NATIONAL CENTER FOR SCIENTIFIC RESEARCH
"DEMOKRITOS"

Institute of Informatics & Telecommunications

Software & **K**nowledge **E**ngineering **L**aboratory

Developers' Guide to *Ellogon*

Stergos D. Afantenos
George Petasis
Vangelis Karkaletsis

June 2002

Table of Contents

1 INTRODUCTION 2

1.1 WHAT IS *ELLOGON*? 2
1.2 ESSENTIAL INGREDIENTS 3
1.3 WHAT FOLLOWS 3
1.4 ACKNOWLEDGMENTS 4

2 COMPONENTS' FILES 5

2.1 CREATING A COMPONENT 5
2.2 COMPONENTS' FILES 5
2.2.1 THE CONFIGURATION FILE 6
2.2.2 THE SOURCE FILE 9

3 *ELLOGON* API THROUGH AN EXAMPLE 12

3.1 THE GAZETTEER LOOKUP 12
3.2 THE CODE 13
3.2.1 THE INITIALIZATION PROCEDURE 13
3.2.2 THE FINALIZATION PROCEDURE 14
3.2.3 THE MAIN PROCEDURE 14

4 IMPORTING EXTERNAL COMPONENTS TO *ELLOGON* 20

4.1 A PART OF SPEECH TAGGER 20
4.2 IMPORTING EXTERNAL COMPONENTS 20

5 FURTHER INFORMATION ABOUT THE *ELLOGON API* 29

Chapter 1

1 Introduction

Natural Language Processing (NLP) emerged as a mixed field of Computational Linguistics and Artificial Intelligence, during the 1950's. Since then there is a constant outburst of interest, not only from people involved in academic research, but also from companies involved in the production and commercial exploitation of language engineering systems.

What all those people have in common is the need for a tool that will assist them on their research. Here, in the SKEL laboratory of the NCSR "Demokritos", we have developed *Ellogon*, a multi-lingual, cross-platform text-engineering environment developed exactly to aid people who are doing research in Computational Linguistics, as well as companies which produce and deliver language engineering systems. *Ellogon* was developed in an attempt to create the necessary infrastructure to facilitate the development and distribution of various NLP tools.

1.1 What is *Ellogon*?

Before proceeding with what the *Ellogon platform* is, we would like to tell some words about what *Ellogon*, as a *word*, means and how come we chose that particular name. *Ellogon* (Ελλογον) is composed from the ancient Greek words εν + λόγος, which taken together mean "in accordance with logic". The *Ellogon platform* is actually in accordance with the logic, the logic that lurks underneath each writer of a text, and helps scientists, working in the area of NLP, exploit that logic and make bare all the information that lie inside a passage. But there is more to the story. "Logos", furthermore, can be translated as oration, talk, utterance or written speech. We chose to name our platform *Ellogon* because it deals with texts, written speeches or passages in other words, and makes the information that lie inside such passages emerge.

Had we to describe the *Ellogon* platform within a few words, we could say that *Ellogon* is a general-purpose text-engineering *platform*. The word *"platform"* was emphasized here, in order to show that *Ellogon* is simply an environment and as such, it does not claim to perform any sort of linguistic processing. This is done with the use of external embeddable components, which may be written either in Tcl/Tk or in C/C++.

In another manual, the *User Guide to Ellogon*, we described the process of creating such components, but didn't delve into the details of how to write code using the API of *Ellogon*. The purpose of this manual is to do exactly that. That is, to explain in detail the *Ellogon* API.

1.2 Essential Ingredients

Before proceeding with the details that will enable one to create, test and debug components on *Ellogon*, there are some things that one has to know and have some experience with them, in order for this manual to be comprehensive.

The best introduction to the *Ellogon* is the *User Guide to Ellogon*. The potential developer of components for *Ellogon* ought to have read that manual and have some experience with the *Ellogon* through the components that someone else has written, or through the components that are provided with the standard edition of *Ellogon*. If this sounds too much, one ought to have acquired an understanding, at least, of the Data Model of *Ellogon*, described in Chapter 2 of the *User Guide to Ellogon*. Some understanding of the details described in Chapter 3 "Working with *Ellogon*" is also needed.

If you have already read that manual, then you will know that the core of *Ellogon*, the *Collection and Document Manager (CDM)*, is written in C++ and that linguistic processing is done with the use of external embeddable components, which may be written either in Tcl/Tk or in C/C++. Thus a basic requirement for writing components in *Ellogon* is to know very well either Tcl/Tk or C/C++ or even, preferably, both. In this manual we shall take for granted that you are an experienced programmer in Tcl/Tk and C/C++.

1.3 What follows

Apart from this introductory chapter, this manual contains three more chapters. In the second chapter we shall freshen up your memory on how to create a component. The main discussion of that chapter will be the files that the *Ellogon* creates for the component, and the procedures that those files contain.

In the third chapter, we shall take a first look on the *Ellogon API* through an example. The example will be the construction of a simple gazetteer lookup.

In the last chapter, we are going to describe the process of importing external executable programs into *Ellogon* Components.

Note that in this preliminary draft version of the *Developers' Guide to Ellogon* we shall limit ourselves to Tcl. In the following versions of this manual, our discussion will be expanded so that it will also include C/C++.

1.4 Acknowledgments

We would like to thank our colleagues from the NLP group of the University of Sheffield and especially the development team of the GATE 1 text engineering platform. Our cooperation with the University of Sheffield in the context of the R&D projects ECRAN[1] and GIE[2], motivated us to be actively involved in the area of text engineering platforms and make our first efforts towards the development of *Ellogon*.

[1] ECRAN is an R&D project on Information Extraction, partially funded by the EC (Telematics Applications / Language Engineering Action), 12/1995 – 02/1999. ECRAN partners were Thomson-CSF (France, coordinator), Universita di Ancona (Italy), Universita di Roma "Tor Vergata" (Italy), Smart Information Services GmbH (Germany), NCSR "Demokritos" (Greece), Friburg University (Switzerland), University of Sheffield (UK).

[2] GIE is a bilateral project (English-Greek) on named entity recognition, funded by the Greek General Secretariat of Research & Technology and the British Council, 05/1997-05/1999. GIE partners were NCSR "Demokritos" and the University of Sheffield.

Chapter

2

2 Components' Files

In Chapter 3 of the *User Guide to Ellogon*, and in particular in sections 3.6 and 3.7, we described how to create and modify components. We presume that you are already acquainted with those sections. In this chapter, we shall refresh your memory on how to create components, and then we shall describe the files that *Ellogon* creates and the procedures that they contain.

