HRC Language Reference

28 April 2005

This version:
take5.beta4: 28 April 2005
(Available as HTML, PDF, DocBook)
Previous versions:
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 `<cail at nm.ru>`

Abstract

This reference defines syntax and semantics of the HRC language, used in Colorer-take5 Library to represent and describe syntax and lexical structure of target programming languages. This description is used by library to parse and colorize text in editors or other systems.

Table of Contents

1. Introduction

HRC is a script language which is describes parsing process of text files to produce syntax highlighting. It is based on XML markup, and defines its own XML vocabulary and structure. HRC language is developed to achieve most flexible and efficient process of describing programming language structures.

Started nearly in year 1999, it was a simple XML-like structure, describing some common language constructions. But later it has grown into the much more complex and powerful language with complex relations between different languages and syntax contexts.

HRC is based on Regular Expressions, which allow to achieve flexible recognition of text elements, lexemes and tokens. But Regular Expressions (RE) allows recognition of rather limited syntax constructions, and often it is needed to describe more complex languages. Because this, HRC language uses special construction, named "scheme", which allows to describe more powerful, recursive class of languages (context free) and in combination with RE brings HRC to strong declarative language.

2. Core Syntax

HRC language allows describing and storing syntax rules for numerous languages. All language descriptions are divided into two parts: **informal** part (used to describe different properties of the language, initial choose rules, service information - `prototype` element), and **formal** part, which defines syntax and semantics of the target parsed language (`type` element). Prototypes are used to determine, which type should be applied to the currently opened file, they define some internal application-dependent properties and other useful information about language. Because you may separate prototype definition from real language definition, full type loading only occurs when it is really requested by the user. Also, all the prototype definitions, collected in one initial source file, allow user to see list of languages, supported by the library and guarantee fast library's bootstrap.

Structure. Each HRC file contains declaration of one or more prototypes or one language type. Root XML content starts with `hrc` element, which contains all other HRC definitions. Each HRC language object is defined using XML elements and attributes.

Element Name: `hrc`, type: `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`, type: `annotation`

Defines formal documentation for the HRC language elements.

Element: `prototype`, type: `prototype`

Defines prototype of single target programming language.

Element: `package`, type: `package`

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

Element: `type`, type: `type`

Language container, used to store all parser specific information.

definitions. Each HRC language object is defined using XML elements and attributes. You can find definition of the HRC XML Syntax in Appendix D, *XML Schema for HRC Language*. For instance, mostly all HRC definitions start with the next syntax:

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>
```

Each element in HRC can be documented with XML Schema-like elements: Annota-

Element Name: `annotation`, type: `annotation`

Defines formal documentation for the HRC language elements.

Content:

Element: `appinfo`, type: `appinfo`

Formal annotation part, used for tools processing.

Element: `documentation`, type: `documentation`

Human documentation part.

Element: `contributors`, type: `contributors`

Contribute information part.

Each element in HRC can be documented with XML Schema-like elements: Annotation object can be used anywhere in the HRC context to document and describe any of the HRC elements.

2.1. File Types

Each language prototype requires definition of the language's name and description. These properties are used to determine language in context of the other language definitions and in inter-language linkage process.

2.1.1. Prototypes

Prototypes are declared with `<prototype>` elements. For instance:

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>
```

declares "C++" language and it properties. These are language's group, description, location of HRC file with language's syntax declaration; RE mask, used to identify this type by file name extension, and one or more masks, used to identify type by file's first line.

Element Name: `prototype`, type: `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`, type: `annotation`

Defines formal documentation for the HRC language elements.

Element: `location`, type: `location`

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

Element: `filename`, type: `filename`

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

Element: `firstline`, type: `firstline`

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

Element: `parameters`, type: `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.

Each language must be chosen by the library before starting syntax highlighting process. This is made with help of `firstline` and `filename` parameters. Each matched instance of one of these parameters adds some additional weight to the total language weight (choosing probability). This value is taken by default, or can be changed explicitly with `weight` attribute of these elements. When total weights of all types are evaluated, first language with maximum weight is selected to assign to the opening file.

Element Name: `filename`, type: `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 Name: `firstline`, type: `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 operators is used more than once, each its matched instance adds a specified weight to the total language weight.

If real language definition is separated from the prototype and placed in other resource, it location is pointed with `location` element. Real type loading occurs when the described type will be really requested.

Element Name: `location`, type: `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`

scribed type will be really requested.

