CodeML: An Open Markup Format the Content and Presentatation of Program Code

Michael Kohlhase
Computer Science, Carnegie Mellon University
Pittsburgh, Pa 15213, USA
http://www.cs.cmu.edu/~kohlhase

February 15, 2008

Abstract

This specification defines the Code Markup Language, or CodeML. CodeML is an XML application for describing progra codem and capturing both its structure and content. The goal of CodeML is to enable program code to be served, received, and processed on the World Wide Web, just as HTML has enabled this functionality for text.

CodeML is inspired by the MathML and OpenMath format for markup of mathematical formulae and can be used in conjuction with these formats in a variety of document formats.

Status of this Document: This specification of the markup language CodeML is intended primarily for a readership consisting of those who will be developing or implementing renderers or editors using it, or software that will communicate using CodeML as a protocol for input or output. It is not a User's Guide but rather a reference document.

The current report is an early draft of the specification of the CodeML format. Even though the basic layout of the format should survive, details like element- or attribute names, and content models are very likely to change as implementation and application experience grows.

Contents

Chapter 1

Introduction

This specification defines the Code Markup Language, or CodeML. CodeML is an XML application for describing program code and capturing both its structure and content. The goal of CodeML is to enable program code to be served, received, and processed on the World Wide Web, just as HTML has enabled this functionality for text.

The *primary purpose* of program code is as a means of communication between the programmer and the machine as a vehicle to express programs that can be executed by the machine. For this purpose, the field of computer science has developed a multitude of programming languages, i.e. formal languages with a well-defined structure and fixed semantics that allow the programmer to express her operational intentions as texts called program code. This code is then interpreted by a suitable programs (e.g. a compiler) and translated into native instructions to the computer, which are then executed to achieve the desired result.

An *auxiliary purpose* of program code is to communicate about algorithms and programs between humans, e.g. in program development documentation, quality assurance, or computer science education. To allow humans to cope with the complexities of program code, the program development process is supported by a variety of "intelligent development environments", which provide features like code pretty-printing (e.g. syntax highlighting, indentation), hyper-linking, and even higher-level techniques like literate programming, or documentation extraction from commented code.

Unfortunately, this functionality is directly tied to the development environment and (usually) to a particular programming language: The program text has to be analyzed, and the structural and syntactic information has

to be conveyed to the user in a dedicated user interface. It is currenly only possible to export the functionality of these development environments to the World Wide Web to a very limited extent.

- Web markup formats like HTML only provide limited primitives for pretty-printing program listings; the possibilities are largely limited to using the `pre` element, tables, or non-breaking spaces ` `. Together with CSS styling this is used for (static, fixed-width) indenting and syntax high-lighting.

- Using technologies like XSL formatting objects (which allow flexible-width formatting[1]) is problematic, since the syntactic and structural information necessary for pretty-printing is implicit in the program structure and a full parser that is needed to reveal it, transcends the possibilities of both `javascript` and and XSLT (which are the computational engines built into current browsers). Moreover, an approach based on parsing programming languages would necessitate the availability of gramars for all programming languages. EdNote(1)

- As programming languages are optimized for compilation and execution, they do not supply an infrastructure for referencing program fragments, which would be[2] EdNote(2)

[3] EdNote(3)

The concrete design of the CodeML format is inspired by the MathML (the Mathematical Markup Language) and OpenMath format for markup of mathematical formulae and can be used in conjuction with these formats in a variety of document formats.

The most relevant related work is probably the `listings.sty` package [**?**] for LaTeX, which allows to mark up program listings for print presentation.[4] EdNote(4)

[5] EdNote(5)

[1] EdNote: check that they do

[2] EdNote: continue

[3] EdNote: say something more, dynamics, local services without program interpretation. This is similar to the situation to math formulae, say something about what we can do with MathML

[4] EdNote: say some more, maybe move somewhere else

[5] EdNote: given an overview over what there already is.

Chapter 2

CodeML Elements

The CodeML format is loosely modeled after the MathML format [**?**] format. It has three kinds of elements, which we will specify in the rest of the section: presentation elements (section 2.1), content elements (section 2.2), and integration elements (section 2.3).

Since the CodeML fragment is intended as a module for the OMDoc [**?**] format, it only needs to be concerned with representing the actual code and can re-use the library- and theory levels of OMDoc[6].

