xxxLanguagePattern.xml - defining how to analyze a language

Introduction

If you want to use the Universal Analyzer to analyze a language, or you want to use the **Universal Importer** tool to import data generated by an external analyzer, in addition to defining the objects that can exist in a given language (see Defining a new language), you also need to describe how to actually analyze the language. To define how to analyze a language, you need to define the following elements in the **xxxLanguagePattern.xml** file:

- Where the language starts/stops in a file; useful when several languages are mixed together in a file
- How comments are defined
- How strings are defined
- All the reserved keywords
- All the types of objects and the patterns used to find these objects in the source code
- Embedded SQL (if it exists) start and end tags

The main points are:

- Rules for pattern matching are regular expressions
- The file containing these rules is only used when the source code is analyzed. It describes the syntax of the objects and their properties in a simple and straightforward manner. Executables like CAST Enlighten do not need this file. This file is not stored in the CAST Analysis Service. A copy of this file must be saved at a safe location.
- The content of this file is related to the content of the xxxMetaModel.xml file. Use only objects, categories and properties already defined in this file.

If you want to retrieve comments, you must define the pattern for matching comments.

xxxLanguagePattern.xml

Location

For each language supported by the Universal Analyzer, you must add a file that describes how to analyze the language. The file name and location must conform to the following points:

- The name of the file is composed of the name of the language (name of the category as defined in the metamodel) followed by "**LanguagePattern.xml**" (such as "**PHPLanguagePattern.xml**").
- The language pattern file must be located in the language package directory.

Please note that if this is not respected, the files will not be analyzed.

Structure

When the name is followed by the character '*', '+' or '?', this means that the element can occur respectively 0 to n times, 1 to n times, and 0 or 1 times. Otherwise the element is mandatory.

When the value is a regular expression (like for matching patterns) any character that is a special character in a regular expression must be preceded by an escape character, e.g.:

```
'?'
```

becomes

```
'\?'
```

If the regular expression contains characters that are not accepted in XML or will make the document incorrect, the value of the XML element must be contained in a CDATA tag.

Main elements

Element	Child Elements		Value	Description	Sample
languagePattern	begin, end, escape, comment, string, keyword, "object-type"		-	The root of the document.	
	begin?		RegExp	Contains the starting language characters.	`<?` `<![CDATA[<\?]]>`
	end?		RegExp	Contains the ending language characters.	`?>` `<![CDATA[\?>]]>`
	escape?		char	The escape character for all the elements of the language.	`<![CDATA[\]]>`
	comment*	begin, end, nested, multiline	-	Description comments	
		Begin	RegExp	Beginning of the comment	`<![CDATA[#]]>`
		End	RegExp	End of the comment	`<![CDATA[\r\n]]>`
		nested?	boolean	Can the comment be nested? For example, '/*' C++ comments can't be nested. Default value is false.	
		multiline?	boolean	Can the comment spread over several lines or not. Default value is false.	
	string*	begin, end, escape	-	Describe a string - used (amongst other things) to handle word based metrics, such as Copy/Paste and Halstead metrics	
		begin	RegExp	Beginning of the string	`<![CDATA["]]>`
		end	RegExp	End of the string	`<![CDATA["]]>`
		escape	char	Escape character for strings.	

	keyword*		String - used (amongst other things) to handle word based metrics, such as Copy/Paste and Halstead metrics	A reserved keyword of the language.	protected, interface Note that using Regular Expressions is not recommended in the keyword section. Note that the following characters must be escaped using a backslash: `() []?*-+` For example, if you need to add **list()** it should be entered as follows (note the escaping on the brackets): <keyword>list\(\)</keyword>

	operator*		RegExp – used to handle word based metrics, such as Copy/Paste and Halstead metrics	Allows a token to be specified as an arithmetical and logical operator.
				
				For example, for C++:
				
				`+, -, *, /, %`
				`<<, >>, <<=, >>=`
				`++, --`
				`<, >, <=, >=, !=, ==`
				`&, |, ~, ^`
				`&&, ||,  !`
					`<operator>`
					`<![CDATA[[!-/:-@\[-\_]]]>`
					`</operator>`

	numeric		RegExp - used to handle word based metrics, such as Copy/Paste and Halstead metrics	Allows a token to be specified as numeric literal.	`<numeric>`
					`<![CDATA[[[0-9]+]]>`
					`</numeric>`

	identifier		RegExp – used to handle word based metrics, such as Halstead metrics	Allows a token to be specified as an identifier.	`<identifier>`
					`<![CDATA[([A-Za-z]|"_")`
					`([0-9]|[A-Za-z]`
					`|"_"|"@")*]]>`
					`</identifier>`

	Types?	"object-type"	See table below in **Object type elements**		
	Links?	"link-type"			Links ?
	"link-type"	pattern, callee	-	Patterns for link recognition.	<inheritLink> <pattern> <callee> </inheritLink>

		pattern	RegExp	Pattern used to find the anchor of the link.	`<pattern>`
					`extends([_]|[\r\n]|`
					`[\t])+`
					`</pattern>`

	callee	backward	RegExp	Pattern used to find the callee of the link.	`<callee>`
					`<![CDATA[[:word:]]`
					`[[:word:]]*]]>`
					`</callee>`

		backward?	empty	Specifies when a callee pattern should be 'backward' matched.	Empty element.