2.1.2. Packages

You can define file type with a special meaning of the internal type, which is used by other types and is not visible to user. This role is managed by the `package` element:

Element Name: `package`, type: `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`, type: `annotation`

Defines formal documentation for the HRC language elements.

Element: `location`, type: `location`

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

other types and is not visible to user. This role is managed by the `package` element: This element doesn't contain `filename` or `firstline` properties because it doesn't directly map to any file type or language. As for the rest, all its behavior is like prototype's one. For example:

Example 3. Package definition

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

type's one. For example:

2.1.3. Types

Each prototype (or package) defines its linkage with real file type, describing information, specific for the syntax parsing process. This information is stored in basic units,

called `type`s. Normally, each type must be defined in a separate file, which contains

Element Name: `type`, type: `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`, type: `annotation`

Defines formal documentation for the HRC language elements.

Element: `import`, type: `import`

External type import statement.

Element: `region`, type: `region`

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

Element: `entity`, type: `entity`

HRC Entity definition.

Element: `scheme`, type: `scheme`

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

called `type`s. Normally, each type must be defined in a separate file, which contains this type and optionally its prototype (if there is no this prototype definition in the global repository). This allows each file to be loaded only once, when required type is really requested by the user.

2.2. Namespaces

Each type defines its own namespace, where different objects are resided. Each object must have unique identifier (name) in this namespace, which is used to reference to it from other objects. Uniqueness is only required for objects of the same type. So, you can create objects of different types with the same name and there will be no conflicts. If object must be referenced from the other type, its fully qualified name is used in form of `typename:objectname`. But often there are too many inter-type links, so it is too tedious to use qualified names each time. To eliminate this, HRC language has an `import` statement. If used, it 'imports' all the object names from the imported type into the current one. There can be as much import statements, as needed. Unqualified name resolving occurs in order of their definition.

Element Name: `import`, type: `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 into the current

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

For instance, you can write to import all definitions from the 'def' type into the current one. Note, that if multiple imported types have equal local names, they would be resolved in order of import declarations.

3. Scheme syntax

Scheme is a common construction in the HRC language, used to express and describe syntax of target languages. Each scheme represents syntax context, which contains different syntax elements, matched in order of text analysis. For example, scheme for "C++" language contains different keywords, string and number tokens, comments, and others. To describe all the information, required to be highlighted, `region` objects are used. Each region defines some syntactically meaningful element. This element always has a name and sometimes a reference to its parent region (if exists). When parsed, source text is described in terms of these regions. This description contains groups of the regions with specified positions and lengths.

In the next stage of the text processing, each region is associated with some handler. For example, a handler can assign color and font style information to the each of the regions, or can produce some operations over these structures.

Each region is defined using a `region` element:

Element Name: `region`, type: `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 convient and friendly way.

Each region is defined using a `region` element:

Scheme is a common construction in the HRC language, which contains syntax definition of the described programming language. Each element in scheme, while parsing, creates one or more syntax regions which are used to highlight parsed text. Resulting parse information contains not only a list of the regions, it also contains a recursive scheme tree, which shows overall text structure.

Each type can define as much schemes, as needed, provided that all of them have names, unique in this type scope. Scheme is defined using `scheme` element: Each

Element Name: `scheme`, type: `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`, type: `annotation`

Defines formal documentation for the HRC language elements.

Element: `regexp`, type: `regexp`

Regular Expression token.

Element: `block`, type: `block`

Context switch operator.

Element: `keywords`, type: `keywords`

List of tokens with equal properties.

Element: `inherit`, type: `inherit`

Scheme inheritance construction.

names, unique in this type scope. Scheme is defined using `scheme` element: Each type must have one scheme, called "base scheme" - this scheme is required to be declared in each type. Only types, declared as `package` can ignore this requirement because they are never used as top level types. Each base scheme must have local (unqualified) name equal to its type name. Base scheme in each type is used as an entry point for parse process.