2.1 Creating a component

In the *Ellogon* main window, select `Create New Module` from the `Module` menu. In the dialog box that opens, define a name for the component and set the various parameters, such as the pre-conditions, post-conditions and the various viewers that will be attached to the component.

For the purpose of the example that will follow in the next chapter we shall simply set the name of the component as `ExampleGazeteer`[3] and the component's title will be set to "Example Gazeteer". No pre-conditions or post-conditions will be set. Also, we will not associate any viewer with the component.

When you have finished push the "Save" button on the dialog. (See also *Figure 2.1*)

2.2 Components' files

Once you have pushed the "Save" button, *Ellogon* will create a new folder in the directory you have specified, which will have the same name as the component name. In the case we have a windows system, that folder will be the following:

```
C:\modules\ExampleGazeteer
```

In case we have a UNIX system, that folder will be

```
~/modules/ExampleGazeteer
```

where the ~ represents your home directory.

[3] Components' names do not contain spaces and some other characters such as (; , : . -) In general valid component names are valid names of C++ functions. See also Section 3.6 of the *User's Guide to Ellogon*.

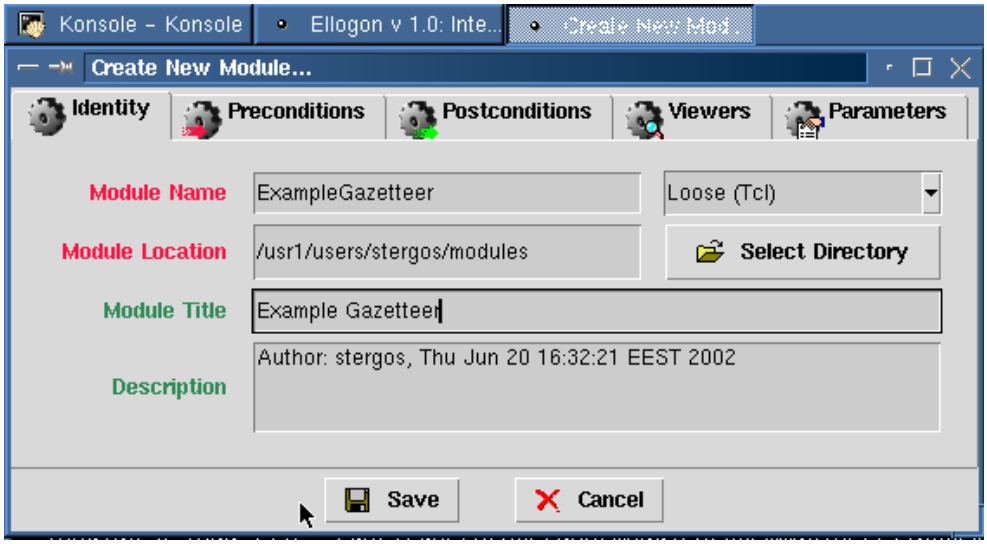

Figure 2.1 Creating a new component

2.2.1 The configuration file

In that directory two fresh files will be created. The first one will always be called `creole_config.tcl` and you can see it in *Figure 2.2*.

As you can see, this file contains several pieces of information. In summary, the information that this file contains are the following:

- *Title:*
  It contains the title of the component. In our example it is the line: `title {Example Gazeteer}`

- *Pre-conditions:*
  It contains information about the collection and document attributes and the annotations that must exist as pre-conditions. These pre-conditions must be satisfied in order for the component to be able to execute correctly. In our example we set no pre-conditions for the Collection and Document attributes, but we set pre-conditions for the annotations that the Collection should contain. The line `annotations {{token type}}` indicates that the Documents of the Collection should contain annotations "token" with the attribute "type".

- *Post-conditions:*
  Similarly, post-conditions contain information about the collection and document attributes and the annotations that will be added when the component is executed. In our example no Collection attributes will be added when the module finishes processing, but the Documents will get the attribute "ExampleGazeteer". The line `annotations {lookup {lookup type}}` indicates that when the module finishes processing the Documents will contain "lookup" annotations with attribute "type".

- *Viewers*
  It contains information about the viewers (such as the single-span or the raw text viewer) that are associated with the component. These viewers can be easily accessible through a special menu offered by *Ellogon* GUI when the component has been executed. That menu appears if you click on the component once it has finished processing. In our example we have associated four viewers with that component. The line `{lookup type} single_span {Lookup Annotations...}` indicates that a single span viewer will be associated with the component which will contain the "lookup" annotations with attribute "type" and will be labelled "Lookup Annotations …". Similarly for the line `{token type} single_span {Token Annotations...}`. The line `{lookup} raw {Raw Lookup Annotations...}` is instructing *Ellogon* to associate a raw viewer with the component, which will show the "lookup" annotations and it will be labelled "Raw Lookup Annotations". Finally the line `{token} AnnotationExplorer {Explore Annotations...}` will associate an Annotation Explorer viewer that will contain the "token" annotations and iyt will be labelled "Explore Annotations…".

- *Parameters*
  Information concerning the parameters (see section 3.6.3 of the *User Guide to Ellogon*) is stored here. In our case we have set three parameters for the example component that we have created. The first two are two files which contain the lists that comprise the gazetteer. The lines `{Location List} %FILE% {$creole_ExampleGazetteer_home/gazloc.lst}` and `{Person List} %FILE% {$creole_ExampleGazetteer_home/gazper.lst}` indicate that two variables `{Location List}` and `{Person List}` will contain the paths for the location and person file respectively (see *Table 3.1*). The variable `$creole_ExampleGazetteer_home` contains the path of the component. The third parameter `{Dummy Boolean Parameter} 0 {}` contains simply a dummy Boolean parameter that serves no actual purpose but only to show you how to define Boolean parameters. It is set to 0 so the checkbox will appear unchecked.

- *Language*
  It contains the information about the programming language in which the component is written. In our example, `coupling loose` means that the component is written in Tcl.