EdNote(6)

The CodeML presentation elements are used to mark up the syntactic structure of program source code. They classify and group sub-strings of code in arbitrary programming languages by their syntactic function. As a consequence, generic tools can perform added-value services, without requiring parsers for the code.[7]

EdNote(7)

2.1 Presentation Elements

In this section we will introduce the CodeML presentation elements, which can be used to classify and group sub-strings of program code by their syntactic function.

All presentation CodeML elements have the following atrributes in common:

`xlink:xref` This optional attribute specifies a cross-reference to a content-CodeML element that corresponds to the code contained in this element. This attribute (together with the next two) conforms to the

[6] EdNote: I hope

[7] EdNote: continue, say something about content, interaction.

Listing 2.1: A java code snippet

```
public int find (int x) {
    if (s[x] < 0) return x;
    return find (s[x]);
    }
```

W3C XLink recommendation [**?**] and is recognized by many XML applications.

`xlink:type` This attribute must have the value `'simple'`, according to the XLink recommendation. It specifies that `xlink:xref` is a so-called simple link, like the `a` element from HTML. The DTD (see appendix **??**) fixes this attribute, so in CodeML documents with DTD, it need not be explicitly specified.

`xlink:role` This attribute specifies the role of link established by the `xlink:xref` as a semantic equivalence. Like the `xlink:type`, the DTD fixes its value.

We will use the code fragment for a Java implementation of the `find` procedure in Listing 2.1 as a running example in this section. The presentation-CodeML equivalent of this presented in Listing 2.2.

2.1.1 Token Elements

Token elements in presentation markup are broadly intended to represent the smallest code fragments which carry meaning. Thus character data in CodeML markup is only allowed to occur as part of the content of token elements. The only exception is whitespace between elements, which is ignored. Token elements can contain any sequence of zero or more Unicode characters.

Token elements should be rendered as their content (i.e. in the visual case, as a closely-spaced horizontal row of standard glyphs for the characters in their content). Rendering algorithms should also take into account the style attributes as described below, and modify surrounding spacing by rules or attributes specific to each type of token element.

CodeML has the following three elements to classify token elements; all of these can contain arbitrary unicode text[1]

[1] and the XML elements specified by changing the `&cpt.extra.content;` parameter entity either in the internal subset of the DOCTYPE declaration or in the DTD driver

Listing 2.2: The CodeML presentation for the code snippet in Listing 2.1

```
<cpg>
  <cpg>
    <cpo>public</cpo><cptype>int</cptype><cpo>find</cpo>
    <cpg open="(" close=")"><cptype>int</cptype><cpi>x</cpi></cpg>
  </cpg>
  <cpg open="{" close="}" breakO="hard">
    <cpo>if</cpo>
    <cpg open="(" close=")">
      <cpo>s</cpo>
      <cpg open="[" close="]"><cpi>x</cpi></cpg>
      <cpo>&amp;lt;</cpo>
      <cpb type="number">0</cpb>
    </cpg>
    <cpg close=";" Cbreak="hard"><cpo>return</cpo><cpi>x</cpi></cpg>
    <cpg close=";" Cbreak="hard">
      <cpo>return</cpo>
      <cpg>
        <cpo>find</cpo>
        <cpg open="(" close=")">
          <cpo>s</cpo>
          <cpg open="[" close="]"><cpi>x</cpi></cpg>
        </cpg>
      </cpg>
    </cpg>
  </cpg>
</cpg>
```

`cpb` for basic objects. This element should be used for values like numbers,[8]. It has the optional attribute `type`, which can take the values `'number'`[9] to classify its contents. EdNote(8) EdNote(9)

`cpo` for operators, i.e. names of symbols with a fixed meaning. These include the built-in operators of the programming language, functions that are imported from a library, and defined ones. The optional `type` supplies the values `'built-in'`, `'imported'`, and `'defined'` for these cases.[10] EdNote(10)

`cpi` for identifiers in the programming language, these are usually variable names[11]. EdNote(11)

`cptype` for data types[12]. EdNote(12)

The classification supported by these elements can be used by rendering engines e.g. to highlight or color program code.

