
HRC Language Reference

26 April 2007

This version:

take5.be5: 26 April 2007
(Available as HTML, PDF, DocBook)

Previous versions:

take5.beta4: 28 April 2005
take5.beta4(draft): 19 February 2005
take5.beta3: 30 January 2004
take5.beta2: 12 September 2003
take5.beta1: 30 March 2003
take5.alpha3: 1 March 2003
take5.alpha2: 30 January 2003

Author:

Igor Russkih <irusskih at gmail.com>

Author:

Anatoly Technonik <tehtonik at gmail.com>

Copyright © 2003, 2004, 2005, 2006, 2007 Igor Russkih (Cail Lomecb)

Abstract

This reference describes HRC language, used in Colorer-take5 Library to define and represent syntax and lexical structure of various programming languages. These syntax definitions are used by library to parse and colorize text in editors and other software.

Table of Contents

1. Introduction	2
2. Basics	3
2.1. File Types	4
2.2. Namespaces	8
3. Scheme syntax	9
3.1. Keyword lists	11
3.2. Regular Expressions	12
3.3. Block context switch	13
3.4. Scheme boundaries and priority	14
4. Inter-scheme links	15
4.1. Inheritance	15
4.2. Scheme substitutions	16
5. HRC Language Features and Conventions	16
5.1. Elements naming	16
5.2. Default package feature	16
5.3. Coding Recommendations	17
A. Regular Expressions syntax	18
1. Introduction	18
2. Syntax	18
3. Metacharacters	18
4. Extended metacharacters	19
5. Operators	20
6. Extended operators	21
7. Examples	21
B. Format of catalog.xml file	21
C. Format of HRD color schemes	23
D. XML Schema for HRC Language	25
E. History of the changes	31
References	32

1. Introduction

HRC is a script language which describes text parsing process to produce syntax highlighting. It is *XML-based* language with its own XML vocabulary and structure. *HRC* is designed to make the process of describing structures of programming languages most flexible and efficient.

Looking back to early 1999, *HRC* had simple XML-like structure describing several common language constructions. Since then it evolved into very powerful way of describing complex relations between different languages and syntax contexts. *HRC* is a full-fledged "XML application" and that means *HRC* definitions can be automatically generated from XML descriptions in other languages and converted to other formats

through XSLT templates or other means.

HRC uses *Regular Expressions* to achieve flexible recognition of text elements, lexemes and tokens. Still Regular Expressions (*RE*) are able to recognise only a limited set of syntax constructions when it is often necessary to describe more complex structures. Therefore HRC uses special construct named "*scheme*" to define behaviour of more powerful recursive set of languages (context free). Such schemes in combination with RE make HRC strong declarative language.

2. Basics

HRC describes and stores syntax rules for numerous languages. All language definitions are divided into two parts: Prototypes are used to detect correct language type

- *informal part* includes different non-syntax specific properties of a language: name, short description, common file extensions and autodetection rules. Informal part is also called *language prototype*.
- *formal part* contains actual definition of target language rules in terms of syntax and semantics. It is referenced as *language type*.

tions are divided into two parts: Prototypes are used to detect correct language type that should be applied to a file, they define some application-dependent properties and other useful information about languages. Because prototypes are separated from real language definitions, full type loading occurs only when language is correctly matched or requested by user. This guarantees fast library bootstrap. Prototype definitions grouped into one file allow users to get a quick overview of the languages supported by the library.

Structure. Each HRC file contains either several language prototypes or one language type. XML content starts with root `<hrc>` element, which contains all other HRC definitions.

Element: `<hrc>`

Root of the HRC file XML structure.

Attribute: `version`, type: `xs:NMTOKEN`

Specifies version of HRC language. For example, 'take5' for Colorer-take5.

Content:

Element: `annotation`

Defines formal documentation for the HRC language elements.

Element: `prototype`

Defines prototype of single target programming language.

Element: `package`

Defines prototype of the defined file type, but use this type as an internal hidden package structure.

Element: type

Language container, used to store all parser specific information.

Every bit of HRC is either XML element or attribute. You can find formal definition of the HRC XML syntax in Appendix D, *XML Schema for HRC Language*. For instance, all HRC files start with the syntax similar to following: Each element in HRC can be

Example 1. Common HRC file

```
<?xml version="1.0"?>
<!DOCTYPE hrc PUBLIC "-//Cail Lomecb//DTD Colorer HRC take5//EN"
  "http://colorer.sf.net/2003/hrc.dtd">
<hrc version="take5" xmlns="http://colorer.sf.net/2003/hrc"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://colorer.sf.net/2003/hrc
                      http://colorer.sf.net/2003/hrc.xsd">

  <annotation>
    <documentation>
      your documentation...
    </documentation>
  </annotation>

  your definitions...

</hrc>
```

all HRC files start with the syntax similar to following: Each element in HRC can be documented with *XML Schema*-like `<annotation>`: Annotations can be used anywhere

Element: <annotation>

Defines formal documentation for the HRC language elements.

Content:

Element: appinfo

Formal annotation part, used for tools processing.

Element: documentation

Human documentation part.

Element: contributors

Contribute information part.

documented with *XML Schema*-like `<annotation>`: Annotations can be used anywhere in HRC file to describe and document syntax elements.

2.1. File Types

HRC rules can reuse or import definitions from each other, some languages (like HTML) may include bits of other languages (i.e. PHP), so HRC files can depend on each other for correct highlighting. Therefore HRC files are more like one big database than a bunch of separate definitions. To link them together several syntax elements are used.

2.1.1. Prototypes

Each language is identified by name and short description. This information is included in language prototype. Names are used to reference languages in HRC rules. Prototypes are usually contained in top level file *proto.hrc*, but considering flexible syntax of HRC they could be just everywhere.

Prototypes are defined by `<prototype>` elements. The example shows prototype for

Example 2. Prototype definition

```
<prototype name="cpp" group="main" description="C++">
  <location link="base/cpp.hrc"/>
  <filename>/\.(cpp|cxx|cc|hpp|h)$/i</filename>
  <firstline>/^\s*(\/* | \/\)/xi</firstline>
  <firstline>/\#include/</firstline>
  <firstline>/\#define|\#if/</firstline>
</prototype>
```

Prototypes are defined by `<prototype>` elements. The example shows prototype for "C++" language. It contains short description, information about language group and location of HRC file with formal part of syntax definition. It also includes RE to identify the language by filename extension and one or more RE to guess the language by first few lines (or several hundred bytes - depends on implementation) of file contents.

Element: `<prototype>`

Defines prototype of single target programming language. This prototype must have name, equals to real type, defined in the linked resource.

Attribute: **name**, type: **xs:NCName**

Common internal name of this language type. Must be valid XML non-qualified name.