- *Description*
  A small description of the component, as given in the dialog box, is contained herein. (see *Figure 2.1*)

- *Compatibility-Mode*
  This variable declares the compatibility mode under which this component should be executed. Currently, two compatibility modes are available: *Ellogon's* native mode (value 0) and GATE 1 compatibility mode (value 1). Available values may be increased in future versions of Ellogon, if addi-

tional compatibility modes are supported. In our example, the component will not use any compatibility mode, as it is an *Ellogon* native component.

```
#
#       ExampleGazetteer/creole_config.tcl - configuration file
#
#       Tuesday June 11 14:42:10 (EEST) 2002
#
#       $Id: template_config.tcl, Ellogon, version 1.0...
#           (Georgios Petasis, 23/11/1998), petasis@iit.demokritos.gr
$


set creole_config(ExampleGazetteer) \
{
    title {Example Gazetteer}
    pre_conditions
    {
      collection_attributes      {}
      document_attributes        {}
      annotations                {{token type}}
    }
    post_conditions
    {
      collection_attributes      {}
      document_attributes        {ExampleGazetteer}
      annotations                {lookup {lookup type}}
    }
    viewers
    {
      {lookup type} single_span        {Lookup Annotations...}
      {token type}  single_span        {Token Annotations...}
      {lookup}      raw                {Raw Lookup Annotations...}
      {token}       AnnotationExplorer {Explore Annotations...}
    }
    parameters
    {
      {Location List} %FILE% {$creole_ExampleGazetteer_home/gazloc.lst}
      {Person List}   %FILE% {$creole_ExampleGazetteer_home/gazper.lst}
      {Dummy Boolean Parameter} 0 {}
    }
    coupling loose
    description {Author: stergos, Tue Jun 11 14:28:06 EEST 2002}
    module_encoding iso8859-7
};# ExampleGazetteer

## Compatibility Mode: Use 0 for Ellogon mode.
set ::CDM::ComponentMode(creole_ExampleGazetteer) 0

#
# End of File
#
```

Figure 2.2 The creole_config.tcl file

The above information was placed to the configuration file according to the specifications we have given on the dialog box depicted in *Figure 2.1*. In case you want later to change something (*e.g.* a pre-condition) you shall have to manually enter the changes into the creole_config.tcl file. Generally, that is not recommended, unless of course you are an expert *Ellogon* user and developer.

2.2.2 The source file

The other file that is created by *Ellogon* is the source file, in which our source code, that will perform a specific job, will be placed. You can see that file in *Figure 2.3*. The name of that file is the name of the component that we gave in the dialog box depicted in *Figure 2.1* with the ".tcl" extension.

As you can see, this file contains three procedures and defines a variable. The variable always has the form:

`creole_`{the name of the component}`_home`

In our example that variable is the `creole_ExampleGazeteer_home`. This variable contains the path in which the component's source and configuration files are and its value is automatically set by *Ellogon*. Component developers should always use the value of this variable in order to locate resources needed by the component, that are stored in files or directories stored relatively to the location of the component.

The procedures have always the form:

`creole_`{the name of the component}
`creole_`{the name of the component}`_Initialize`
`creole_`{the name of the component}`_Finish`

In our example, the procedures have the form:

```
creole_ExampleGazeteer
creole_ExampleGazeteer_Initialize
creole_ExampleGazeteer_Finish
```

From the names of the procedures, quite intuitively, you can understand what the role of each one is. Sometimes, when you open a Collection, or a single Document, you might want some initial job to be done as necessary preamble for the main job you are going to perform to the Collection later. For example, you might want to open some files and initialize some lists ad arrays, according to those files, or allocate some memory, etc. Similarly, after you have done some job to the Collection, you might want to write a piece of code which will perform a finalization to the main code you have written. For example you might want to close some open files, free the memory you allocated, etc.

Such initializations and finalizations are very common. Thus apart from the main procedure, (`creole_ExampleGazeteer` in our example), we provide you with two more procedures (`creole_ExampleGazeteer_Initialize` and `creole_ExampleGazeteer_Finish` in our example) which should contain the initializations and finalizations you might wish to have.

```
##############################################################################
#
#       ExampleGazeteer.tcl - Thursday May 23 17:48:29 (GTB Daylight Time) 2002
#        This is a loosed coupled (Tcl) Module for use with
#        Ellogon, version 1.00...
#
##############################################################################


# Location of this module (ExampleGazeteer)
global creole_ExampleGazeteer_home

## creole_ExampleGazeteer
#
proc creole_ExampleGazeteer {doc args} {
  global creole_ExampleGazeteer_home
  set current_dir [pwd]
  cd $creole_ExampleGazeteer_home

  ## Under Ellogon, the window that desplays the "wait" message can
  ## additionally display a progress bar. In order to activate this feature,
  ## you must execute the command "ggi_wait_update document percent"
  ## In fact, if you enclose the command in a catch block, this module will
  ## also run under GATE without problems. The second argument to the
  ## ggi_wait_update procedure is the percent, which is in the range [0,100]
  ## Setting percent a negative value causes the progress bar to disappear.
  ##   catch {ggi_wait_update $doc 0}

  ## Put your code here...

  # record the fact that we ran and exit normally
  cd $current_dir
  tip_PutAttribute $doc [tip_CreateAttribute ExampleGazeteer \
                              [tip_CreateAttributeValue GDM_STRING {}]]
};# creole_ExampleGazeteer


## Procedure: creole_ExampleGazeteer_Initialize
#   Use this function in order to do some initialization. This function will
#   be called just before creole_ExampleGazeteer in the following situations:
#    *) If the user has opened a whole Collection, this function will be
#       called just before the first document in Collection gets proccessed
#       with creole_ExampleGazeteer
#    *) If the user has opened a single Document, this function will be called
#       just before calling creole_ExampleGazeteer
#   Please, refer to the Ellogon's Programming Manual for more
#   information on this function...
proc creole_ExampleGazeteer_Initialize {col doc args} {
  global creole_ExampleGazeteer_home
  ## Do some initilization here...
};# creole_ExampleGazeteer_Initialize

## Procedure: creole_ExampleGazeteer_Finish
#   Use this function in order to do some Clean-Up. This function will
#   be called after all calls to creole_ExampleGazeteer in the following
#   situations:
#    *) If the user has opened a whole Collection, this function will be
#       called just after the last document in Collection gets proccessed with
#       creole_ExampleGazeteer
#    *) If the user has opened a single Document, this function will be called
#       just after calling creole_ExampleGazeteer
#   Please, refer to the Ellogon's Programming Manual for more
#   information on this function...
proc creole_ExampleGazeteer_Finish {col doc args} {
  global creole_ExampleGazeteer_home
  ## Clean-Up...
};# creole_ExampleGazeteer_Finish

#
# End of File
#
```

Figure 2.3 The source file

One thing that is very important to keep in mind, is that the initialization and finalization procedures are performed *only once* on the whole Collections, whereas the finalization procedure is preformed on *every Document* of the Collection. In other words, once you open a Collection and you begin to run a component (our example component, lets say) the first procedure that will run (and only once) is the `creole_ExampleGazeteer_Initialize`. Then, *for every Document of the Collection* the `creole_ExampleGazeteer` will be called. Finally, after the whole Collection has been processed, the `creole_ExampleGazeteer_Finish` will be called once, in order to perform some finalization.