Element	Attributes		Content
	Required	Optional	
`cpb`		`id, xlink:*, type, variant, style, color, background`	`CDATA`
`cpo`		`id, xlink:*, type, variant, style, color, background`	`CDATA`
`cpi`		`id, xlink:*, variant, style, color, background`	`CDATA`
`cptype`		`id, xlink:*, variant, style, color, background`	`CDATA`

Figure 2.1: The CodeML Token Elements

2.1.2 Token Element Attributes

Until full Unicode support is widely available, CodeML provides four code style attributes (see Figure 2.2). These attributes are valid on all presenta-

[8] EdNote: Maybe better call it cpv?
[9] EdNote: what else?
[10] EdNote: are there more? yes: recursive-call and definiiens
[11] EdNote: what else?
[12] EdNote: do we want this as a separate thing, or can we subsume them into cpo

tion token elements and on no other elements except `cstyle`.

Name	values	default
`variant`	`'normal'`, `'bold'`, `'italic'`, `'bold-italic'`, `'double-struck'`, `'bold-fraktur'`, `'script'`, `'bold-script'`, `'fraktur'`, `'sans-serif'`, `'bold-sans-serif'`, `'sans-serif-italic'`, `'sans-serif-bold-italic'`, `'monospace'`	normal
`size`	`'small'`, `'normal'`, `'big'`, `'number'`, `'v-unit'`	inherited
`color`, `background`	`'#rgb'`, `'#rrggbb'`, `'aqua'`, `'black'`, `'blue'`, `'fuchsia'`, `'gray'`, `'green'`, `'lime'`, `'maroon'`, `'navy'`, `'olive'`, `'purple'`, `'red'`, `'silver'`, `'teal'`, `'white'`, `'yellow'`	inherited

Figure 2.2: CodeML Style Attributes

The style attributes define logical classes of token elements. Each class is intended to correspond to a collection of typographically-related symbolic tokens that have a meaning within a given math expression, and therefore need to be visually distinguished and protected from inadvertent document-wide style changes which might change their meanings.

When CodeML rendering takes place in an environment where CSS is available, the style attributes can be viewed as predefined selectors for CSS style rules. When CSS is not available, it is up to the internal style mechanism of the rendering application to visually distinguish the different logical classes.

Token elements also permit `id`, `class` and `style` attributes for compatibility with style sheet mechanisms.[13]

EdNote(13)

Since CodeML expressions are often embedded in a textual data format such as XHTML, the surrounding text and the CodeML must share rendering attributes such as font size, so that the renderings will be compatible in style. For this reason, most attribute values affecting text rendering are inherited from the rendering environment, as shown in the 'default' column in the table above.

[13] EdNote: rethink; put this into DTD

The `color` attribute controls the color in which the content of tokens is rendered. Additionally, when inherited from `cstyle`, it controls the color of all other drawing by CodeML elements.

The values of `color`, and `background` can be specified as as an HTML color name[2] or as a string consisting of '#' followed without intervening whitespace by either 1-digit or 2-digit hexadecimal values for the red, green, and blue components, respectively, of the desired color, with the same number of digits used for each component (or as the keyword 'transparent' for background). The hexadecimal digits are not case-sensitive. The possible 1-digit values range from 0 (component not present) to F (component fully present), and the possible 2-digit values range from 00 (component not present) to FF (component fully present), with the 1-digit value x being equivalent to the 2-digit value xx (rather than x0).

The color syntax described above is a subset of the syntax of the color and background-color properties of CSS. The background-color syntax is in turn a subset of the full CSS background property syntax, which also permits specification of (for example) background images with optional repeats. The more general attribute name background is used in CodeML to facilitate possible extensions to the attribute's scope in future versions of CodeML.

2.1.3 Grouping, Indenting and Breaking

One of the most important CodeML uses of the information in presentation CodeML markup is to indent and linebreak program code. Conventionally, this is hardcoded by tabstops and spaces in the source code, which is sufficient for program development, but carries implicit assumptions about screen size and output format, which are not sustainable in the Internet and ubitiquos computing age (just imagine debugging lisp code on a palmtop). Therefore CodeML relies on the concept of "semantic" concept of code grouping to convey the syntactical structure of programming language expressions, and relies on style sheets or presentation engine to translate this into visual aids, such as indentation, or folding.