Attribute: **description**, type: **xs:string**

User description, used to represent language in target IDE.

Attribute: **group**, type: **xs>Name**

Group of languages, this language belongs to.

Attribute: **targetNamespace**, type: **xs:anyURI**

Applicable to the XML group of languages. Specifies namespace, this HRC file describing. Allows automatically linking and combining different XML languages in HRC.

Content:

Element: **annotation**

Defines formal documentation for the HRC language elements.

Element: **location**

Points to the location of a HRC file with this language description.

Element: **filename**

Defines Regular Expression, used to identify programming language by its file name.

Element: `firstline`

Defines Regular Expression, used to identify programming language by its starting content.

Element: `parameters`

Custom parameters, used to specify additional properties of this language type.

If language is not specified explicitly library needs to detect it to start syntax highlighting process. This is the purpose of `<firstline>` and `<filename>` parameters. Each matched instance of one of these parameters adds additional weight to the language. Default amount of points added can be specified explicitly with *weight* attribute of these elements. When all weights are calculated, the first language with maximum weight is selected to highlight the file.

Element: `<filename>`

Defines Regular Expression, used to identify programming language by its file name. This can include file's extension or some more complex dependencies.

Attribute: `weight`, type: `xs:decimal`, default: 2

This attribute defines weight, added to the total language weight, when choosing one from a list of available.

Element: `<firstline>`

Defines Regular Expression, used to identify programming language by its starting content. First line can be used, or some small part of text. This entry has less default weight against filename one.

Attribute: `weight`, type: `xs:decimal`, default: 1

This attribute defines weight, added to the total language weight, when choosing one from a list of available.

If any of these two elements is used more than once, each matched instance adds specified amount to the total weight of a language.

Actual language definition can be separated from its prototype and placed into other file (or resource). In this case `<location>` element specifies where to find the definition. The file or resource specified will not be loaded until language matches and is selected for highlighting process.

Element: `<location>`

Points to the location of a HRC file with this language description. Link is a well formed URI address of the requested HRC file. This location can be relative to the current location of the parent type, or absolute (with URI schemas, supported by library). If URI schema is absent, 'file://' is assumed.

Attribute: `link`, type: `xs:anyURI`
for highlighting process.

Element: `<parameters>`

Custom parameters, used to specify additional properties of this language type. These can include different language resources (icons, templates and so on). Also these parameters could be referenced from schema declaration, this allows to customize schemes loading process.

Content:**Element: param**

Single parameter [name,value] pair.

2.1.2. Packages

Some syntax rules are common across various languages and it makes sense to define them separately and reference from other definitions. These definitions will not be visible to end users, so they can be thought of as "internal types". Such internal types are represented by <package> element: This element doesn't contain <filename> or

Element: <package>

Defines prototype of the defined file type, but use this type as an internal hidden package structure.

Attribute: name, type: xs:NCName

Common internal name of this package. Must be valid XML non-qualified name.

Attribute: description, type: xs:string

User description, used to represent package in target IDE.

Attribute: targetNamespace, type: xs:anyURI

Applicable to the XML group of languages. Specifies namespace, this HRC file describing. Allows automatically linking and combining different XML languages in HRC.

Content:**Element: annotation**

Defines formal documentation for the HRC language elements.

Element: location

Points to the location of a HRC file with this language description.

represented by <package> element: This element doesn't contain <filename> or <firstline> properties, because it doesn't directly map to any type of file or language. In everything else its behaviour is identical to <prototype> element. Packages can be found in any HRC file including *proto.hrc*. For example:

Example 3. Package definition

```
<package name="def" group="packages" description="basic
definitions">
  <location link="default.hrc"/>
</package>
<package name="regexp" group="packages" description="Regexp common
library">
  <location link="lib/regexp.hrc"/>
```

```
</package>
```

2.1.3. Types

Type is a formal definition of a language. It is normally contained in a separate file, which is referenced by `<location>` element of language prototype. `<type>` element is the starting point for parsing process, which holds syntax specific information. **Element: `<type>`**

Language container, used to store all parser specific information. These defines are used by parser to analyze and colorize target text data.

Attribute: `name`, type: `xs:NCName`

HRC Language type name.

Content:

Element: `annotation`

Defines formal documentation for the HRC language elements.

Element: `import`

External type import statement.

Element: `region`

Definition of basic syntax region - text range with assigned syntax meaning.

Element: `entity`

HRC Entity definition.

Element: `scheme`

HRC scheme is a basic unit, which represents some fixed set of lexemes, tokens and syntax regions (lexical context).

the starting point for parsing process, which holds syntax specific information. Normally, each type is defined in a separate file, which may optionally contain corresponding prototype (if there is no prototype definition in the global repository).

2.2. Namespaces

Each type defines its own name space with its elements. Each element must have unique identifier (local name) in this namespace, which is used to reference it from other elements. Within the same type all elements should be unique, but elements with the same name can belong to different types.

An element can be referenced from the other type with its fully qualified name in form of *typename:elementname*. Sometimes there are a lot of inter-type links and use of qualified names can become a tedious task. To make the job easier HRC language has `<import>` statement. It 'imports' all element names from other type into the current. There can be as many import statements as needed. Unqualified names are resolved in order of their definition.

Element: `<import>`

External type import statement. This statement imports all definitions from the

specified type into the current one, so you can use them without explicit type qualifier.

Attribute: type, type: xs:NCName

For instance, you can write to import all definitions from the 'def' type. Note, that if

```
<import type='def' />
```

For instance, you can write to import all definitions from the 'def' type. Note, that if several imported types have some identical local names, they are resolved in order of import statements, i.e. the first one is used.

3. Scheme syntax

Scheme is a generic structure of the HRC language to define syntax of programming languages. Every scheme contains various syntax elements, matched or not matched as text analysis goes on. For example, a scheme for "C++" language contains different keywords, strings, numbers, comments etc. The scheme is defined by <scheme> element.

Scheme alone is not very useful for analysis. It is much more convenient to think about text of a language to be highlighted in terms of *regions*. When schema matches a piece of text it can assign various parts of this text to different regions. Each <region> defines some meaningful part of the syntax. This part or region always has a name and sometimes a reference to its parent region (if any). When parsed, source text is described as a set of these regions with specified positions and lengths.

Next stage of the text processing associates each region with some handler. A handler, for example, can assign color and font style information to each of the regions or apply other operations to these structures.

Each region is defined using a <region> element:

Element: <region>

Definition of basic syntax region - text range with assigned syntax meaning. Later, these regions can be mapped into required color information and displayed on screen.

Attribute: name, type: xs:NCName

HRC Region name.