The above will become more concrete with an example, which is the topic of the next chapter.

Chapter

3

3 *Ellogon* API through an Example

In this chapter, we are going to cast an initial glance to the *Ellogon API* through an example. In this example, we are going to see several essential procedures of the *Ellogon API*. They are essential in the sense that those procedures are the most commonly used. After we have gone through this example, you are going to be able to write code for *Ellogon* components that will tackle a fairly big proportion of common tasks. Note that you can get more help about the *Ellogon* API if you click on the "Help Contents" of the "Help" menu in the *Ellogon* interface.

3.1 The Gazetteer Lookup

Before presenting you the code and going through it, we shall have to give you more information concerning the component we are going to built and the Collection in which it is going to run.

What we are going to build is a very simple gazetteer lookup. In essence, a gazetteer is a list, or a collection of lists. Each list groups together several items belonging to the same "category". In our example the gazetteer contains information on Named Entities. More specifically it contains two lists. The first list is a list of locations and the second list is a list of persons' names. You can see the gazetteer in *Table 3.1*. Actually, the files comprising the gazetteer were given as parameters to the component (see the previous chapter).

gazloc.lst		gazper.lst
Greece	Berlin	Emmanuel
France	Thessalonica	Douglas
Oklahoma	Mississippi	Daniel
Paris	Missouri	Nicola
Corfu	California	Maria
Budapest	Barcelona	Vanessa
Rome	Gibraltar	
Madrid	Aegean	
Spain	Crete	
Portugal	Rhodes	
Germany	Santorini	

Table 3.1 The Gazetteer lists.

The Collection, upon which the component is going to run, consists of news documents written in English.

Our aim is to build a gazetteer lookup component which will locate all the instances of the gazetteer's elements and create an annotation of type `lookup`. Each annotation will have an attribute `type` which will declare whether the instance found in the corpus is a person or location. In other words, if an instance of an item of the person list was found, the attribute will be like this:

```
type=person
```

In order to search for words in the corpus and compare them against the gazetteer, we assume that the Collection has been previously processed by a tokenizer and a sentence splitter. We also assume that a tokenizer is a component which takes a Document and creates an annotation of type `token` for every token (word, punctuation mark, etc) found in the Document. The code of the component, which follows, presumes that this simple tokenizer has been run against the Collection, as is indicated by the pre-conditions. Additionally, the sentence splitter component is assumed to identify sentences and to create an Annotation of type "sentence" for each identified sentence. All "sentence" Annotations must contain an Attribute named "constituents" that should contain the Annotation Ids of all token Annotations contained inside the corresponding sentence, ordered according to the way they appear in the corpus.

3.2 The Code

The code of the component is presented in *Figure 3.1*, *Figure 3.2* and *Figure 3.3*, which contain the code for the initialization procedure, the main procedure and the finalization procedure, respectively.

3.2.1 The Initialization Procedure

The initialization procedure is quite simple. In it we define two global variables (`creole_ExampleGazetteer_Location` and `creole_EaxampleGazeteerPerson`) which will be the arrays in which the contents of the lists will be placed. The lists are contained in the files that were given as parameters to the component. Here you can see a convention that we use in order to avoid conflicts when we define global variables. We prefix each global variable with creole_{name of the module}_. Of course you could create a namespace and use that instead of the prefix.

We place the three parameters, including the dummy Boolean parameter, inside variables with the line

```
foreach {LocationList PersonList Boolean} $args {break}
```

The next step is to initialize the arrays. We open each file in turn, read its contents and initialize the arrays by placing every line at the index of the array. We set the contents of each element of the array to 0. When we are finished with each file, we close it.

3.2.2 The Finalization Procedure

The finalization procedure is even simpler than the initialization. In it we simply delete the two arrays we created in the Initialization procedure, so that they will not be in the memory any more.

```
proc creole_ExampleGazetteer_Initialize {col doc args} {
  # global arrays that will contan the information from the gazeteers
  global creole_ExampleGazetteer_Location
  global creole_ExampleGazetteer_Person

  # Place parameters in variables...
  foreach {LocationList PersonList Boolean} $args {break}

  # Initialize the arrays
  set creole_ExampleGazetteer_GazLoc [open $LocationList]
  while {[gets $creole_ExampleGazetteer_GazLoc line] >= 0} {
    set creole_ExampleGazetteer_Location($line) 0
  }
  close $creole_ExampleGazetteer_GazLoc

  set creole_ExampleGazetteer_GazPer [open $PersonList]
  while {[gets $creole_ExampleGazetteer_GazPer line] >= 0} {
    set creole_ExampleGazetteer_Person($line) 0
  }
  close $creole_ExampleGazetteer_GazPer
};# creole_ExampleGazetteer_Initialize
```

Figure 3.1 The Initialization Procedure

3.2.3 The Main Procedure

The main procedure begins by declaring that the `creole_ExampleGazetteer_Location` and `creole_ExampleGazetteer_Person` variables are global. In other words, the arrays used in the initialization procedure will be used again.

The following lines:

```
ggi_wait_update $doc 0
set number_of_tokens [llength \
[tip_SelectAnnotationsSorted $doc token]]
set progress 0.0
set step [expr {100.0/($number_of_tokens+1)}]
```

are concerned with the appearance of a progress bar, which will be filled as more tokens are being processed. The first line defines that progress bar. Then we get the number of tokens, using the `tip_SelectAnnotationsSorted`. This procedure takes two arguments. The first one is the current document (`$doc`) and the next one is the type of annotations we want (`"token"`). It returns a new list object that will contain all the Annotations of the specified Document that their type is the same as the value of the type parameter (the "token" in our case). The annotations will be sorted according to their first span range, in ascending order. We can have access to the current document through the `doc` variable passed as an argument to the main procedure. We get the number of annotations using the `llength` procedure of Tcl. The `progress` variable indicates the progress thus far (0 for the moment). The `step` indicates how much the progress will be incremented every time we process a token.

The next step is to get the text of the Document that is being processed with the procedure

```
tip_GetRawData
```

This procedure gets an argument which is the current Document of the Collection.

The next step is to get all the annotations of type "token" and iterate over all of them comparing them against all the slots of our arrays and creating the appropriate annotation, where appropriate. To get all the annotations of type "token", we use, as before, the following procedure

```
tip_SelectAnnotationsSorted
```

After that, we iterate over all the tokens contained in the sorted annotations that we got. For every token, we get the value of the annotation. In other words we get the text which corresponds to the annotation. In order to do so, we use the following procedure

```
tip_GetFirstAnnotatedTextRange
```

This procedure takes two arguments. The first one is the text of the document, which in our case is stored into the `text` variable, and the second argument is the annotation object itself, which is stored into the token variable. As a result it returns the text of the first span of the annotation.