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>
```

You can customize scheme loading and overall HRC structure using if/unless attributes of the scheme element. If used, they have to reference to a common parameter declaration of this scheme's type. These parameters values could be changed within Colorer's API, this allows to customize HRC loading and suggest different language profiles to user's choice.

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

3.1. Keyword lists

This common and often used syntax construction describes a list of keywords with similar properties.

Element Name: `keywords`, type: `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`, type: `word`

Keyword tokens - use specified word edges.

Element: `symb`, type: `symb`

Symbol tokens - ignores specified word edges.

similar properties.

Element Name: `word`, type: `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.

Element Name: `symb`, type: `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 this list can define its own region assignment or derive the global one, defined in `keywords` element. Symbols never checks for characters surrounding them, while words only can matches if they are surrounded by not-word symbols. These word dividers can be redefined using `worddiv` attribute of `keywords` element.

3.2. Regular Expressions

Regular expression tokens are used to create custom syntax elements. These elements are described with RE rules, which makes them very powerful and flexible. Each RE token can be used to create a number of different syntax regions (up to 16).

Element Name: `regexp`, type: `regexp`

Regular Expression token.

Attribute: `match`, type: `REstring`

RE syntax

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

Priority of any token can be normal and low.

For details of colorer-take5 regular expressions syntax please see Appendix A, *Regular Expressions syntax*. This syntax is used in `match` attribute. Each `<regexp>` region can have a number of optional attributes - `region0, region1, ... regionf` (total 16 regions). A value of each attribute is a name of the corresponding syntax region, which will be used to highlight text. A number in attribute's name means corresponding bracket in a RE. `region0` means full matched range of the RE (this could be changed with `\m` and `\M` RE metasymbols). Regular Expression can also include named brackets in syntax of `(?{name} ... )`. In this case the name of the bracket itself is a name of corresponding syntax region.

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

Element Name: `entity`, type: `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 defined priority attribute (by default it `normal`). This means normal RE priority over enwrapped syntax elements. Enwrapped element is an element, which waits to be matched from a top-level scheme in a parse sequence. `low` priority means that such a top-level element (such a `end` attribute of a `block` element) will be matched first, if there will be choice conflict between these two.

3.3. Blocked context switch

Although regular expressions is very powerful feature, their syntax doesn't allow to express some complex language constructions. First, this is due to general limitation of colorer's RE parser - single line of text scope. This means that no regular expression can work on multiple lines of the parsed text. Often programming languages have constructions which could be wrapped into each other unlimited number of times. This is also an area, where Regular Expression would not help much.

To express much more complex syntax and allow to declare context-free grammar constructions, HRC defines a special token, named "block".

Element Name: `block`, type: `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 Name: `blockInner`, type: `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.

Each block defines its `start` and `end` tags, each with a RE syntax already described. Everything enwrapped within these two marks will be highligted as a syntax of some other `scheme`, also pointed in this element's attribute.

This means that using `block` attribute you can switch context between differen highligting schemas. This allows to define great number of syntax variations and particulars.

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 order they were defined in HRC. This means that any conflict between multiple match possibilities is resolved according to order of RE, defined in HRC file. After RE match the parse position is increased by width of that RE. By default the width is from first matched symbol till last matched. However it is possible to redefine regular expression boundaries and therefore shift somehow parse position increase. This could be done with special `\m` (redefines RE start) and `\M` (redefines RE end) metasymbols. Such a behaviour allows you to define overlapped tokens, where next token parse starts somewere in the middle of the previous. In such a case color definitions occur in the order they are parsed.

3.4.1. priority

Additional selection rules are applied to a usecase, where return occurs from an inner scheme to its caller scheme (via `end` tag of the `block` element). 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 a conflict HRC defines a special attribute for `regexp` and `block` objects: `priority`. This attribute's default value is `"normal"`. When changed to `low` it tells Colorer not to take this object into account, when resolv-

ing conflicts upon exit from inner scheme. This means that `end` tag of the outer `block` element will be used instead of the object with lowered priority (when a conflict will occur). When reviewing nested `block` tags, `priority` attribute relates to the inner object's `start` tag.

3.4.2. content-priority

Often it is required to define a behavior of an element dynamically, depending on usage context. With `priority` attribute it is impossible to change element's priority depending on called context. The element will always be the same priority. But it is possible to change whole scheme's definition priority (i.e. all it's elements priority) - using `content-priority` attribute of a `block` element.

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

3.4.3. inner-region

