitStreamDriverGenerazioneDoc.ProcessaRigaDocumentazione 68>
  <TSlitStreamDriverGenerazioneDoc.ProcessaOpzione 69>
  <TSlitStreamDriverGenerazioneDoc.ProcessaAggiungiNellaMacro 67>

end.
```

Capitolo 8. Documentation generating backend

Slit can managed different documentation formats. The backend that manage the documentation output are designed in this way:

⟨ ***TSlitOutput*** *71* ⟩≡

```
EScrapType = (DefinitionScrap, AppendScrap, FileScrap);

TSlitOutput = class
public
  procedure ScriviScrap(tipo:EScrapType; nome, contenuto:String); virtual; abstract;
  procedure PutLine(str:String); virtual; abstract;
end;
```

Usata da: 72

With the method `ScriviScrap` you can write to the documentation a scrap of code.

⟨ ***FILE slitoutput.pas*** *72* ⟩≡

```

{$MODE OBJFPC}
{$H+}
unit slitoutput;

interface
  uses macrostore;

type
  <TSlitOutput 71>

implementation

end.
```

8.1. HTML Output

This backend write the output documentation in HTML format.

The created file has `.html` extension added to the documentation filename:

⟨ ***TSlitOutputHtml.CreateForFile*** *73* ⟩≡

```
  constructor TSlitOutputHtml.CreateForFile(fileName:String);
```

```
begin
  Assign(handle, ExtractFileName(fileName)+’.html’);
  Rewrite(handle);
end;
```

Usata da: 77

The file is closed when the output stream get destroyed:

⟨ ***TSlitOutputHtml.Destroy*** *74* ⟩≡

```
destructor TSlitOutputHtml.Destroy;
begin
  Close(handle);
end;
```

Usata da: 77

Documentation lines are added exactly as they are in the documentation file so you can add your tag to documentation:

⟨ ***TSlitOutputHtml.PutLine*** *75* ⟩≡

```
procedure TSlitOutputHtml.PutLine(str:String);
begin
  writeln(handle, str);
end;
```

Usata da: 77

Scraps are wrote as `div` elements and are inserted in preformatted blocks:

⟨ ***TSlitOutputHtml.ScriviScrap*** *76* ⟩≡

```
procedure TSlitOutputHtml.ScriviScrap(tipo:EScrapType; nome, contenuto:String);
var
  titolo: String;
begin
  if tipo = FileScrap then
  begin
    titolo := ’File’;
  end
  else if tipo = AppendScrap then
  begin
    titolo := ’Aggiunta alla definizione’;
  end
  else
  begin
    titolo := ’Definizione’;
  end;

  writeln(handle,
    ’<div class=’’testata’’>’, text2html(titolo), ’ ’, text2html(nome), ’</div>’);
  writeln(handle, '<pre>');
  writeln(handle, text2html(contenuto));
  writeln(handle, '</pre>');
end;
```

Usata da: 77

This is the definition of the HTML backend:

⟨ ***FILE slithtml.pas*** *77* ⟩≡

```
{$MODE OBJFPC}
{$H+}
unit slithtml;

interface
  uses slitoutput, macrostore;

type
  TSlitOutputHtml = class(TSlitOutput)
  private
    handle:Text;
  public
    constructor CreateForFile(fileName:String);
    destructor Destroy; override;

    procedure ScriviScrap(tipo:EScrapType; nome, contenuto:String); override;
    procedure PutLine(str:String); override;
  end;

implementation
  uses sysutils, strutils, classes, htmlutils;

  <TSlitOutputHtml.CreateForFile 73>
  <TSlitOutputHtml.Destroy 74>
  <TSlitOutputHtml.PutLine 75>
  <TSlitOutputHtml.ScriviScrap 76>

end.
```

8.2. Output in formato testo

Slit can also write text files. The format used is designed to use the `txt2tags` software.

⟨ ***TSlitOutputTxt.CreateForFile*** *78* ⟩≡

```
constructor TSlitOutputTxt.CreateForFile(fileName:String);
begin
```

```
    Assign(handle, ExtractFileName(fileName)+'.txt');
    Rewrite(handle);
  end;
```

Usata da: 82

When the backend is destroyed the stream get closed:

⟨ ***TSlitOutputTxt.Destroy*** *79* ⟩≡