Once we have taken that text and stored it into the `tokenValue` variable, we want to check whether that text is the same with any of the slots of the arrays. In order to examine that, we use the `info exists` procedure of Tcl to examine whether the `$tokenText` exists as an index first to the `creole_ExampleGazetteer_Location` array and then to the `creole_ExampleGazetteer_Person` array. If it exists we are trying to create a new annotation (lookup), which will have the same span as the token annotation, and will have the attribute "location" or "person", according to the array in which we found it.

In order to create the annotation we use the procedure

```
tip_CreateAnnotation
```

which takes three arguments. The first argument is the type of the annotation (`lookup` in our case). The second argument is the spans of the annotation. The third one is the attributes of the annotation.

The type of the annotation is `lookup` and it is simply a string. In order to set the spans, we use the spans of the `token` annotation. We stored them previously in the `span` variable using the procedure

```
tip_GetSpans
```

which takes as argument an annotation and returns the spans of the annotation. We let the attributes empty. Instead we used the procedure

```
tip_CreateAttribute
```

```
# Location of this module (ExampleGazetteer)
global creole_ExampleGazetteer_home

## creole_ExampleGazetteer
#
proc creole_ExampleGazetteer {doc args} {
  global creole_ExampleGazetteer_Location creole_ExampleGazetteer_Person

  ## Display a Progress bar...
  ggi_wait_update $doc 0

  set number_of_tokens [llength [tip_SelectAnnotationsSorted $doc token]]
  set progress 0.0
  set step [expr {100.0/($number_of_tokens+1)}]

  # Get the text of the document
  set text [tip_GetRawData $doc]

  # Select all the annotations of type "token"
  set tokens [tip_SelectAnnotations $doc "token"]

  # Iterate over all tokens
  foreach token $tokens {

    # Get the text annotated by the first span of the token annotation...
    set tokenText [tip_GetFirstAnnotatedTextRange $text $token]

    # Check whether the text matches any of the elements of the location
array
    if {[info exists creole_ExampleGazetteer_Location($tokenText)]} {

      # ... then get the spans of the token annotation ...
      set spans [tip_GetSpans $token]

      # ... create an annotation with no attributes ...
      set locAnnotation [tip_CreateAnnotation "lookup" $spans {}]

      # ... and then add an attribute to it ...
      set attr [tip_CreateAttribute "type" \
                  [tip_CreateAttributeValue GDM_STRING "location"]]
      # ... add the attribute into the annotation ...
      set locAnnotation [tip_PutAttribute $locAnnotation $attr]

      # ... finally, add the annotation to the document
      tip_AddAnnotation $doc $locAnnotation

    } elseif {[info exists creole_ExampleGazetteer_Person($tokenText)]} {

      # Repeat the same job for the person array...
      set spans [tip_GetSpans $token]
      # ... only that now we insert attributes immediatelly,
      set attr [tip_CreateAttribute "type" \
                  [tip_CreateAttributeValue GDM_STRING "person"]]
      set perAnnotation [tip_CreateAnnotation "lookup" $spans [list $attr]]

      tip_AddAnnotation $doc $perAnnotation
    }

    # Update the progress bar
    set progress [expr {$progress+$step}]
    catch {ggi_wait_update $doc $progress}

  };# foreach token $tokens
```

Figure 3.2 The Main Procedure

```
proc creole_ExampleGazetteer_Finish {col doc args} {

  global creole_ExampleGazetteer_Location cre-
ole_ExampleGazetteer_Person

  # Remove our arrays from memory...
  unset -nocomplain creole_ExampleGazetteer_Location \
                    creole_ExampleGazetteer_Person

};# creole_ExampleGazetteer_Finish
```

Figure 3.3 The Finalization Procedure

to create an attribute. This procedure takes as arguments the type of the attribute (`type` in our case) and the value of the attribute. In order to create the attribute value, we used the

```
tip_CreateAttributeValue
```

procedure which takes two arguments: an integer and a string. The integer is the type of the attribute value, and in our case we used the constant

```
GDM_STRING
```

The string (*i.e.* the value of the attribute) is set to `location` or `person`, according to the array in which we iterate.

Once the attribute is created, and stored in the `attr` variable, we use the

```
tip_PutAttribute
```

procedure to put that attribute into the annotation. This procedure takes two arguments: the annotation and the created attribute. It returns a new annotation with the attribute now added.

Finally, the annotation is created. The only thing left now is to put that annotation into the Document. We accomplish this by the

```
tip_AddAnnotation
```

procedure. This one takes two arguments: the document that the annotation is going to be added in and the annotation itself.

Note that the same algorithm is used for both arrays. The only difference is that in the case of the `creole_ExampleGazetteer_Person` array, we do not use the `tip_PutAttribute` procedure, instead we insert the attribute immediately in the `tip_CreateAnnotation` procedure, by using the expression `[list $attr]` instead of an empty list that we used before.

The result is that new annotations of type NE have been created. You can see them in *Figure 3.4*.