While defining scheme context switch there is a possibility to set a default region, used for all called scheme content. This region will be used as a "background" for all other regions, defined in that scheme. It is possible to manage boundaries this region will use while instantiation. In a normal case all the scheme's content including its `start` and `end` attributes is handled inside of this default region. Therefore the region starts where `start` token starts, and ends where `end` token ends.

Sometimes it is required to change this behaviour and handle `start` and `end` tokens (and all the regions they instantiate) outside of called scheme default region. This could be achieved with `inner-region` attribute with `"yes"` value. When defined it tells parser to include start/end regions outside of called scheme, and to change called scheme default region boundaries. In this case they start at the end of `start` token and ends 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 Name: `inherit`, type: `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`, type: `virtual`

Inheritance substitution element.

Element Name: `virtual`, type: `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. Schemes substitutions

5. HRC Language Coding Conventions

Although HRC itself could be used in an arbitrary way, Colorer-take5 library has a number of coding, naming and other conventions, which applies not to HRC syntax itself, but to the library which reads and parses output of an analizer.

5.1. Object naming

All regions in Colorer-take5 HRC database are going to be named in a same way: name starts with capital letter, each name-part also starts with capital letter. For instance: `StringQuote`. Any separate type or package is named in low-case letters with possible shortcuts, to make package name shorter. Any region in this case is addressed as `def:StringQuote`.

Scheme names are going to be context dependent and could be used in any case. Dash or Dot delimiter is often used to make them looks better: `<scheme name="Comment.content">` for instance.

All files in file system storage should be placed in low-case with possible Dash or Dot delimiters. External XML entities should be used to simplify complex HRC files structure and make more easier autogenerated HRC files creation and integration. External entities should not have `hrc` extension. They should use a `xml`, as more compliant.

5.2. Default package

Colorer-take5 defines special package type to declare general syntax regions framework, simplify HRC database support and decouple parse content and its presentation. The name of this package is `def`. It is placed in `hrc/lib/default.hrc`. The general purpose of this file is to declare a basic number of syntax regions, all other HRC regions should be inherited from this set. This allows to flexibly define HRD color highlighting rules to make them unique across all supported syntaxes and languages. Any HRC package can explicitly import and use them. Or it can declare its own syntax regions, derived from the defaults.

5.2.1. Pair construction matching

Colorer-take5 library itself (independently on HRC syntax) implements a number of extensions to make editing process more flexible. These are paired construction matching and file's structure/error list outlining. These features are implemented through the specially declared regions, which are mapped to more complex syntax region values. To declare a matching paired construction, package should instantiate regions with special `def:PairStart` and `def:PairEnd`. These regions parse layout should be properly enwrapped in a valid recursive sequence. Using this information Colorer-take5 library can make actions to provide user an 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 can provide is a possibility to build a tree of some valuable syntax tokens with possibility to quickly navigate over them in editor. These tokens can represent programming language's functions, procedures, or any other logical structure of the text. While parsing such constructions are collected into a special outline container which can represent them to user in realtime or by request. Colorer-take5 editor implements two basic forms of outliner: functions and errors list. Any programming language can instantiate token with region equals or derived from `def:Outlined`. All tokens with this region are considered to be outliner-targeted tokens and collected while parsing. Outliner can also take into account information about parse tree structure to generate tree-like text outliners. Moreover, any language can provide special algorithmic support or logic to implement special outlined regions parsing and building valid outline tree. For instance EclipseColorer editor evaluates a name of each outlined region and searches an icon with such a name. If found, it uses this icon to customize outliner window items with graphic objects, not only text.

Outliner can generally be setted up agains any region type. It works as a kind of filter, gathering from parser only required information. In such a way works Errors list, which collects regions, derived from `def:Error`. Each language uses this region to mark problems in text, it found during parsing.

5.3. Coding Recommendations

HRC database has long standing up history, many times its format, syntax and meaning were changed to reach more logic and formal structure. Because of this there are

still some file type descriptions, which are not fully comply with general HRC conventions. These should not be supported in their current form, but should be reworked to be more complient with all HRC descriptions. In general these includes invalid package naming, invalid region/schemes naming.

It is good point in HRC to have an `import` feature, which allows to use other package's objects in their unqualified names, but in general this feature should not be overused. It is much more convenient to use fully qualified regions and schemes names to explicitly show the package reusage/intersections.

A. Regular Expressions syntax

1. Introduction

All work of the Colorer library and HRC language is based on the regular expressions (RE) usage. They allow you to create universal syntax rules of highlighting in HRC.

Regular expressions consist of the 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 class is operators. RE operators can be applied to single character, to block, enwrapped in brackets or into other operators. You can use brackets to group any sequence of characters. Regular expressions in HRC Language are like Perl regexp in their base syntax. There are some differences in extended operators.