```
  destructor TSlitOutputTxt.Destroy;
  begin
    Close(handle);
  end;
```

Usata da: 82

The documentation lines are added to the output file exactly as they are:

⟨ ***TSlitOutputTxt.PutLine*** *80* ⟩≡

```
  procedure TSlitOutputTxt.PutLine(str:String);
  begin
    writeln(handle, str);
  end;
```

Usata da: 82

A scrap is written in `txt2tags` format. The title of the scrap is written in bold and the content of the scrap is added to a text block.

⟨ ***TSlitOutputTxt.ScriviScrap*** *81* ⟩≡

```
  procedure TSlitOutputTxt.ScriviScrap(tipo:EScrapType; nome, contenuto:String);
  var
    titolo: String;
  begin
    if tipo = FileScrap then
    begin
      titolo := 'File';
    end
    else if tipo = AppendScrap then
    begin
      titolo := 'Aggiunta alla definizione';
    end
    else
    begin
      titolo := 'Definizione';
    end;

    writeln(handle, '------------------------------');
    writeln(handle, '| **', titolo, ' ', nome, '**');
    writeln(handle, '------------------------------');
    writeln(handle, ’“‘’);
    writeln(handle, contenuto);
    writeln(handle, ’“‘’);
    writeln(handle, ’------------------------------’);
  end;
```

Usata da: 82

This is the definition of the text backend:

⟨ ***FILE slittxt.pas*** *82* ⟩≡

```
  {$MODE OBJFPC}
  {$H+}
  unit slittxt;

  interface
    uses slitstream, slitoutput, macrostore;

  type

    TSlitOutputTxt = class(TSlitOutput)
    private
      handle:Text;
    public
      constructor CreateForFile(fileName:String);
      destructor Destroy; override;

      procedure ScriviScrap(tipo:EScrapType; nome, contenuto:String); override;
      procedure PutLine(str:String); override;
    end;

  implementation
    uses sysutils, strutils, classes;

    <TSlitOutputTxt.CreateForFile 78>
    <TSlitOutputTxt.Destroy 79>
    <TSlitOutputTxt.PutLine 80>
    <TSlitOutputTxt.ScriviScrap 81>

  end.
```

8.3. Lout output

Lout is a typesetting system as Tex but has a more simple sintax and is easy to program.

Lout is also really lightweight. Slit can create Lout documentation.

Lout files haven’t a predefined extension: slit assumes that this extension is `.lout`.

⟨ ***TSlitOutputLout.CreateForFileAndStore*** *83* ⟩≡

```
constructor TSlitOutputLout.CreateForFileAndStore(fileName:String; store:TMacroStore);
begin
  Assign(handle, ExtractFileName(fileName)+'.lout');
  Rewrite(handle);
  FStore := store;
end;
```

Usata da: 92

Stream get closed when the backend is destroyed:

⟨ ***TSlitOutputLout.Destroy*** *84* ⟩≡

```
destructor TSlitOutputLout.Destroy;
begin
  Close(handle);
end;
```

Usata da: 92

The documentation lines are added exactly as they are so you can add your Lout directives inside Slit files.

When slit writed the Lout file it write, as a comment, a reference to the source file so you can easily fix the errors given by Lout compiler.

To write comment lines Slit would identify if we are inside a **verbatim** block or not. To exactly identify this condition we should laxically analyse the output file. This approach is too heavyweight for a simple software like Slit so I have choosen to identify **verbatim** blocks by adding this comments only if the line starts with a section starting directive like `@Section`. The same check is done when you are starting a chapter or the introduction.

⟨ ***TSlitOutputLout.PutLine*** *85* ⟩≡

```
procedure TSlitOutputLout.PutLine(str:String);
var
  tempStr : String;

begin
  tempStr := Trim(str);
  if (AnsiStartsStr('@Section', tempStr)) or
     (AnsiStartsStr('@Chapter', tempStr)) or
     (AnsiStartsStr('@Introduction', tempStr)) then
  begin
    writeln (handle, '# ', GetCurrentParsingFile(), ':',
      GetCurrentParsingLine() );
  end;

  writeln(handle, str);
end;
```

Usata da: 92

The scrapt must be formatted in a particular way to make Lout handle them correctly.

This is a simple example of a code scrap formatted by Slit:

```
@LeftDisplay lines @Break {
@Sym angleleft @I @Verbatim @Begin definizione di ciao
 @End @Verbatim
 @Sym angleright @Sym equivalence
  @Verbatim @Begin while(true) { @End @Verbatim
   @Verbatim @Begin putstrln("ciao!"); @End @Verbatim
  @Verbatim @Begin } @End @Verbatim
}
```

The macro name is written in italic inside angular brackets.

The header get written like this:

⟨ ***TSlitOutputLout scrivi testata*** *86* ⟩≡

```
write(handle, '@Sym angleleft { BoldSlope } @Font @','Verbatim @Begin ');
if tipo=FileScrap then
begin
  write(handle, 'FILE ');
end;
write(handle, nome, ' @','End @','Verbatim ');

write(handle, '@I {', currentMacro.macroProgr, ' } ');

write(handle, ' @Sym angleright');
if tipo=AppendScrap then
begin
  write(handle, ' @Sym plus');
end;
writeln(handle, ' @Sym equivalence');
```

Usata da: 91

For every macro name a tag is generated so Lout can generate links for every macro definition.

Every created tag has a name generated using an auto-incremented integer.

⟨ ***TSlitOutputLout scrivi tag*** *87* ⟩≡

```
writeln(handle, '@PageMark { ', currentMacro.macroProgr, ' } ');
```

Usata da: 91

Il codice scritto dall'utente viene inserito fra blocchi verbatim ovvero fra "@Verbatim @Begin" e "@End @Verbatim".

Tutto il codice è racchiuso fra "@LeftDisplay lines @Break", che permette di rendere l'indentazione significativa.

Il codice viene prima diviso in linee e gli spazi che vengono prima del primo scritto vengono isolati dal codice perché sono significativi per la versione:

⟨ ***TSlitOutputLout scrivi codice*** *88* ⟩≡

```
stringhe := TStringList.Create;
stringhe.Text := contenuto;

for i := 0 to stringhe.Count-1 do
begin
  <TSlitOutputLout.ScriviScrap processa linea 89>
end;

FreeAndNil(stringhe);
```

Usata da: 91

If a line in a scraps represent a reference to a macro the link is written in bold face.

⟨ ***TSlitOutputLout.ScriviScrap processa linea*** *89* ⟩≡

```
spazi := Length( stringhe.Strings[i] );
spazi := spazi - Length(TrimLeft(stringhe.Strings[i]));
for j := 1 to spazi do
begin
  write(handle, ’ ’);
end;
write(handle, ’   ’);

stringaPulita := Trim(stringhe.Strings[i]);
if AnsiStartsStr(’@<’, stringaPulita) and
   AnsiEndsStr(’@>’, stringaPulita) then
begin
  nomeDefinizione := MidStr(stringaPulita, 3, Length(stringaPulita)-4);
  macroTemp := FStore.GetMacro( nomeDefinizione );
  if macroTemp<>Nil then
  begin
    write(handle, ’@I { ’, macroTemp.macroProgr, ’ } @CrossLink ’);
    stringaPulita := ’<’ + nomeDefinizione + ’ ’ + IntToStr(macroTemp.macroProgr) + ’>’;
  end;
end;

write(handle, ’@’,’Verbatim @’,’Begin ’);
write(handle, stringaPulita);
writeln(handle, ’@’,’End @’,’Verbatim’);
```

Usata da: 88

At the end of a scrap we write also the cross-reference:

⟨ ***TSlitOutputLout scrivi riferimenti*** *90* ⟩≡

```
if currentMacro.macroUsersCount <> 0 then
begin
```

```
  write(handle, ’{ -1p setsmallcaps 0.9 } @Font { ’);
  write(handle, ’Usata da: ’);
  for i:=0 to currentMacro.macroUsersCount-1 do
  begin
    write(handle, ’ { ’, currentMacro.macroUsers[i], ’ } @CrossLink { ’);
    write(handle, currentMacro.macroUsers[i], ’ } ’);
  end;
  write(handle, ’ } ’);
end;
```

Usata da: 91

In summary this is the code to write a scrap:

⟨ ***TSlitOutputLout.ScriviScrap*** *91* ⟩≡