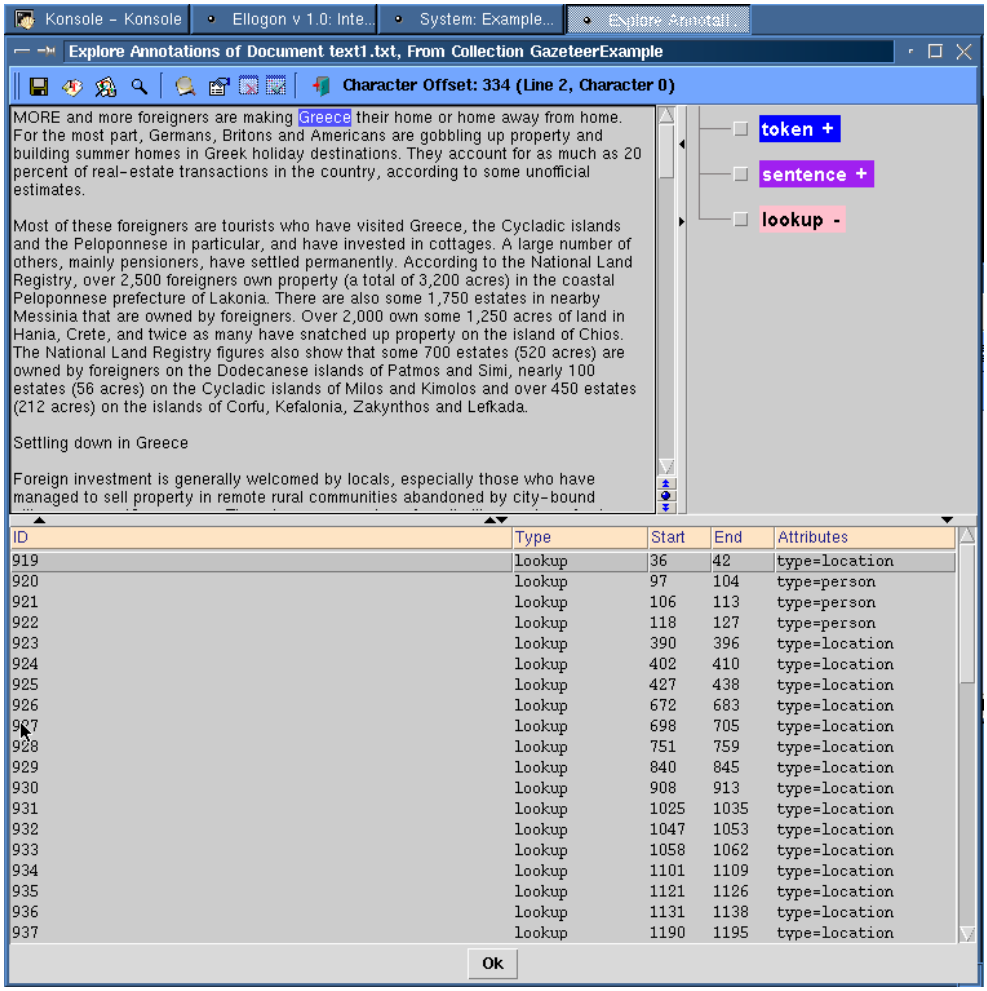

Figure 3.4 The Created Annotations

Finally, we have added some more code, which does nothing essential to the algorithm, just to show you some more procedures of the *Ellogon API*. You can see that segment of code in *Figure 3.5*.

```
# Create a temporary file with a unique name write the parameters in
it...
  global CDM_TempDir
  # Use file join to create the filename, in order to be platform in-
dependent.
  set TempFileName [file join $CDM_TempDir cre-
ole_ExampleGazetteer_[pid]]
  set TempFile     [open $TempFileName w]
    foreach {LocationList PersonList Boolean} $args {break}
    puts $TempFile "Location List Parameter: $LocationList"
    puts $TempFile "Person List Parameter:   $PersonList"
    puts $TempFile "Dummy Boolean Parameter: $Boolean"
  close $TempFile

  # Now, delete the temporary file...
  file delete -force $TempFileName

  # record the fact that we ran and exit normally
  tip_PutAttribute $doc [tip_CreateAttribute ExampleGazetteer \
                              [tip_CreateAttributeValue GDM_STRING {}]]
};# creole_ExampleGazetteer
```

Figure 3.5 The final segment of code for the main procedure of the component

In that segment of code, we are trying to create a file with a unique name in a temporary folder, print the parameters in there, close the file and then delete it, since we do not actually need it.

The unique file name is achieved by appending to `creole_ExampleGazetteer_` the process id, which we take from the operating system using the `pid` procedure. The path of the temporary folder is returned from the *Ellogon* variable `CDM_TempDir`.

We put the contents of the parameters in variables by using the following code

```
foreach {LocationList PersonList Boolean} $args {break}
```

exactly as we did in the initialization procedure. Then we write each parameter in the file along with a comment and close it. Finally we delete the file using the following code

```
file delete -force $TempFileName
```

Chapter

4

4 Importing External Components to *Ellogon*

Under the *Ellogon* platform you can not only write your own Components, but you can import external components written in some other language. The general idea behind importing external components is that an executable file needs some files as input and writes its output in some other files; so, we only have to write a fragment of code that will create the input files that the executable needs and use the output files that it creates. Then we can do whatever job we want with the output files, *e.g.* create some *Ellogon* annotations and place them inside the Collection.

4.1 A Part of Speech Tagger

Let us look at an example, so that the above will become clearer. In this example we are going to import an executable file which is in fact a part of speech tagger (POS tagger) which was written by Eric Brill in 1995 and was modified by George Petasis in 1999.

This POS tagger is looking at each token in a sentence and it tries to identify what part of speech this token is. In order to do so, it uses four additional files as input:

Bigrams, ContextualRules, FinalLexicon, LexicalRules

If no errors occur, this POS tagger creates a new file which contains the part of speech for every token in the text.

Our aim is to prepare those input files for the POS tagger, execute it, and get the output file. Then, we have to take the information contained inside that output file and convert them into *Ellogon* annotations.

4.2 Importing External Components

The process of importing external components into *Ellogon* is quite easy. We begin by creating a new component for *Ellogon*, exactly as we did in the example of the previous chapter. *Ellogon* will create two files, the configuration file and the source

file. In our case we have named the component HBrill and you can see the configuration file in *Figure 4.1*, and the source file in *Figure 4.2*.

```
#
#        HBrill/creole_config.tcl - configuration file
#
#        Saturday September 11 14:31:19 (EEST) 1999
#
#        $Id: template_config.tcl, Ellogon, version 1.0...
#            (Georgios Petasis, 23/11/1998), petasis@iit.demokritos.gr
$


set creole_config(HBrill) {
      title {Greek POS Tagger}
      pre_conditions
      {
        collection_attributes      {}
        document_attributes        {language_english}
        annotations                {}
      }
      post_conditions
      {
        collection_attributes      {}
        document_attributes        {HBrill}
        annotations                {{token pos}}
      }
      viewers
      {
        {token pos} single_span {Brill POS Tags...}
        {token}     raw         {Raw Token}
        $creole_HBrill_home/GreekTags.html text_file \
                                {Greek POS Tag Definitions}
      }
      parameters
      {
        {Lexicon}          %FILE% \
                           "$creole_HBrill_home/greek/FinalLexicon"
        {Bigrams}          %FILE% "$creole_HBrill_home/greek/Bigrams"
        {Lexical Rules}    %FILE% \
                           "$creole_HBrill_home/greek/LexicalRules"
        {Contextual Rules} %FILE% \
                           "$creole_HBrill_home/greek/ContextualRules"
        {Additional Wordlist}     {-w %FILE%}   {}
        {Intermediate output}     {-i %FILE%}   {}
        {Process lines}           {-s %NUMBER%} {}
        {Start State Tagger only} -S            {}
      }
      coupling dynamic
      description
        "This is the Greek Part-of-Speech Tagger \
         (based on Brill Tagger).\n\
         Author: Petasis Georgios, petasis@iit.demokritos.gr"
      module_encoding iso8859-7
}

## Compatibility Mode: Use 0 for Ellogon mode, 1 for GATE compatibility
## mode...
set ::CDM::ComponentMode(creole_HBrill) 0

#
# End of File
#
```