CodeML uses the `cpg` element to group program code fragments, as a consequence, it can contain any (number of) presentation CodeML elemements. Since grouping is often supported by bracketing structures in programming langues, the `cpg` element has three optional attributes `open` and `close` which specify the opening and closing brackets used the presented code. The children of the group are separated by single spaces, unless this

[2]Note that the color name keywords are not case-sensitive, unlike most keywords in CodeML attribute values for compatibility with CSS and HTML.

is overridden by setting the optional `separator` attribute to something else (e.g. a linefeed). Furthermore, the `cpg` element has an optional `indent` attribute, which allows the author to specify a factor for the default indentation increment used by the presentation agent. Finally, the `cpg` element has six attributes that control the linebreaking. They come in three groups of pairs one each for opening, closing brackets and separators. `breakO` specifies the break before the opening bracket, and `Obreak` a break after it. Attributes `breakC` and `Cbreak` to the analogous for the closing brackets, while `breakS` and `Sbreak` address the breaks for the separators[3]. Currently, CodeML does not restrict the values of these attributes leaving the specifics to applications; we suggest using the keyword `hard` for "hard" (unconditional) breaks. A system of numbers might be used for s.[14]

EdNote(14)

Even if the ultimate linebreaking of the code presentation has to be determined by the presentation agent, CodeML provides the empty `cpbr` element that allows the author to force a line break even if the presentation agent does not see the need for one on the basis of the grouping information.[15]

EdNote(15)

Element	Attributes		Content
	Required	Optional	
`cpg`		`id, xlink:*, open, close, separator, indent, breaks`	`(cpg| cpb| cpo| cpi| cpbr| cptype| cpd| cpc |cpr | cpstyle)*`
`cpbr`		`id, xlink:*`	`CDATA`

Figure 2.3: The CodeML Grouping Elements

2.1.4 Descriptive Text and Comments

Many programs contain descriptive text, either as documentation, comments, or (in the case of pseudo-code) even as part of the "programming language". CodeML provides the elements `cpd` for descriptive text and `cpc` for comments. Both elements can contain a multilingual group of `cpt` elements that contain the actual natural language text.

`cpt` elements have an `xml:lang` attribute that specifies the language they are written in. Thus using multilingual groups of `cmp` elements with

[3]Note that we cannot specify a breaking strategy for individual separator positions; all are treated alike.

[14]EdNote: give/discuss examples

[15]EdNote: do we really need this

Element	Attributes		Content
	Required	Optional	
`cpt`		`variant, style, color, background`	`CDATA`
`cpd`			`cpt*`
`cpc`			`cpt*`

Figure 2.4: The CodeML Text Elements

different languages can promote CodeML internationalization. Conforming with the XML recommendation, we use the ISO 639 two-letter country codes (`en` ≙ English, `de` ≙ German, `fr` ≙ French, `nl` ≙ Dutch,...). This optional attribute has the default "`'en'`", so that if no `xml:lang` is given, then English is assumed. Of course it is forbidden to have more than one `cpt` per value of `xml:lang` per `cpd` or `cpc` element, moreover, `cpt`s that are siblings must be translations of each other.

Let us consider an example: Listing 2.3 shows some presentation-CodeML for our running example from Listing 2.1. This example is mildly internationalized: in line 10, we have a `cpd` element and in line 19, we have a comment; both English and German text. A suitable presentation engine could generate localized presentations like the ones in Figure 2.5 from the source in Listing2.3.[16] EdNote(16)

```
procedure find (x)
    if s[x] is negative
        then return x            (we are at the root)
        otherwise return s[x]    (way to go!)
```

```
procedure find (x)
    if s[x] ist negativ
        then return x            (Wir sind an der Wurzel)
        otherwise return s[x]    (auf gehts!)
```

Figure 2.5: Multi-lingual Pseudo-code presentation from Listing 2.3

[16] EdNote: can the text contain code snippets again? Update the DTD and the examples to allow for that.

Listing 2.3: Pseudo-code for the code snippet in Listing 2.1

```
<code format="pseudo-code">
<cpg>
  <cpo>procedure</cpo>
  <cpg indent="0">
    <cpi>find</cpi>
    <cpg open="(" close=")"><cpi>x</cpi></cpg>
    <cpg ="break" breaks="bo-c">
      <cpg>
        <cpo>if</cpo>
        <cpd>
          <cpt>s[x] is negative</cpt>
          <cpt xml:lang="de">s[x] ist negativ</cpt>
        </cpd>
      </cpg>
      <cpg>
        <cpo>then</cpo>
        <cpo>return</cpo>
        <cpi>x</cpi>
        <cpc>
          <cpt>we are at the root</cpt>
          <cpt xml:lang="de">Wir sind in der Wurzel</cpt>
        </cpc>
      </cpg>
      <cpg>
        <cpo>otherwise</cpo>
        <cpo>return</cpo>
        <cpi>s[x]</cpi>
        <cpc>
          <cpt>way to go!</cpt>
          <cpt xml:lang="de">auf gehts!</cpt>
        </cpc>
      </cpg>
    </cpg>
  </cpg>
</cpg>
</code>
```