Object type elements

Element	Child Elements		Value		Description	Sample (PHP)
header	pattern, begin, end		-		Patterns for object header recognition.	<header><pattern>… <begin>… <end>… </header>

	pattern		Reg Exp		Pattern used to find the anchor of the header.	`<pattern>`
						`[fF][uU][nN][cC]`
						`[tT][iI]`
						`[oO][nN]([ ]|`
						`[\r\n]|[\t])+]`
						`</pattern>`

	begin?		Reg Exp		Beginning of the header	`<begin>`
						`public:([_]|`
						`[\r\n]|[\t])+]`
						`</begin>`

	end?		Reg Exp		End of the header	`<end>`
						`\(.*\)([_]|`
						`[\r\n]|[\t])+]`
						`</end>`

category.property	pattern, value, backward				Pattern used for object property recognition.	<identification.name><pattern>… <value>… <backward> </identification.name>

	pattern		Reg Exp		Pattern used to find the anchor of the property.	`<pattern>`
						`[fF][uU][nN][cC]`
						`[tT][iI][oO]`
						`[nN]([ ]|[\r\n]|`
						`[\t])+]`
						`</pattern>`

	value	backward		Reg Exp	The Value of the property	`<value>`
						`<![CDATA[[[:`
						`word:]]`
						`[[:word:]]*]]>`
						`</value>`

	backward	empty			Specifies when a property pattern should be 'backward' matched.	Empty element: <backward/>
endwithoutbody	-		Reg Exp		Used to determine the end of an object which doesn't have a body	<endwithoutbody>; </endwithoutbody>
body	begin, end, nested, noendpattern		-		Pattern for object body recognition	<body> <begin>… <end>…<nested>… </noendpattern> </body>

	begin		Reg Exp		Pattern for the beginning of the body of the object.	`<begin>`
						`<![CDATA[{]]>`
						`</begin>`

	end		Reg Exp		Pattern for the end of the body of the object.	<end> <![CDATA[}]]> </end>
	nested?		boolean		Can the character for the beginning and end of the body be found inside the body? Default value is 'true'.	<nested>false</nested>
	noendpattern	empty	-		Use this element to indicate to the Universal Analyzer that an object does not have an end tag. You can also use this at type level.	Empty element: <noendpattern/>
noendpattern	empty		-		Use this element to indicate to the Universal Analyzer that an object does not have an end tag. You can also use this at body level	Empty element: <noendpattern/>

Note about the <backward/> element

Property values or link callees can be bi-directionally matched. The default match is 'forward', but when values are located before the pattern, you may use the "backward" element to reverse the search direction.

(Backward Properties) << "Object Type Pattern" >> (Forward Properties)

Example:

There may be a function definition in a language 'lng', whereas your code contains the following statement:

```
lng_my_function function
\{
\}
```

When configuring your 'lng' language, specify the search direction of the property 'identification.name' by adding the '**backward'** element so that the function name will be searched before the matched object type pattern "function":

```
<lngFunction>
<header> ...</header>
<identification.name>
<pattern>function</pattern>
<value>
<![CDATA[([a-z])+]]>
<backward/>
</value>
</identification.name>
<body> ...</body>
</lngFunction>
```

Objects with no end pattern and siblings that are recognized using <backward/>

Please see Appendix - Special Cases for more information about this.

Embedded SQL support

If the language you want to analyze includes embedded SQL (i.e. calls to server-side database objects), then you need to tell the Universal Analyzer how to identify the embedded calls. This can be done using a specific tag as follows:

```
<esql>
<begin><![CDATA[BEGIN_ESQL]]></begin>
<end><![CDATA[END_ESQL]]></end>
</esql>
```

Modify the [BEGIN_ESQL] and [END_ESQL] fields to reflect the start and end tags of your embedded SQL code. You can add as many <begin> and <end> tags as necessary, for example:

```
<esql>
<begin><![CDATA[BEGIN_ESQL_1]]></begin>
<end><![CDATA[END_ESQL_1]]></end>
</esql>
<esql>
<begin><![CDATA[BEGIN_ESQL_2]]></begin>
<end><![CDATA[END_ESQL_2]]></end>
</esql>
```

Please make sure you also modify the corresponding **xxxMetaModel.xml** file in your language package so that each object that needs to be searched for embedded SQL inherits from the ESQLSearchable category. You can find out more about this in the section **Embedded SQL support** in the page Defining a new language.