2. Syntax

All regexps must be in slashes `/.../`. After the end slash there can be the next parameters: Each symbol in RE is linearly compared with the target string. Everything, that

- `i` - ignore symbol case
- `x` - ignore direct spaces and crlf (for comfort)
- `s` - suppose, that regexp is single line - it means, than '.' class should include \r\n symbols.

meters: Each symbol in RE is linearly compared with the target string. Everything, that doesn't looks like metacharacters, means simple character.

3. Metacharacters

Table A.1. Metacharacters

^	Match the beginning of the line
$	Match the end of the line
.	Match any character (except \r\n)
[...]	Match characters in set
[^...]	Match characters not in set. All the operators are disabled here, but you can use other metacharacters and range operator: a-z means all chars from first to second (a - z), [{ASSIGNED}-[{Lu}]-[{Ll}]] - unicode classes reference. -[] - Class substraction. &&[] - Class join (can be dropped). |[] - Class intersection.
\#	Next symbol '#' after slash (except a-z and 1-9)
\b	Word break at this point
\B	No word break at this point
\xHH, \x{HHHH}	HH, HHHH - character code (hex)
\n	0x10 (lf)
\r	0x13 (cr)
\t	0x09 (tab)
\s	Whitespace character (tab/space/cr/lf)
\S	Not whitespace
\w	Word symbol (chars, digits, _)
\W	Not word symbols
\d	Digit
\D	Not Digit
\u	Uppercase symbol
\l	Lowercase symbol

4. Extended metacharacter

These metacharacters are incompatible with Perl

Table A.2. Extended Metacharacters

\c	Means 'not word' before
\N	Link inside of regexp to one of its brackets. N -

	needed brackets pair. This operator works only with non-operator symbols in a bracket.

The next operators are only available in Colorer-take5 regexp module parser, when in compiled for Colorer library:

Table A.3. Colorer-take5 Parsing Metacharacters

`~`	Matches for the start of parent scheme (end of `start` tag).
`\m`	Changes start of regexp
`\M`	Changes end of regexp
`\yN \YN \y{name} \Y{name}`	Link to the external regexp (in `end` token to `start` 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 block of their combination (enclosed with brackets).

Table A.4. Operators

`( )`	Group and remember characters to form one pattern.
`(?{name} )`	Group and remember characters into the named group.
`(?{} ) or (?: )`	Group but don't remember characters into the group (unnamed group).
`(?{} )`	Group and remember characters into the unnamed uncounted group.
`|`	Match previous or next pattern.
`*`	Match previous pattern 0 or more times.
`+`	Match previous pattern 1 or more times.
`?`	Match previous pattern 0 or 1 times.
`{n}`	Repeat n times.

{n,}	Repeat n or more times.
{n,m}	Repeat from n to m times.

If you'll add ? after operator, it becomes nongreedy. For example * operator becomes nongreedy if placing *? Greedy operator tries to take as much in string, as it can. Non-greedy takes by minimum.

6. Extended operators

Table A.5. Extended Operators

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

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

B. Format of `catalog.xml` file

Catalog of all Colorer Library resources is a convenient way to unify creation and management of all the Colorer features. This catalog is stored in `catalog.xml` file, and mapped into the ParserFactory class. Catalog supports storing of all installed HRC modules, management of error logging and listing of available HRD sets.

Element Name: `catalog`, type: `catalog`

Describes all available Colorer Library resources.

Content:

Element: `hrc-sets`, type: `hrc-sets`

Lists all installed root locations of HRC codes.

Element: `hrd-sets`, type: `hrd-sets`

Lists all available HRD sets.

Element Name: `hrc-sets`, type: `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`, type: `location`

Single resource location.

Element Name: `hrd-sets`, type: `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 Name: `hrd-entry`, type: `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`, type: `location`

Single resource location.

Element Name: `location`, type: `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 storage is used to assign each HRC Region with some editor-specific properties. Commonly, these include color and style information. HRD file consists of the list of the entries, each of them describing one HRC Region.

Element Name: `hrd`, type: `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: `assign`, type: `assign`

Single entry, describes region's properties.

Element Name: `assign`, type: `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 settings for different languages, or to compile them into one single HRD file. The former allows you to distribute recommended settings with each language, and the latter - to unify modifying and storing changed HRD settings with 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 minOccurs="0" maxOccurs="unbounded">
      <element name="assign" type="assign"/>
    </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="[1-3]"/>
    </restriction>
  </simpleType>
</schema>
```

D. XML Schema for HRC Language

This XML Schema instance 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 in other, than informational purposes, use up-to-date version, available on 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>
    <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" minOccurs="0">
        <complexType>
          <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>
      </element>
    </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="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="REentity" 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">
  <simpleContent>
    <extension base="REstring-or-null">
      <attribute name="match" type="REstring">
      </attribute>
      <attribute name="priority" type="priority" default="normal"/>
      <attributeGroup ref="regionX"/>
    </extension>
  </simpleContent>
</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"/>
    <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"/>
      </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>
    <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.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.