2.1.5 Raw Code

To facilitate migration from raw code and to allow for partial markup, CodeML allows to include[17] raw code in the `cpr` element[18].[19] Issue(17) Issue(18)

2.1.6 Style Change EdNote(19)

The `cstyle` element is used to make style changes that affect the rendering of its contents. `cstyle` can be given any attribute accepted by any CodeML presentation element provided that the attribute value is inherited, computed or has a default value; presentation element attributes whose values are required are not accepted by the mstyle element. In addition mstyle can also be given certain special attributes listed below.

The `cstyle` element accepts any number of arguments. If this number is not 1, its contents are treated as a single 'inferred `cpg`' formed from all its arguments.

Loosely speaking, the effect of the `cstyle` element is to change the default value of an attribute for the elements it contains. Style changes work in one of several ways, depending on the way in which default values are specified for an attribute. The cases are:

- Some attributes, such as `size` or `color`, are inherited from the surrounding context when they are not explicitly set. Specifying such an attribute on an `cstyle` element sets the value that will be inherited by its child elements. Unless a child element overrides this inherited value, it will pass it on to its children, and they will pass it to their children, and so on. But if a child element does override it, the new (overriding) value will be passed on to that element's children, and then to their children, etc, until it is again overridden.

- Other attributes, such as `variant`, have default values that are not normally inherited. That is, if the `variant` attribute is not set on an element, it will normally use the default value of `'normal'`, even if it was contained in a larger element that set this attribute to a different value. For attributes like this, specifying a value with an `cstyle` element has the effect of changing the default value for all elements within its scope. The net effect is that setting the attribute value with `cstyle` propagates the change to all the elements it contains

[17] Issue: do we also want to allow this in content?
[18] Issue: collapse this with the `rawcode` element?
[19] EdNote: table?

directly or indirectly, except for the individual elements on which the value is overridden. Unlike in the case of inherited attributes, elements that explicitly override this attribute have no effect on this attribute's value in their children.

Note that attribute values inherited from an `cstyle` in any manner affect a given element in the `cstyle`'s content only if that attribute is not given a value in that element's start tag. On any element for which the attribute is set explicitly, the value specified on the start tag overrides the inherited value.

As stated above, `style` accepts all attributes of all CodeML presentation elements which do not have required values. That is, all attributes which have an explicit default value or a default value which is inherited or computed are accepted by the mstyle element.

2.2 Content Elements

The content fragment of CodeML is used to represent the abstract structure of the algorithm underlying the program and unambiguously identify the symbols used in it, independently of the surface form. We will again use the Java code fragment for the `find` procedure in Listing 2.1 as a running example in this section. The content-CodeML equivalent of this is given in Listing 2.4.

EdNote(20)

The CodeML content elements only have the common CodeML attributes[20] attributes, and an optional `id` attribute, which can be used for cross-referencing by by the `xlink` attributes by the presentation-CodeML elements (see the introduction to section 2.1 and section 2.3).

2.2.1 Token Elements

The CodeML token elements are similar in nature to the OpenMath elements, we have:

`ccv` for variables: this element has a required attribute `name` for the name of the variable. Since for variables the name determines the content, this element is empty.

`ccsym` for defined symbols: this element is empty as well and has two required attributes `name` for the name of the symbols, and `cd` for the

[20] EdNote: crossref, define them somewhere above

Listing 2.4: The CodeML content for the code snippet in Listing 2.1

```
<ccdef export="find">
  <ccsym cd="java.dec" name="public-type-function"/>
  <apply>
    <ccsym cd="java.types" name="funtype"/>
    <ccsym cd="java.types" name="int"/>
    <ccsym cd="java.types" name="int"/>
  </apply>
  <bind>
    <ccsym cd="java.proc" name="function"/>
    <bvar><ccv name="x"/></bvar>
    <apply>
      <ccsym cd="java.control" name="if"/>
      <apply>
        <ccsym cd="java.arith" name="less"/>
        <apply>
          <ccsym cd="java.array" name="select"/>
          <ccv name="s"/>
          <ccv name="x"/>
        </apply>
        <ccb type="number">0</ccb>
      </apply>
      <apply>
        <ccsym cd="java.proc" name="return"/>
        <ccv name="x"/>
      </apply>
      <apply>
        <ccsym cd="java.proc" name="return"/>
        <apply>
          <ccsym cd="union-find" name="find"/>
          <apply>
            <ccsym cd="java.array" name="select"/>
            <ccv name="s"/>
            <ccv name="x"/>
          </apply>
        </apply>
      </apply>
    </apply>
  </bind>
</ccdef>
```