Attribute: parent, type: QName

Region's parent reference. If region has parent, its properties can be inherited from this one. Also region inheritance creates tree structure of HRC Regions.

Attribute: description, type: xs:string

Optional description, used to represent region's purpose and to show it to user in convenient and friendly way.

Each region is defined using a <region> element:

During parsing process each element in a scheme not only creates one or more syntax <region>s used to highlight parsed text. Resulting information also contains a recurs-

ive scheme tree showing overall text structure.

Each type may define as many schemes as needed provided that all their names are unique within the type. Scheme is defined using <scheme> element: Every type is re-

Element: <scheme>

HRC scheme is a basic unit, which represents some fixed set of lexemes, tokens and syntax regions (lexical context). Each time at any position in the text only one schema is active. Its content is applied to the current text position. When the text parsing process starts, the scheme is used whose name equals the name of the corresponding type (the base scheme of the type).

Attribute: name, type: xs:NCName

HRC Scheme name. Unique in this type scope.

Attribute: if, type: xs:NCName

Load and use this scheme's content only if parameter, to which references this attribute is truth. In other case this scheme is used as an empty one.

Attribute: unless, type: xs:NCName

Load and use this scheme's content only if parameter, to which references this attribute is not truth. In other case this scheme is used as an empty one.

Content:

Element: annotation

Defines formal documentation for the HRC language elements.

Element: regexp

Regular Expression token.

Element: block

Context switch operator.

Element: keywords

List of tokens with equal properties.

Element: inherit

Scheme inheritance construction.

unique within the type. Scheme is defined using <scheme> element: Every type is required to have one scheme called "*base scheme*" which is used as an entry point for parsing process of the type. Base scheme is named after its type, i.e. local name of the scheme is equal to the name of the type. Only internal types defined with <package> element can ignore this requirement because they are never used at the top level.

Example 4. Sample type definition

```
<type name="somelang">
  <region name="Keyword" description="This language's keyword"/>
  <scheme name="somelang">
    <keywords region="Keyword">
      <word name='word1'/><word name='word2'/>
      <word name='otherkeyword'/>
    </keywords>
    <regexp match="/other(keyword)?/i" region="Keyword"/>
  </scheme>
```

```
</type>
```

Scheme element may contain *if/unless* attributes to customize parsing process according to contents of <parameters> definitions in the type of the schema. Parameters can be flexibly changed at runtime by the means of Colorer API. This allows customizing load process and suggesting various language profiles to be chosen by user.

The following sections describe different types of syntax elements, available in the HRC language.

3.1. Keyword lists

<keywords> is the most simple HRC element used to quickly define words with similar properties and highlight them in a text.

Element: <keywords>

List of tokens with equal properties. Keywords, symbols and so on... These lists are used to make processing of many tokens faster, when it isn't required to use RE to define syntax tokens.

Attribute: ignorecase, default: yes

Match this list of tokens with case sensitive or no.

Attribute: region, type: QName

Region, assigned to this list of tokens. Each token can define its custom region.

Attribute: priority, type: priority, default: low

Priority of any token can be normal and low.

Attribute: worddiv, type: REworddiv

Class of characters, used to search words edges.

Content:

Element: word

Keyword tokens - use specified word edges.

Element: symb

Symbol tokens - ignores specified word edges.

ar properties and highlight them in a text.

Element: <word>

Keyword tokens - use specified word edges.

Attribute: name, type: xs:string

Attribute: region, type: QName

A pair of type name and valid XML name.

ar properties and highlight them in a text.

Element: <symb>

Symbol tokens - ignores specified word edges.

Attribute: name, type: xs:string

Attribute: region, type: QName

A pair of type name and valid XML name.

Each element in the list may assign its own region or use region of its parent `<keywords>` element. Symbols never check surrounding characters, while words match only if surrounded by not-word symbols. These word delimiters can be redefined with *worddiv* attribute of `<keywords>`.

3.2. Regular Expressions

Regular expression rules is a powerful and flexible way to define custom syntax structures. Each RE token can be used to create several different syntax regions (up to 16). Keep in mind, however, that the scope of every RE in Colorer is limited to one line (the only exception is `<firstline>` element matched against several lines to detect file type).

Element: `<regexp>`

Regular Expression token.

Attribute: `region`, type: QName

A pair of type name and valid XML name.

Attribute: `priority`, type: priority, default: normal

Priority of any token can be normal and low.

Attribute: `match`, type: REstring

RE syntax

Actual RE is contained within *match* attribute of `<regexp>` element. Detailed explanation of Colorer-take5 regular expressions is in Appendix A, *Regular Expressions syntax*. Each `<regexp>` can have up to 16 optional attributes named *region0*, *region1*, ... *regionf* where hexadecimal digit corresponds to the part of RE surrounded by round brackets counted from left to right. *region0* means whole sequence matched by RE (this can be changed with `\m` and `\M` RE metasymbols). The value of each attribute is a name of the syntax region used to highlight text. Regular Expression can also contain named brackets what explicitly specify corresponding syntax region in the form of `(?{name} ...)`.

Each RE definition can include references to any predefined sequence of RE code. Such references are called *entities*. Entities are defined in `<type>` element and have their own qualified namespace. To include entity's value into RE, special syntax of `%entityname;` is used.

Element: `<entity>`

HRC Entity definition. Entities are some form of macro-definitions, they lately can be used in regular expressions syntax to make them simpler. Each entity consists of Entity name and Entity content, which would be substituted into regular expression, when parser finds entity reference. Each entity can be referenced with `%entityname;` syntax.

Attribute: `name`, type: xs:NCName

HRC Entity name.

Attribute: value, type: REentity

HRC Entity value, used to substitute entity in RE string.

Each RE has a priority attribute (by default its value is *normal*). Priority is mainly used to detect errors when closing matching region. When everything within the region is already matched and parser needs to close the block, it applies rule to match closing sequence. If match fails then rule with *low* priority within the block is tested. This is explained in Section 3.4, "Scheme boundaries and priority"

3.3. Block context switch

Although regular expressions are very powerful feature, they do not allow to express some complex language constructions. This is due to general limitation of colorer's RE parser which scope is limited to a single line of text. Regular expression can't work on multiple lines of the parsed text. Often programming languages have constructions wrapped into each other unlimited number of times. This is also an area where Regular Expression do not help much.

To define more complex syntax structures and context-free grammar constructions HRC has a special element named `<block>`.

Element: <block>

Context switch operator. Used to switch currently used context into the specified one. Context is switched, if RE pattern, placed in 'start' attribute, is matches. Switched context is closed, when parser finds match of the 'end' RE.

Attribute: start, type: REstring

Regular Expression

Attribute: end, type: REstring

Regular Expression

Attribute: scheme, type: QName

A pair of type name and valid XML name.