Figure 4.1 The configuration file

In the configuration file, you can see that we have defined several values for the pre-conditions and post-conditions, we have associated a single span and a raw viewer, we have set as parameters the four files that the Component needs as input, *etc.*

The only pre-condition that we have set is for the natural language of the Documents to be English, as can be seen by the lines

```
pre_conditions
{
	collection_attributes	{}
	document_attributes	{language_english}
	annotations		{}
}
```

Then, we have set two post conditions. The one concerns a document attribute which states that the Document has been run against this module (HBrill) and the second that two annotations will be added to the Documents of this Collection, the token and pos annotations. Those are indicated by the lines

```
post_conditions
{
	collection_attributes	{}
	document_attributes	{HBrill}
	annotations		{{token pos}}
}
```

We have also associated two viewers, a raw and a single span viewer, as you can see from the lines

```
viewers
{
	{token pos} single_span {Brill POS Tags...}
	{token} raw {Raw Token}
	$creole_HBrill_home/GreekTags.html text_file \
		{Greek POS Tag Definitions}
}
```

Furthermore, we have set several parameters, including the four files that are to be the input to the external module, as is shown in the lines

```
parameters
{
	{Lexicon}		%FILE% \
			"$creole_HBrill_home/greek/FinalLexicon"
	{Bigrams}		%FILE%
			"$creole_HBrill_home/greek/Bigrams"
	 {Lexical Rules}	%FILE% \
			"$creole_HBrill_home/greek/LexicalRules"
	 {Contextual Rules} %FILE% \
			 "$creole_HBrill_home/greek/ContextualRules"
	 {Additional Wordlist}	{-w %FILE%}	{}
	 {Intermediate output}	{-i %FILE%}	{}
	 {Process lines}	{-s %NUMBER%}	{}
	 {Start State Tagger only} -S		{}
}
```

The procedure for adding the above is exactly the same as the one we followed in the example of the previous chapter.

```
##############################################################################
##
#
#       HBrill.tcl - Saturday September 11 14:31:19 (EEST) 1999
#        This is a loosed coupled (Tcl) Module for use with
#        Ellogon, version 1.0...
#
##############################################################################
##


# Location of this module (HBrill)
global creole_HBrill_home HBrill_tagger_home
set HBrill_tagger_home \
    [file join $creole_HBrill_home RULE_BASED_TAGGER_V1.14 Bin_and_Data]

## creole_HBrill
#
proc creole_HBrill {doc args} {
  global creole_HBrill_home HBrill_tagger_home CDM_TempDir
  set current_dir [pwd]
  cd $creole_HBrill_home

  ## Under Ellogon, the window that desplays the "wait" message can
  ## additionally display a progress bar. In order to activate this
  ## feature,you must execute the command "ggi_wait_update document
  ## percent". In fact, if you enclose the command in a catch block,
  ## this module will also run under GATE without problems. The second
  ## argument to the ggi_wait_update procedure is the percent, which is
  ## in the range [0,100] setting percent a negative value causes the
  ## progress bar to disappear.
  catch {ggi_wait_update $doc 0}

  ## Create temporary file names...
  if {![info exists CDM_TempDir]} {set CDM_TempDir /tmp}
  set tmp_dump_fname [file join $CDM_TempDir HBrill_tmp[pid]_dump]
  set tmp_read_fname [file join $CDM_TempDir HBrill_tmp[pid]_read]
  set dev-null       [file join $CDM_TempDir HBrill_tmp[pid]_delete]

  ## The first four arguments are the needed files...
  foreach {lexicon_file bigrams_file lexrule_file context_file} $args
          {break}
  if {![file readable $lexicon_file]} {
    error "unreadable lexicon file parameter $lexicon_file: \
           creole_HBrill"
  }
  if {![file readable $bigrams_file]} {
    error "unreadable bigrams file parameter $bigrams_file: \
           creole_HBrill"
  }
  if {![file readable $lexrule_file]} {
    error "unreadable lexical rule file parameter $lexrule_file:\
           creole_HBrill"
  }
  if {![file readable $context_file]} {
    error "unreadable contextual rule file parameter $context_file:\
           creole_HBrill"
  }
  set other_args [lrange $args 4 end]
  foreach arg $other_args {
    set flag [string range [lindex $arg 0] 1 end]
    set value [lindex $arg 1]
```

```
  switch -exact -- $flag {
    w {
      if {![file readable $value]} {
        error "unreadable wordlist file parameter $value: \
               creole_HBrill"}
    }
    i {
      if {[file exists $value]} {
        if {![file writable $value]} {
          error "unwritable intermediate file parameter $value:\
                 creole_HBrill"
        }
      } else {
        if {![file writable [file dirname $value]]} {
          error "unwritable directory for intermediate file parameter\
                 $value: creole_HBrill"
        }
      }
    }
    s {
      if {$value < 100} {
        error "too small value for process lines parameter $value:\
               creole_HBrill"}
    }
    S {}
    default {error "bad option parameter $flag: creole_HBrill"}
  }
}

# open file to use as input to external process
if {[catch {open $tmp_dump_fname w} tmp_dump]} {
    error "Cannot open $tmp_dump_fname: $tmp_dump"
}
# write out required info
creole_dump_HBrill_file $tmp_dump $doc
close $tmp_dump

# call the external process

# (HBrill needs to run in its own directory)
set current_dir [pwd]
cd $HBrill_tagger_home
# (and needs its directory in the path as well, so use a subshell)
if {[catch {
  eval exec tagger [list [file nativename $lexicon_file] \
              [file nativename $tmp_dump_fname] \
             [file nativename $bigrams_file] \
              [file nativename $lexrule_file] \
             [file nativename $context_file]] \
             [join $other_args] > $tmp_read_fname 2> ${dev-null}} \
             error]} {
  # (HBrill's exit status is garbage so ignore it, but report others)
  global errorCode
  catch [file delete -force ${dev-null}]
  if {[lindex $errorCode 0] != "CHILDSTATUS"} {
    error "HBrill exited abnormally: $error"
  }
}
cd $current_dir
catch [file delete -force ${dev-null}]

# open the external process's output file
if {[catch {open $tmp_read_fname r} tmp_read]} {
  error ">>Cannot open $tmp_read_fname: $tmp_read"
}
# parse and add info to the database
if {[catch {creole_read_HBrill_file $tmp_read $doc} error]} {
  close $tmp_read
  error $error
}
close $tmp_read
catch {ggi_wait_update $doc 100}
```

```
  # delete tmp files
  catch [file delete -force $tmp_dump_fname]
  catch [file delete -force $tmp_read_fname]

  # record the fact that we ran and exit normally
  cd $current_dir
  tip_PutAttribute $doc [tip_CreateAttribute HBrill \
                             [tip_CreateAttributeValue GDM_STRING {}]]
  return
} ;# creole_HBrill


# creole_dump_HBrill_file
#
proc creole_dump_HBrill_file { dump_file doc } {

  # get text string
  set ByteSequence [tip_GetByteSequence $doc]

  # find sentences
  set SentenceAnns [tip_SelectAnnotations $doc sentence {}]
  set items [llength $SentenceAnns]
  if {!$items} {
    error "creole_HBrill: No Sentence Annotations Found!"
  }
  set step [expr {33.0/$items}]
  set progress $step

  foreach sentence $SentenceAnns {
    catch {ggi_wait_update $doc $progress}
    set progress [expr {$progress + $step}]
    set constituents [tip_GetValueValue \
                          [tip_GetAttribute $sentence constituents]]
    foreach token_id $constituents {
      set token [tip_GetAnnotation $doc $token_id]
      set span  [lindex [tip_GetSpans $token] 0]
      set start [tip_GetStart $span]
      set end   [tip_GetEnd   $span]
      set text  [string range $ByteSequence $start [expr {$end - 1}]]
      puts -nonewline $dump_file "$text "
    }
    puts $dump_file {};#newline
  }
} ;# creole_dump_HBrill_file


# creole_read_HBrill_file
#
proc creole_read_HBrill_file {read_file doc} {

  # find sentences
  set SentenceAnns [tip_SelectAnnotations $doc sentence {}]
  set items [llength $SentenceAnns]
  if {!$items} {
    error "creole_HBrill: No Sentence Annotations Found!"
  }
  set step     [expr {33.0/$items}]
  set progress [expr {66.0 + $step}]
```

```
foreach sentence $SentenceAnns {
    catch {ggi_wait_update $doc $progress}
    set progress [expr {$progress + $step}]
    set constituents [tip_GetValueValue \
                          [tip_GetAttribute $sentence constituents]]
    set line [gets $read_file]
    if {[eof $read_file]} {break}
    set line [split $line]
    set TaggedTokens {}
    foreach token $line {
      if {[string length $token]} {lappend TaggedTokens $token}
    }
    foreach id $constituents TaggedToken $TaggedTokens {
      set token [tip_GetAnnotation $doc $id]
      # get pos tag from tagger output
      if {[regexp {.+/([^/]+)} $TaggedToken match tag]} {
        # create a new attribute
        set attribute [tip_CreateAttribute pos \
                          [tip_CreateAttributeValue GDM_STRING $tag]]
        # add it to the current token annotation
        set token [tip_PutAttribute $token $attribute]
        # add the annotation back to the doc
        tip_AddAnnotation $doc $token
      }
    }
  }
} ;# creole_read_HBrill_file

## Procedure: creole_HBrill_Initialize
#   Use this function in order to do some initialization. This function
#   will be called just before creole_HBrill in the following
#   situations:
#    *) If the user has opened a whole Collection, this function will be
#       called just before the first document in Collection gets
# proccessed with creole_HBrill
#    *) If the user has opened a single Document, this function will be
#       called
#       just before calling creole_HBrill
#   Please, refer to the Ellogon's Programming Manual for more
#   information on this function...
proc creole_HBrill_Initialize { col doc args } {
  global creole_HBrill_home
  ## Do some initilization here...
} ;# creole_HBrill_Initialize

## Procedure: creole_HBrill_Finish
#   Use this function in order to do some Clean-Up. This function will
#   be called after all calls to creole_HBrill in the following
#   situations:
#    *) If the user has opened a whole Collection, this function will be
#       called just after the last document in Collection gets
#        proccessed with creole_HBrill
#    *) If the user has opened a single Document, this function will be
#       called just after calling creole_HBrill
#   Please, refer to the Ellogon's Programming Manual for more
#   information on this function...
proc creole_HBrill_Finish { col doc args } {
  global creole_HBrill_home
  ## Clean-Up...
} ;# creole_HBrill_Finish

#
# End of File
#
```