Element	Attributes		Content
	Required	Optional	
apply		id	(apply| bind| bvar| ccv| ccb| ccsym)*
bind		id	ccsym, bvar, (apply| bind| bvar| ccv| ccb| ccsym)
bvar		id	ccv*
ccv	name	id	EMPTY
ccysm	cd, name	id	EMPTY
ccdef	export	id	ccsym, (apply| bind| bvar| ccv| ccb| ccsym)

Figure 2.6: The CodeML Content Elements

that describes, implements, or defines the meaning of the symbol. These two attributes totally determine the content of the symbol (by reference to a defining document[21]), so the `ccsym` element is empty.

EdNote(21)

`ccb` that contains basic data, such as numbers, strings, etc. The type of this data can be specified in the optional attribute `type`[22]

EdNote(22)

2.2.2 Complex Elements

Content-CodeML knows the follwing three complex elements, which allow to combine content-CodeML expressions to larger ones.

`apply` for (function/procedure) application. This element is a constructor for function- and procedure calls: The first child is interpreted as a function or procedure and is given the remaining children of the `apply` element as argument.

`bind` this binding constructor `bind` is used to represent binding and abstraction constructions. It is primarily used in function and procedure definitions. The first child of this a CodeML binding expression must be a `ccsym` element, which defines a binding symbol. The second child must be a `bvar` element, which contains `ccv` elements for all the formal parameters of the binding. The third child of the `bind` element is known as the body. It is an arbitrary content-CodeML element,

[21] EdNote: say some more about the module system, ... later

[22] EdNote: do we want to enumerate the basic data types, or leave them open for the user?

which may (but in general need not) contain occurrences of the bound variables. Note that the bound variables specified in the `bvar` element only have or in the body. An occurrence of an `ccv` element outside the body is semantically a different variable, even if it carries the same name.

`ccdef` for definitions and declarations: In contrast to the `bind` element, this one defines a set of symbols to be visible outside its body: Its required `exports` attribute specifies a set of symbol names as a whitespace-separated list of names.

The first child of the `ccdef` element is a declaration symbol (specified as a `ccsym` element that specifies the definition schema. For instance, in the `ccdef` element in Listing 2.4, this is the symbol

<**ccsym cd**="java.dec"**name**="public−type−function"/>

that specifies that the second child is a type (the type of the defined symbol), and the third child is a representation of a function (in this case via a `bind` element).

2.3 Interaction of Content and Presentation

BegNP(23)

In the last sections we have seen two styles of markup for program code:

Presentation markup captures notational structure. It encodes the notational structure of an expression in a sufficiently abstract way to facilitate rendering to various media. It does this by providing information such as structured grouping of expression parts, classification of symbols, etc.

Presentation markup does not directly concern itself with the mathematical structure or meaning of the code. In many situations, notational structure and mathematical structure are closely related, so a sophisticated processing application may be able to heuristically infer mathematical meaning from notational structure, provided sufficient context is known. However, in practice, the inference of mathematical meaning from mathematical notation must often be left to the reader. As a consequence, employing presentation tags alone may limit the ability to re-use a CodeML object in another context, especially evaluation by external applications.

Content markup captures mathematical structure. It encodes mathematical structure in a sufficiently regular way in order to facilitate the assignment of mathematical meaning to an expression by application programs.

[23] New Part: copied from MathML, rework heavily

Though the details of mapping from mathematical expression structure to mathematical meaning can be extremely complex, in practice, there is wide agreement about the conventional meaning of many basic mathematical constructions. Consequently, much of the meaning of a content expression is easily accessible to a processing application, independently of where or how it is displayed to the reader. In many cases, content markup could be cut from a Web browser and pasted into a mathematical software tool with confidence that sensible values will be computed.