Attribute: priority, type: priority, default: normal

Priority of any token can be normal and low.

Attribute: content-priority, type: priority, default: normal

Priority of any token can be normal and low.

Attribute: inner-region, default: no

Defines if this scheme region to be located inside of the start/end region edges. In this case all the block's regions are located outside of the scheme region. By default ("no" value) scheme region is a background region for all this block's start/end regions, and wraps them all.

Content:**Element: start, type: blockInner**

Alternative style of RE definition.

Element: end, type: blockInner

Alternative style of RE definition.

Element: <blockInner>

Alternative style of RE definition. Could be used, when RE is very complex and it is easier to use character (or CDATA) sections to define it.

Attribute: match, type: REstring

RE syntax

Each block has <start> and <end> tags, each with the RE syntax already described. Everything contained within these two marks will be highlighted as a syntax of some other <scheme>, also pointed by this element's attribute. It is also possible to paint the portions of these matched tags. Much like <regex> element - <block> can contain up to 32 region attributes - *region*, *region00*, *region01*, ... *region1f*. *region0x* corresponds to round brackets of <start> tag, *region1x* is for <end> tag brackets and *region* attribute contains a name of region to paint the whole block. So it is not necessary to define scheme for assigning region to the whole block, but since scheme is a required attribute there is a stub empty scheme you can use named *def:empty*

Using <block> element you can switch context between different highlighting schemes. This way it is possible to define a great number of different syntax combinations.

3.4. Scheme boundaries and priority

Both regular expressions and block'ed scheme switches work in the same scheme context, and tested against text in the order they defined in HRC. Any conflict between multiple possible matches is resolved according to the order of elements in HRC file. After successful RE match parse position is increased by the width of that RE. By default the width is calculated from the first matched symbol till the last inclusive. However it is possible to adjust these boundaries and shift parse position. This is done with special $\backslash m$ (redefines RE start) and $\backslash M$ (redefines RE end) metasymbols. It becomes possible to define overlapped elements, where parsing of the following element starts somewhere in the middle of the previous.

3.4.1. priority

Additional selection rules are applied in case of returning from an inner scheme to its caller scheme (via <end> tag of the <block> element for instance). Information about relative position of the <block> element can't help here to determine what to apply: <end> RE of the outer block or a next regular expression/keyword/block, defined in the inner (called) scheme. To resolve such conflict HRC defines a special attribute for <regexp> and <block> elements: *priority*. Its default value is "normal". If it is changed to "low" then Colorer does not take into account this element when resolving conflicts upon exit from inner scheme. This means that <end> tag of the outer <block> element will be used instead of the element with lowered priority (when a conflict occurs). In case of nested <block> tag, *priority* attribute relates to the <start> tag. *priority* attribute is often used in rules that highlight syntax errors.

3.4.2. content-priority

Sometimes it is required to dynamically define priority of a child scheme within a

block. With *priority* attribute it is impossible to change element's priority depending on a context from where the element is called, because the element will always have the priority specified. Instead *content-priority* attribute of a `<block>` element is used to change priority for all elements of referenced scheme.

When changed into *low* it causes all the elements of that scheme to change their priority to *low* no matter what is the value of their particular *priority* attribute.

3.4.3. inner-region

When defining scheme context switch it is possible to set a default region for content of called scheme through *region* attribute of `<block>` element. The region will be used as a "background" for all other regions defined in that scheme. It is possible to manage boundaries of this region. Normally the whole scheme's content together with contents of `<start>` and `<end>` tokens is included in this default region. Region starts where `<start>` token starts, and ends where `<end>` token ends.

Sometimes it is desirable to change this behaviour and handle `<start>` and `<end>` tokens (and all the regions they may define) outside of default region of the called scheme. This could be achieved by setting *inner-region* attribute to "yes" value. When set it tells parser to exclude start/end tokens from default region of called scheme by changing default region boundaries to begin at the end of `<start>` token and finish just before `<end>` token area.

Inner region feature could be used to implement special wrapped areas and in general can affect special background color treatment.

4. Inter-scheme links

4.1. Inheritance

Element: `<inherit>`

Scheme inheritance construction. If one scheme is inherited in another, then the latter scheme takes all the definitions from the former, as it was included directly in place of inherit operator. One scheme can't inherit another, if that scheme is already makes inheritance (even indirect) of the first one.

Attribute: `scheme`, type: QName

Inherited scheme name.

Content:

Element: `virtual`

Inheritance substitution element.

Element: `<virtual>`

Inheritance substitution element. While inheriting one scheme in another, it is possible to redefine inner inherited schemes with some others. This can be used to change inherited language behavior.

Attribute: `scheme`, type: `QName`

Redefined scheme.

Attribute: `subst-scheme`, type: `QName`

Scheme to use instead redefined one.

4.2. Scheme substitutions

5. HRC Language Features and Conventions

Although HRC itself could be used in an arbitrary way, Colorer-take5 library has a number of coding and naming conventions for consistency to make maintenance and expansion of HRC library easier. Features are implemented using special conventions Colorer library knows about and does extra processing.

5.1. Elements naming

Colorer names are case sensitive. All regions in Colorer-take5 HRC database are named with capital letter, each name-part also starts with capital letter. For instance: *StringQuote*. Any separate type or package is named in lower case and shortened if possible. So, the full name of a region is written as *def:StringQuote*.

Scheme names are context dependent and could be used with words in either case. Dash or Dot delimiter makes them more readable: `<scheme name="Comment.content">` for instance.

All HRC files are named in lower-case with possible Dash or Dot delimiters. External XML entities should be used to split complex HRC files in parts that simplifies generation of automatically derived HRC schemes. Entity files carry double *ent.hrc* extensions to distinguish them from ordinary HRC schemes.

5.2. Default package feature

Colorer-take5 defines a basic set of common syntax regions through special package named *def*. Default package simplifies support of HRC database and separates parse content and its presentation. It is located in *hrc/lib/default.hrc*. The general purpose of this file is to define a basic set of syntax regions. These regions already have assigned colors via universal HRD color mappings bundled with Colorer library. All other HRC regions should be inherited from this set to flexibly define HRD color rules and unify them across all supported syntaxes and languages. Any HRC package can explicitly import and use them or define its own syntax regions, derived from the defaults.

5.2.1. Pair construction matching

Colorer-take5 library uses HRC syntax conventions for additional processing that make editing process more intuitive. This processing includes paired construction matching and file's structure/error list outline. These features are implemented through specially defined regions. Matching paired construction consists of two special regions *def:PairStart* and *def:PairEnd* that package should define. Parsing layout for these regions should be properly wrapped in a valid recursive sequence. Using this information Colorer-take5 library provides user with ability to jump over text blocks in target language and highlight them during editing process.