Figure 4.2 The Source Code

In the source file, we do nothing in the Initialization and Finish procedures. In the main procedure we try to open the four files which are given as parameters to the Component. If any errors occur we try to handle them. The handling of the errors that might occur, is simply to issue an error message and terminate there the execution of the program. This is done in the lines which immediately follow those comments

```
## Create temporary file names...

## The first four arguments are the needed files...
```

If everything goes well, that is all files open without any errors occurring, we open the file for output, as indicated by the lines

```
# open file to use as input to external process
  if {[catch {open $tmp_dump_fname w} tmp_dump]} {
      error "Cannot open $tmp_dump_fname: $tmp_dump"
  }
```

As you can, again if an error occurs we issue an error message and terminate the program.

Now, if no error has occurred, we call the external program `tagger` with those four files as arguments and write to the output file the output of this external programm

Then we read the output file with the aid of the procedure

```
creole_read_HBrill_file
```

and then we close all the unnecessary files.

The `creole_read_HBrill_file` procedure, tries to convert the information in the output file into legitimate *Ellogon* annotations, using similar procedures from the *Ellogon API* as the ones mentioned in the previous chapter.

This procedure begins by saving all the `sentence` annotations in the `sentenceAnns` variable, with the aid of the

```
tip_SelectAnnotations
```

procedure, which takes as arguments the document, the type of the annotation and a list of constraints which are empty in our case. If there are actually no such annotations, an error is issued, as indicated by the lines

```
if {!$items} {
  error "creole_HBrill: No Sentence Annotations Found!"
}
```

If there are some annotations and no error occurs, we iterate through all the annotations, *i.e.* all sentences, as shown from the lines which follow the statement

```
foreach sentence $SentenceAnns {
```

In this iteration we read each line of the output file and we try to identify which tokens have been indeed tagged with a part of speech and which not. For all the tagged tokens, we create a new annotation that contains the information about the part of speech.

In essence, this is how we import external modules into *Ellogon* components. We have of course to know what input they need, what their output is and how to handle it.

Chapter 5

5 Further Information about the *Ellogon API*

Ellogon has its own site which we invite you to visit at

http://www.iit.demokritos.gr/skel/Ellogon/

This site contains, apart from this manual, also the *Users' Guide to Ellogon* and the *Ellogon's Components Specification*.

Furthermore, you can get more help about the *Ellogon API* if you click on the "Help Contents" of the "Help" menu in the *Ellogon* interface.