Since content markup is not directly concerned with how an expression is displayed, a renderer must infer how an expression should be presented to a reader. While a sufficiently sophisticated renderer and style sheet mechanism could in principle allow a user to read mathematical documents using personalized notational preferences, in practice, rendering content expressions with notational nuances may still require intervention of some sort. As a consequence, employing content tags alone may limit the ability of the author to precisely control how an expression is rendered.

Both content and presentation tags are necessary in order to provide the full expressive capability one would expect in a markup language for program code. CodeML offers authors elements for both content and presentation markup. Whether to use one or the other, or a combination of both, depends on what aspects of rendering and interpretation an author wishes to control, and what kinds of re-use he or she wishes to facilitate.

EndNP(23)

In many common situations, an author or authoring tool may choose to generate either presentation or content markup exclusively. Some applications however are able to make use of both presentation and content information. For these applications it is desirable to provide both forms of markup for the same mathematical expression.

Element	Attributes		Content
	Required	Optional	
semantics			(apply| bind| bvar| ccv| ccb| ccsym, ccdef)*, (pcode| rawcode)*
pcode	format	type, href,id, xlink:*, open, close, separator, indent, breaks	(cpg| cpb| cpo| cpi| cpbr| cptype| cpd| cpc |cpr | cpstyle)*
rawcode	format	type, href	PCDATA

Figure 2.7: The CodeML Interaction Elements

Listing 2.5: Conbining markup Styles

```
<semantics>
  <ccdef export="find"><!--...see Listing 2.4 ...--></ccdef>
  <pcode format="pseudo"><cpg> <!--...see Listing 2.3 ...--></cpg></pcode>
  <pcode format="java"><cpg> <!--...see Listing 2.2 ...--></cpg></pcode>
  <rawcode format="java"><!--...see Listing 2.1 ...--></rawcode>
</semantics>
```

To combine presentation markup and content markup, CodeML provides the `semantics` element. The first child of this element is a content CodeML expression, and remaining children are `pcode` or `rawcode` elements. The `pcode` elements contain presentation-CodeML representations of the program in the first child, and the `rawcode` elements raw code, i.e. code fragments that have not been marked up in CodeML. The `format` specifies the programming language, the code fragment is in. Note that it is an error to have more than one `pcode` or `rawcode` element with the same `format` attribute. As the example in Listing 2.5 shows, it is allowed to have `pcode` and `rawcode` elements of the same `format`.

In contrast to representation formats like MathML, which allow mixing presentation and content at arbitrary levels, the CodeML format only allows this at the top-level. This is called parallel markup.

2.3.1 Parallel Markup by Cross-references

Top-level pairing of independent presentation and content markup is sufficient for many, but not all, situations. Applications that allow treatment of sub-expressions of mathematical objects require the ability to associate presentation, content or information with the parts of an object with mathematical markup. Top-level pairing with a `semantics` element is insufficient in this type of situation; identification of a sub-expression in one branch of `semantics` element gives no indication of the corresponding parts in other branches.

To better accommodate applications that must deal with sub-expressions of large objects, CodeML uses cross-references between the branches of a `semantics` element to identify corresponding sub-structures.

Cross-referencing is achieved using `id` and `xlink:xref` attributes within the branches of a containing semantics element. These attributes may optionally be placed on MathML elements of any type.

An `id` attribute and a corresponding `xlink:xref` appearing within the same `semantics` element create a correspondence between sub-expressions.

In general, there will not be a one-to-one correspondence between nodes in parallel branches. For example, a presentation tree may contain elements, such as parentheses, that have no correspondents in the content tree. Therefore CodeML puts the `id` attributes on the branch with the content tree, which has the finest-grained node structure and the `xlink:xref` attributes on the presentation-branches.