5.2.2. Outliner construction

Another feature Colorer-take5 library provides is a tree of valuable syntax tokens in a text. The tree allows to quickly switch among these tokens in editor. Tokens may represent programming language's functions, procedures, or any other logical structures of the text. During parsing process these constructions are collected into a special outline container, which can present them to user in realtime or by request. Colorer-take5 editor implements two basic forms of outliner: functions and errors list. Any HRC scheme may define an element with region equal to or derived from *def:Outlined*. All elements with this region are considered to be outliner-targeted and are collected during parsing. Outliner may analyse parse tree structure to generate tree-like text outliners. Moreover, any language can provide special algorithmic support or logic to implement parsing for special outlined regions and building valid outline tree. For instance EclipseColorer editor evaluates a name of each outlined region and searches an icon with such name. If found, it uses this icon to customize outliner window items with graphic objects, not only text.

Outliner can generally be set up against any region type. It works as a kind of filter, gathering only required information from parser. This is a way Errors list works, where regions derived from *def:Error* are collected. Every HRC language uses this region to mark problems it found while parsing text.

5.3. Coding Recommendations

HRC database has a long history, during which the format, syntax and meaning of its compounds were changed to reach more logical and formal structure. As a consequence there still could be some type definitions, which are not fully comply with general HRC conventions. In general these include invalid names of packages and region/schemes. They won't be supported in their current form and will be reworked one day to become compliant with other HRC definitions.

It may seem a good point to have an *import* element in HRC, which allows to use objects from other package with unqualified names, but in general this should not be overused to avoid confusion. It is much more convenient to use fully qualified regions and scheme names to explicitly show additional package usage/intersections.

A. Regular Expressions syntax

1. Introduction

Colorer library and HRC language rely heavily upon regular expressions (RE). They allow you to create universal syntax highlighting rules in HRC. The major difference from other RE engines is that Colorer RE are all limited to one line to make text processing faster.

Regular expression consists of a set of characters. Some of these are simple, and some are special (metacharacters). All metacharacters (escapes) are divided into three categories: first - zerolength (words boundaries and so on); second - class metacharacters (`\w`, `\s`.); and the third - operators. RE operators can be applied to a single character, to block, enwrapped in brackets or into other operators. You can use round brackets to group any sequence of characters. Regular expressions in HRC Language are much like Perl regexps in their base variant. There are some differences in extended operators.

2. Syntax

All regexps must be in slashes `/.../`. After the end slash there can be modifiers: Each

- *i* - ignore symbol case
- *x* - ignore direct spaces and crlf (for comfort)
- *s* - treat regexp like single line - i.e. make '.' class include `\r\n` symbols (works only for `<firstline>` element) as all other RE can't exceed line boundary

All regexps must be in slashes `/.../`. After the end slash there can be modifiers: Each symbol in RE is sequentially compared with the target string. Everything that doesn't look like metacharacter is a simple character.

3. Metacharacters

Table A.1. Metacharacters

<code>^</code>	Match the beginning of the line
<code>\$</code>	Match the end of the line
<code>.</code>	Match any character (except <code>\r\n</code>)
<code>[...]</code>	Match any character in set
<code>[^...]</code>	Match any character that is not in set. None of RE operators works here, but you

	some metacharacters and range operator are possible: a-z stands for all alphabet chars between a and z, <i>[[ASSIGNED]-[Lu]-[Ll]]</i> - unicode classes reference. <i>-[]</i> - Class substraction. <i>&&[]</i> - Class join (can be dropped). <i>[][]</i> - Class intersection.
<i>\#</i>	The symbol '#' after slash (except a-z and 1-9)
<i> b</i>	Word break at this point
<i> B</i>	No word break at this point
<i>\xHH, \x{HHHH}</i>	<i>HH, HHHH</i> - character code (hex)
<i>\n</i>	0x10 (lf)
<i>\r</i>	0x13 (cr)
<i>\t</i>	0x09 (tab)
<i> s</i>	Whitespace character (tab/space/cr/lf)
<i> S</i>	Not whitespace
<i> w</i>	Word symbol (chars, digits, _)
<i> W</i>	Not word symbols
<i> d</i>	Digit
<i> D</i>	Not Digit
<i> u</i>	Uppercase symbol
<i> l</i>	Lowercase symbol

4. Extended metacharacters

These metacharacters are incompatible with Perl

Table A.2. Extended Metacharacters

<i> c</i>	Means 'not word' before
<i> N</i>	Reference from inside of regexp to one of its brackets. <i>N</i> - the number of brackets pair. This operator works only with non-operator symbols in a bracket.

Next operators are only available in Colorer-take5 regexp parser module, when it is compiled for Colorer library (means that Colorer regex module can be used separately):

Table A.3. Colorer-take5 Parsing Metacharacters

~	Matches for the start of parent scheme (end of <i><start></i> tag).
\m	Changes start of regexp
\M	Changes end of regexp
\yN \YN \y{name} \Y{name}	Link to the external regexp (in <i><end></i> token to <i><start></i> token param). N - required bracket pair, name - named bracket.

For more information about \m \M meaning see in Section 3.4, “Scheme boundaries and priority”.

5. Operators

Operators can't be used without some preceding character sequence. Each operator must be applied to the appropriate character, metacharacter, or their combination enclosed in brackets.

Table A.4. Operators

()	Group and remember characters for later use.
(? <i>{name}</i>)	Group and remember characters using named group.
(? <i>{}</i>) or (? <i>:</i>)	Group characters, but don't remember (unnamed group).
(? <i>{}</i>)	Group and remember characters using unnamed uncounted group.
	Alternative. Match previous or next pattern.
*	Match preceding pattern 0 or more times.
+	Match preceding pattern 1 or more times.
?	Match preceding pattern 0 or 1 time.
{ <i>n</i> }	Repeat <i>n</i> times.
{ <i>n</i> ,}	Repeat <i>n</i> or more times.
{ <i>n</i> , <i>m</i> }	Repeat from <i>n</i> to <i>m</i> times.

Question sign ? after operator makes it nongreedy. For example * operator becomes

nongreedy if placing `*?` Greedy operator tries to eat as many chars in string as possible. Nongreedy takes minimum.

6. Extended operators

Table A.5. Extended Operators

<code>?#N</code>	Look-behind. N - symbol number to look behind.
<code>?~N</code>	Negative look-behind.
<code>?=</code>	Look-ahead.
<code>?!</code>	Negative Look-ahead.