```
procedure TSlitOutputLout.ScriviScrap(tipo:EScrapType; nome, contenuto:String);
var
  stringhe:TStringList;
  spazi:integer;
  stringaPulita, nomeDefinizione: String;
  i, j:integer;
  currentMacro, macroTemp: TMacroRecord;
begin
  currentMacro := FStore.GetMacro(nome);

  if currentMacro<>Nil then
  begin
    <TSlitOutputLout scrivi tag 87>

    writeln(handle, ’@LeftDisplay lines @Break {’);
    <TSlitOutputLout scrivi testata 86>
    <TSlitOutputLout scrivi codice 88>
    <TSlitOutputLout scrivi riferimenti 90>
    writeln(handle, ’}’);
  end;
end;
```

Usata da: 92

This is the lout backend definition:

⟨ ***FILE slitlout.pas*** *92* ⟩≡

```
{$MODE OBJFPC}
{$H+}
unit slitlout;

interface
  uses slitoutput, macrostore;

type
```

```
  TSlitOutputLout = class(TSlitOutput)
  private
    handle:Text;
    FStore: TMacroStore;
  public
    constructor CreateForFileAndStore(fileName:String; store:TMacroStore);
    destructor Destroy; override;
    procedure PutLine(str:String); override;
    procedure ScriviScrap(tipo:EScrapType; nome, contenuto:String); override;
  end;

implementation
  uses sysutils, strutils, classes, slitstatus;

  <TSlitOutputLout.CreateForFileAndStore 83>
  <TSlitOutputLout.Destroy 84>
  <TSlitOutputLout.PutLine 85>
  <TSlitOutputLout.ScriviScrap 91>

end.
```

Capitolo 9. Source code generation

This unit will create source code files using the content already added to the macro store.

The procedure `ProcessaFiles` read all the macro store and find the macro that define output files:

⟨ ***procedure ProcessaFiles*** *93* ⟩ ≡

```
procedure ProcessaFiles();
var
  i:Integer;
  Marcatore_Inizio:String;
  Marcatore_Fine:String;
begin
  for i:=0 to store.MacroCount-1 do
  begin
    if store.GetRecord(i).macroType = FileMacro then
    begin
      Assign(streamOutputSorgenti, Trim(store.GetRecord(i).macroName));
      Rewrite(streamOutputSorgenti);

      PrendiMarcatori (store.GetRecord(i).macroName,
        Marcatore_Inizio, Marcatore_Fine);
      ScriviScrapEspanso(True, store.GetRecord(i).macroName, 0,
        Marcatore_Inizio, Marcatore_Fine);
      Close(streamOutputSorgenti);
    end;
  end;
end;
```

Usata da: 97

Every file is generated using the macro name. This procedure will divide the scrap in lines and will process directives including other macros.

If it finds a macro reference it will recursively call the same procedure processing the indentation level which, initially, is zero.

⟨ ***procedure ScriviScrapEspanso*** *94* ⟩ ≡

```
procedure ScriviScrapEspanso(isMain:Boolean; nome:String;
  indent:Integer; inizioCommento:String; fineCommento:String);
var
  rec:TMacroRecord;
  i, j:integer;
  tempStringa:String;
  tempIndentazione:String;
  linea:String;
  indicazioneRiga:String;

begin
  tempIndentazione := ’’;
  for i := 1 to indent do
  begin
    tempIndentazione := tempIndentazione + ’ ’;
  end;

  if (not isMain) and GetGenerazioneMarcatoriAbilitata() then
  begin
    writeln (streamOutputSorgenti, tempIndentazione, inizioCommento,
      ’[open] ’, nome, fineCommento);
  end;

  <ScriviScrapEspanso, scrittura righe del file sorgente 95>

  if (not isMain) and GetGenerazioneMarcatoriAbilitata() then
  begin
    writeln (streamOutputSorgenti, tempIndentazione, inizioCommento,
      ’[close] ’, nome, fineCommento);
  end;
end;
```

Usata da: 97

The line inside a file are written from the macro store:

⟨ ***ScriviScrapEspanso, scrittura righe del file sorgente*** *95* ⟩≡