Note, that two last operators exist in Perl - in form of `(?=foobar)`. But colorer uses syntax `(foobar)?=`

7. Examples

Example A.1. RE examples

<code>/foobar/</code>	will match "foobar", "foobar barfoo"
<code>/FOO bar /ix</code>	will match "foobar" "FOOBAR" "foobar and two other foos"
<code>/(foo)?bar/</code>	will match "foobar", "bar"
<code>^foobar\$/</code>	will match <code>_only_</code> with "foobar"
<code>/([d\.])/</code>	will match any number
<code>/(foo bar)+/</code>	will match "foofoofoobarfoobar", "bar"
<code>/[obar]+r/</code>	will match "foobar", "for", "far"

B. Format of catalog.xml file

Catalog for Colorer Library resources is a convenient way to centralize maintenance and development of all Colorer features. This catalog is stored in `catalog.xml` file and mapped into the ParserFactory class. Catalog contains information about all installed

HRC modules, error logging configuration and listing of available HRD sets.

Element: <catalog>

Describes all available Colorer Library resources.

Content:

Element: hrc-sets

Lists all installed root locations of HRC codes.

Element: hrd-sets

Lists all available HRD sets.

Element: <hrc-sets>

Lists all installed root locations of HRC codes. These locations are loaded when HRC bases are created.

Attribute: log-location, type: xs:string

Path to the default library log file. If missed, there is no logging.

Content:

Element: location

Single resource location.

Element: <hrd-sets>

Lists all available HRD sets. Each HRD Entry describes single color scheme, used to represent colored text. Note, that one Entry

Content:

Element: hrd, type: hrd-entry

Describes one HRD properties set.

Element: <hrd-entry>

Describes one HRD properties set.

Attribute: class, type: xs:NMTOKEN

HRD class. Currently available 'console', 'rgb' and 'text' classes.

Attribute: name, type: xs:NMTOKEN

Internal name of this set, used to referring from executable codes.

Attribute: description, type: xs:string

User-friendly description of this HRD set.

Content:

Element: location

Single resource location.

Element: <location>

Single resource location. Path can be relative to the catalog location, or absolute URI with or without URI schema specification.

Attribute: link, type: xs:string

```
<schema targetNamespace="http://colorer.sf.net/2003/catalog"
elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <element name="catalog" type="catalog"/>
```

```

<complexType name="catalog">
  <sequence>
    <element name="hrc-sets" type="hrc-sets"/>
    <element name="hrd-sets" type="hrd-sets"/>
  </sequence>
</complexType>

<complexType name="hrc-sets">
  <sequence>
    <element name="location" type="location" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="log-location" type="xs:string">
</attribute>
</complexType>

<complexType name="hrd-sets">
  <sequence>
    <element name="hrd" type="hrd-entry" minOccurs="0 "
maxOccurs="unbounded"/>
  </sequence>
</complexType>

<complexType name="hrd-entry">
  <sequence>
    <element name="location" type="location" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="class" type="xs:NMTOKEN" use="required">
</attribute>
  <attribute name="name" type="xs:NMTOKEN" use="required">
</attribute>
  <attribute name="description" type="xs:string">
</attribute>
</complexType>

<complexType name="location">
  <attribute name="link" type="xs:string" use="required"/>
</complexType>
</schema>

```

C. Format of HRD color schemes

HRD files used to assign some editor-specific properties to each HRC Region. Usually these include color and style information. HRD file is a list of entries each describing one HRC Region.

Element: <hrd>

List of assigns between regions and their external properties. These properties commonly include text decoration parameters, such as color, style, font and so on... Global color layering model can be chosen by the target application, depending on its text presentation style, features and requirements. In general, all transparent colors inherit color value from its parent schema fill color. If the current schema is a top-level, default fore- and back-ground colors are used. Default Colors can be stored in HRD, using standard default region 'def:Text', or can be requested by application from the GUI environment. Note that color properties are requested from Region's parent (in HRC structure) if this region is not declared in HRD. However if region was declared but misses some properties, they are requested from underlying schema fill region which

is determined in runtime.

Content:

Element: documentation

Human documentation part

Element: assign

Single entry, describes region's properties.

Element: <assign>

Single entry, describes region's properties. If an entry is specified more than one time, then the latest definition is used. This allows several HRD files to be processed to complete color description of target HRC regions.

Attribute: name, type: region-name

Full qualified region name (a pair [type:name]). Note, that if region has no HRD properties associations, it inherits properties from its parent. If any of its ancestors has no assigned properties, region visualization must be skipped (it becomes fully transparent).

Attribute: fore, type: color

Foreground color. If missed, transparent color assumed.

Attribute: back, type: color

Background color. If missed, transparent color assumed.

Attribute: style, type: style

Style bits (bold, italic, underline).

Attribute: stext, type: xs:string

Text prefix mapping (foreground).

Attribute: etext, type: xs:string

Text prefix mapping (background).

Attribute: sback, type: xs:string

Text Suffix mapping (foreground).

Attribute: eback, type: xs:string

Text Suffix mapping (background).

It is possible to maintain different HRD files for different languages, or to compile them into one single HRD file. The former allows you to distribute recommended settings with each language, while the latter to unify modification and storage of changed HRD settings within provided UI.

```
<schema targetNamespace="http://colorer.sf.net/2003/hrd"
elementFormDefault="qualified"
xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <element name="hrd" type="hrd"/>

  <complexType name="hrd">
    <sequence>
      <element name="documentation" type="documentation"
minOccurs="0"/>
      <sequence minOccurs="0" maxOccurs="unbounded">
        <element name="assign" type="assign"/>
      </sequence>
    </sequence>
  </complexType>
</schema>
```

```

    </sequence>
  </complexType>

  <complexType name="documentation" mixed="true">
    <sequence minOccurs="0" maxOccurs="unbounded">
      <any namespace="##other" processContents="skip"/>
    </sequence>
  </complexType>

  <complexType name="assign">
    <attribute name="name" use="required" type="region-name">
    </attribute>
    <attribute name="fore" type="color">
    </attribute>
    <attribute name="back" type="color">
    </attribute>
    <attribute name="style" type="style">
    </attribute>
    <attribute name="stext" type="xs:string">
    </attribute>
    <attribute name="etext" type="xs:string">
    </attribute>
    <attribute name="sback" type="xs:string">
    </attribute>
    <attribute name="eback" type="xs:string">
    </attribute>
  </complexType>

  <simpleType name="region-name">
    <restriction base="xs:string">
      <pattern value="\i\c*\:\i\c*" />
    </restriction>
  </simpleType>

  <simpleType name="color">
    <restriction base="xs:string">
      <pattern value="#?[\dA-Fa-f]{1,6}" />
    </restriction>
  </simpleType>

  <simpleType name="style">
    <restriction base="xs:string">
      <pattern value="\d" />
    </restriction>
  </simpleType>
</schema>

```

D. XML Schema for HRC Language

This XML Schema was automatically generated from the original *hrc.xsd* source, available at <http://colorer.sf.net/2003/hrc.xsd>. All comments and documentation tags were stripped to achieve more compact format. To use this schema for other than informational purposes use up-to-date version available from the link above.

```

<schema targetNamespace="http://colorer.sf.net/2003/hrc"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <simpleType name="REstring">
    <restriction base="xs:string">

```

```

    <whiteSpace value="collapse"/>
    <pattern value="/.*/[ix]*/"/>
  </restriction>
</simpleType>

<simpleType name="REworddiv">
  <restriction base="xs:string">
    <whiteSpace value="collapse"/>
    <pattern value="\[.*\]|%.*;"/>
  </restriction>
</simpleType>

<simpleType name="REentity">
  <restriction base="xs:string">
    <whiteSpace value="collapse"/>
    <pattern value=".*"/>
  </restriction>
</simpleType>

<simpleType name="REstring-or-null">
  <union memberTypes="REstring">
    <simpleType>
      <restriction base="xs:string">
        <enumeration value=""/>
      </restriction>
    </simpleType>
  </union>
</simpleType>

<simpleType name="QName">
  <restriction base="xs:QName">
    <pattern value="(\i\c*:)?\i\c*/"/>
  </restriction>
</simpleType>

<attributeGroup name="regionX">
  <attribute name="region" type="QName"/>
  <attribute name="region0" type="QName"/>
  <attribute name="region1" type="QName"/>
  <attribute name="region2" type="QName"/>
  <attribute name="region3" type="QName"/>
  <attribute name="region4" type="QName"/>
  <attribute name="region5" type="QName"/>
  <attribute name="region6" type="QName"/>
  <attribute name="region7" type="QName"/>
  <attribute name="region8" type="QName"/>
  <attribute name="region9" type="QName"/>
  <attribute name="regiona" type="QName"/>
  <attribute name="regionb" type="QName"/>
  <attribute name="regionc" type="QName"/>
  <attribute name="regiond" type="QName"/>
  <attribute name="regione" type="QName"/>
  <attribute name="regionf" type="QName"/>
</attributeGroup>

<element name="hrc" type="hrc"/>

<complexType name="hrc">
  <sequence>
    <element name="annotation" type="annotation" minOccurs="0"/>
    <choice minOccurs="0" maxOccurs="unbounded">
      <element name="prototype" type="prototype"/>
      <element name="package" type="package"/>
      <element name="type" type="type"/>
    </choice>
  </sequence>

```

```

    </sequence>
    <attribute name="version" type="xs:NMTOKEN" use="required">
  </attribute>
</complexType>

<complexType name="annotation">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="appinfo">
      <complexType mixed="true">
        <sequence minOccurs="0" maxOccurs="unbounded">
          <any namespace="##other" processContents="lax"/>
        </sequence>
      </complexType>
    </element>
    <element name="documentation">
      <complexType mixed="true">
        <sequence minOccurs="0" maxOccurs="unbounded">
          <any namespace="##other" processContents="skip"/>
        </sequence>
      </complexType>
    </element>
    <element name="contributors">
      <complexType mixed="true">
        <sequence minOccurs="0" maxOccurs="unbounded">
          <any namespace="##other" processContents="lax"/>
        </sequence>
      </complexType>
    </element>
  </choice>
</complexType>

<complexType name="package">
  <sequence>
    <element name="annotation" type="annotation" minOccurs="0"/>
    <element name="location" type="location" minOccurs="0"/>
  </sequence>
  <attribute name="name" type="xs:NCName" use="required">
</attribute>
  <attribute name="description" type="xs:string" use="required">
</attribute>
  <attribute name="targetNamespace" type="xs:anyURI">
</attribute>
</complexType>

<complexType name="prototype">
  <sequence>
    <element name="annotation" type="annotation" minOccurs="0"/>
    <element name="location" type="location" minOccurs="0"/>
    <element name="filename" type="filename" minOccurs="0"
maxOccurs="unbounded"/>
    <element name="firstline" type="firstline" minOccurs="0"
maxOccurs="unbounded"/>
    <element name="parameters" type="parameters" minOccurs="0"/>
  </sequence>
  <attribute name="name" type="xs:NCName" use="required">
</attribute>
  <attribute name="description" type="xs:string" use="required">
</attribute>
  <attribute name="group" type="xs:Name">
</attribute>
  <attribute name="targetNamespace" type="xs:anyURI">
</attribute>
</complexType>

<complexType name="location">

```

```

    <attribute name="link" type="xs:anyURI" use="required"/>
  </complexType>

  <complexType name="filename">
    <simpleContent>
      <extension base="REstring">
        <attribute name="weight" type="xs:decimal" default="2">
          </attribute>
        </extension>
      </simpleContent>
    </complexType>

  <complexType name="firstline">
    <simpleContent>
      <extension base="REstring">
        <attribute name="weight" type="xs:decimal" default="1">
          </attribute>
        </extension>
      </simpleContent>
    </complexType>

  <complexType name="parameters">
    <sequence minOccurs="0" maxOccurs="unbounded">
      <element name="param">
        <complexType>
          <attribute name="name" type="xs:string" use="required"/>
          <attribute name="value" type="xs:string" use="required"/>
          <attribute name="description" type="xs:string"
use="optional"/>
        </complexType>
      </element>
    </sequence>
  </complexType>

  <complexType name="type">
    <choice minOccurs="0" maxOccurs="unbounded">
      <element name="annotation" type="annotation"/>
      <element name="import" type="import"/>
      <element name="region" type="region"/>
      <element name="entity" type="entity"/>
      <element name="scheme" type="scheme"/>
    </choice>
    <attribute name="name" type="xs:NCName" use="required">
      </attribute>
  </complexType>

  <complexType name="scheme">
    <sequence>
      <element name="annotation" type="annotation" minOccurs="0"/>
      <choice minOccurs="0" maxOccurs="unbounded">
        <element name="regexp" type="regexp"/>
        <element name="block" type="block"/>
        <element name="keywords" type="keywords"/>
        <element name="inherit" type="inherit"/>
      </choice>
    </sequence>
    <attribute name="name" type="xs:NCName" use="required">
      </attribute>
    <attribute name="if" type="xs:NCName" use="optional">
      </attribute>
    <attribute name="unless" type="xs:NCName" use="optional">
      </attribute>
  </complexType>

  <complexType name="import">