```
rec := store.GetMacro(nome);
if rec=Nil then
begin
  writeln(streamOutputSorgenti, ’<’, nome, ’>’);
  LogErrorMessage(’Attenzione: macro ’ + nome + ’ sconosciuta’);
end
else
begin
  for i := 0 to rec.MacroLinesCount-1 do
  begin
    linea := rec.MacroLine[i].Content;
    tempStringa := Trim(linea);

    <ScriviScrapEspanso, calcolo indicazione del numero di riga 96>

    if AnsiStartsStr(’@<’, tempStringa) and AnsiEndsStr(’@>’, tempStringa) then
    begin
      ScriviScrapEspanso(False,
        MidStr(tempStringa, 3, Length(tempStringa)-4),
        indent +
        Length(linea) - Length(TrimLeft(linea)),
        inizioCommento,
        fineCommento);
    end
    else
    begin
      writeln(streamOutputSorgenti, tempIndentazione, linea, indicazioneRiga);
    end;
  end;
end;
```

Usata da: 94

The row number is calculated, if the parameters are saying so:

⟨ ***ScriviScrapEspanso, calcolo indicazione del numero di riga*** *96* ⟩≡

```
if GetGenerazioneNumeriRigaAbilitata() then
begin
  indicazioneRiga := ’’;

  for j := 1 to GetColonnaNumeriRiga()-(length(linea)+length(tempIndentazione)) do
  begin
    indicazioneRiga := indicazioneRiga + ’ ’;
  end;

  indicazioneRiga := indicazioneRiga +
    inizioCommento+
    rec.MacroLine[i].FileName +
    ’:’ +
    IntToStr(rec.MacroLine[i].LineNumber) +
    fineCommento ;
end
else
begin
  indicazioneRiga := ’’;
end;
```

Usata da: 95

We are at the point to enounce the main file:

⟨ ***FILE slit.pas*** *97* ⟩≡

```
{$MODE objfpc}
{$H+}
program slit;
```

```
uses Classes, macrostore, sysutils,
    strutils, slitstream, slithtml,
    slittxt, slitoutput, slitlout,
    drivermagazzino, driverdoc,
    slitstatus;

var
  store:TMacroStore;
  stream:TSlitStream;
  streamOutputDocumentazione: TSlitOutput;
  streamOutputSorgenti:Text;
  driverMagazzinoMacro: TSlitStreamDriver;
  driverScriviDocumentazione: TSlitStreamDriver;

  <procedure ScriviScrapEspanso 94>

  <procedure ProcessaFiles 93>

  <slit ControllaMacroNonUtilizzate 6>

  <slit procedura principale 1>
```

Goodbye and happy literate programming!

Capitolo 10. HTML generation utilities

HTML generation is a little pecurial because text documentation need a specific syntax: some character must be escaped as HTML entities.

So I've made a function to output a string of text in HTML format and this function is used to print scraps.

This is the declaration:

⟨ ***text2html dichiarazione*** *98* ⟩≡

```
function text2html(str:String):String;
```

Usata da: 101

The implementation need to read every character in the string:

⟨ ***text2html*** *99* ⟩≡

```
function text2html(str:String):String;
var
  buffer:String;
  i:Integer;
begin
  buffer := '';
  for i:=1 to Length(str) do
  begin
    <text2html controllo carattere 100>
  end;

  Result := buffer;
end;
```

Usata da: 101

Every character, if needed, is escaped with the relative HTML entity:

⟨ ***text2html controllo carattere*** *100* ⟩≡

```
if str[i]='<' then
begin
  buffer := buffer + '&lt;';
end
else if str[i]='>' then
begin
  buffer := buffer + '&gt;';
end
else if str[i]='&' then
begin
```

```
    buffer := buffer + ’&amp;’;
  end
  else
  begin
    buffer := buffer + str[i];
  end;
```

Usata da: 99

This is the declaration of the `htmlutils` unit:

⟨ ***unit htmlutils*** *101* ⟩≡

```
{$MODE OBJFPC}
{$M+}
unit htmlutils;

interface

  uses sysutils, strutils;

  <text2html dichiarazione 98>

implementation

  <text2html 99>
end.
```

Usata da: 102

⟨ ***FILE htmlutils.pas*** *102* ⟩≡

```
  <unit htmlutils 101>
```