```

```

    <attribute name="type" type="xs:NCName" use="required"/>
  </complexType>

  <complexType name="entity">
    <attribute name="name" type="xs:NCName" use="required">
    </attribute>
    <attribute name="value" type="REntity" use="required">
    </attribute>
  </complexType>

  <complexType name="region">
    <attribute name="name" type="xs:NCName" use="required">
    </attribute>
    <attribute name="parent" type="QName">
    </attribute>
    <attribute name="description" type="xs:string">
    </attribute>
  </complexType>

  <complexType name="regexp">
    <complexContent>
      <extension base="blockInner">
        <attribute name="region" type="QName"/>
        <attribute name="priority" type="priority" default="normal"/>
      </extension>
    </complexContent>
  </complexType>

  <simpleType name="priority">
    <restriction base="xs:string">
      <enumeration value="low"/>
      <enumeration value="normal"/>
    </restriction>
  </simpleType>

  <complexType name="block">
    <sequence minOccurs="0">
      <element name="start" type="blockInner"/>
      <element name="end" type="blockInner"/>
    </sequence>
    <attribute name="start" type="REstring"/>
    <attribute name="end" type="REstring"/>
    <attribute name="scheme" type="QName" use="required"/>
    <attribute name="priority" type="priority" default="normal"/>
    <attribute name="content-priority" type="priority"
default="normal"/>
    <attribute name="inner-region" default="no">
      <simpleType>
        <restriction base="xs:string">
          <enumeration value="yes"/>
          <enumeration value="no"/>
        </restriction>
      </simpleType>
    </attribute>
    <attributeGroup ref="regionXX"/>
  </complexType>

  <attributeGroup name="regionXX">
    <attribute name="region" type="QName"/>
    <attribute name="region00" type="QName"/>
    <attribute name="region01" type="QName"/>
    <attribute name="region02" type="QName"/>
    <attribute name="region03" type="QName"/>
    <attribute name="region04" type="QName"/>
    <attribute name="region05" type="QName"/>
  </attributeGroup>

```

```

<attribute name="region06" type="QName" />
<attribute name="region07" type="QName" />
<attribute name="region08" type="QName" />
<attribute name="region09" type="QName" />
<attribute name="region0a" type="QName" />
<attribute name="region0b" type="QName" />
<attribute name="region0c" type="QName" />
<attribute name="region0d" type="QName" />
<attribute name="region0e" type="QName" />
<attribute name="region0f" type="QName" />
<attribute name="region10" type="QName" />
<attribute name="region11" type="QName" />
<attribute name="region12" type="QName" />
<attribute name="region13" type="QName" />
<attribute name="region14" type="QName" />
<attribute name="region15" type="QName" />
<attribute name="region16" type="QName" />
<attribute name="region17" type="QName" />
<attribute name="region18" type="QName" />
<attribute name="region19" type="QName" />
<attribute name="region1a" type="QName" />
<attribute name="region1b" type="QName" />
<attribute name="region1c" type="QName" />
<attribute name="region1d" type="QName" />
<attribute name="region1e" type="QName" />
<attribute name="region1f" type="QName" />
</attributeGroup>

<complexType name="blockInner">
  <simpleContent>
    <extension base="REstring">
      <attributeGroup ref="regionX"/>
      <attribute name="match" type="REstring">
    </attribute>
    </extension>
  </simpleContent>
</complexType>

<complexType name="inherit">
  <sequence>
    <element name="virtual" type="virtual" minOccurs="0"
maxOccurs="unbounded" />
  </sequence>
  <attribute name="scheme" type="QName" use="required">
  </attribute>
</complexType>

<complexType name="virtual">
  <attribute name="scheme" type="QName" use="required">
  </attribute>
  <attribute name="subst-scheme" type="QName" use="required">
  </attribute>
</complexType>

<complexType name="keywords">
  <choice minOccurs="0" maxOccurs="unbounded">
    <element name="word" type="word" />
    <element name="symb" type="symb" />
  </choice>
  <attribute name="ignorecase" default="yes">
  <simpleType>
    <restriction base="xs:string">
      <enumeration value="yes" />
      <enumeration value="no" />
    </restriction>
  </simpleType>
  </attribute>
</complexType>

```

```

    </simpleType>
  </attribute>
  <attribute name="region" type="QName">
</attribute>
  <attribute name="priority" type="priority" default="low"/>
  <attribute name="worddiv" type="REworddiv">
</attribute>
</complexType>

<complexType name="symb">
  <attribute name="name" type="xs:string" use="required"/>
  <attribute name="region" type="QName"/>
</complexType>

<complexType name="word">
  <attribute name="name" type="xs:string" use="required"/>
  <attribute name="region" type="QName"/>
</complexType>
</schema>

```

E. History of the changes

take5.be5, 26 April 2007

- Anatoly Tehtonik:

```

* disambiguation concerning language features and conventions
+ finish rewriting - file still contains some other ideas
  for improvement in comments
+ rephrasings and fixes to chapter 5 and more
+ proofreading fixes up to part 5 - coding conventions
+ rephrase regex notes and document region features of the block
* minor clarifications
core syntax -> basics
+ Igor remarks
quickly go with edit fixes till the end of 3rd chapter
rewrote keyword lists and RE descriptions
process attributes derived from extended types
spellcheck
reduce amount of text to be placed in reader's buffer to gain
  an idea of connection between scheme and region
name space clarifications (not complete yet)
consistent description of what type is
rewrap a package
story continues - expanded prototypes language (confusing, eh?) -
simplify
core syntax - mess with the rest
core syntax - two parts explanations simplified
reworded introduction
rephrased abstract
+ ids in examples and tables to fix compiler warnings
replace confusing entities with meaningful names
display element type only if it is different from element name
add angle brackets to make XML nature of HRC elements obvious
it doesn't make sense to output complexType element name twice
consistent ids also for xsd reference
generate consistent ids to be referenced from manual, add x:hrc
reference element
move 'parameters' complexType to be found by xslt for reference

```

take5.beta4, 28 April 2005

- New Section 3.4.3, “inner-region” attribute description.
- Minor HRD schema clarifications.

take5.beta4(draft), 19 February 2005

- Clarification of <regexp> and <block> regions usage.
- "Scheme boundaries and priority" explained.
- "HRC Language Coding Conventions" section was added.

References

- [XML 1.0] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, Eve Maler, editors. *Extensible Markup Language (XML) 1.0 Second Edition*. W3C (World Wide Web Consortium), 2000.
- [XSLT 1.0] James Clark, editor. *XSL Transformations (XSLT) 1.0*. W3C (World Wide Web Consortium), 1999.
- [W3C XML Schema Structures] Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn, editors. *XML Schema Part 1: Structures*. W3C (World Wide Web Consortium), 2001.
- [W3C XML Schema Datatypes] Paul V. Biron, Ashok Malhotra, editors. *XML Schema Part 2: Datatypes*. W3C (World Wide Web Consortium), 2001.