Let us fortify our intuition about this with an extended development of the `find` procedure that combines all the material discussed above.

```
<?xml version="1.0" encoding="utf-8"?>
<code format="multi" xmlns="http://www.mathweb.org/codeml">
 <semantics>
  <ccdef export="find" id="find-def">
    <ccsym cd="java.dec" name="public-type-function"/>
    <apply id="find-type">
      <ccsym cd="java.types" name="funtype"/> <!--+\label{lst-cr:fig:find-ccode:ccsym}+-->
      <ccsym cd="java.types" name="int"/>
      <ccsym cd="java.types" name="int"/>
    </apply>
    <bind>
      <ccsym cd="java.proc" name="function"/> <!--+\label{lst-cr:fig:find-ccode:ccsym-bind}+-->
      <bvar><ccv name="x"/></bvar>   <!--+\label{lst-cr:fig:find-ccode:bvar}+-->
      <apply id="find-body">         <!--+\label{lst-cr:fig:find-ccode:body}+-->
        <ccsym cd="java.control" name="if"/>
        <apply id="cond">
          <ccsym cd="java.arith" name="less"/>
          <apply>
            <ccsym cd="java.array" name="select"/>
            <ccv name="s"/>          <!--+\label{lst-cr:fig:find-ccode:ccv}+-->
            <ccv name="x"/>
          </apply>
          <ccb type="number">0</ccb>   <!--+\label{lst-cr:fig:find-ccode:ccb}+-->
        </apply>
        <apply>
          <ccsym cd="java.proc" name="return"/>
          <ccv name="x"/>
        </apply>
        <apply>
          <ccsym cd="java.proc" name="return"/>
          <apply>
            <ccsym cd="union-find" name="find"/>
            <apply>
              <ccsym cd="java.array" name="select"/>
              <ccv name="s"/>
              <ccv name="x"/>
            </apply>
          </apply>
        </apply>
      </apply>
    </bind>
  </ccdef>
  <pcode format="java">
    <cpg>
      <cpo>public</cpo><cptype>int</cptype><cpo>find</cpo>
```

```
      <cpg open="(" close=")"><cptype>int</cptype><cpi>x</cpi></cpg>
    </cpg>
    <cpg open="{" close="}" breakO="hard" xref="find-body">
      <cpo>if</cpo>
      <cpg open="(" close=")" xref="cond">
        <cpo>s</cpo>
        <cpg open="[" close="]"><cpi>x</cpi></cpg>
        <cpo>&amp;lt;</cpo>
        <cpb type="number">0</cpb>
      </cpg>
      <cpg close=";" breakC="hard"><cpo>return</cpo><cpi>x</cpi></cpg>
      <cpg close=";" breakC="hard">
        <cpo>return</cpo>
        <cpg>
          <cpo>find</cpo>
          <cpg open="(" close=")">
            <cpo>s</cpo>
            <cpg open="[" close="]"><cpi>x</cpi></cpg>
          </cpg>
        </cpg>
      </cpg>
    </cpg>
  </pcode>
  <pcode format="pseudo-code">
  <cpg>
    <cpo>procedure</cpo>
    <cpg indent="0">
      <cpi>find</cpi>
      <cpg open="(" close=")"><cpi>x</cpi></cpg>
      <cpg separator="" Sbreak="hard" Obreak="hard" xref="find-body">
        <cpg>
          <cpo>if</cpo>
          <cpd xref="cond">
            <cpt>s[x] is negative</cpt>
            <cpt xml:lang="de">s[x] ist negativ</cpt>
          </cpd>
        </cpg>
        <cpg>
          <cpo>then</cpo>
          <cpo>return</cpo>
          <cpi>x</cpi>
          <cpc>
            <cpt>we are at the root</cpt>
            <cpt xml:lang="de">Wir sind in der Wurzel</cpt>
          </cpc>
        </cpg>
        <cpg>
          <cpo>otherwise</cpo>
          <cpo>return</cpo>
          <cpi>s[x]</cpi>
          <cpc>
            <cpt>way to go!</cpt>
            <cpt xml:lang="de">auf gehts!</cpt>
          </cpc>
        </cpg>
      </cpg>
```

```
    </cpg>
  </cpg>
</pcode>
 <rawcode format="java"><![CDATA[
public int find (int x) {
     if (s[x] < 0) return x;
     return find (s[x]);
     }
]]>
 </rawcode>
 </semantics>
</code>
```

??

Chapter 3

Conclusion

[24] [25]

EdNote(24)
EdNote(25)

[24] EdNote: say something about what we have done
[25] EdNote: say something about why this is useful

Index

Appendix A

Quick-Reference Table to the CodeML Elements

Element	p.	Type	Required Attribs	Optional Attribs	Content
apply	16	Content		id	(apply| bind| bvar| ccv| ccb| ccsym)*
bind	16	Content		id	ccsym, bvar, (apply| bind| bvar| ccv| ccb| ccsym)
bvar	16	Content		id	ccv*
ccdef	16	Content	export	id	ccsym, (apply| bind| bvar| ccv| ccb| ccsym)
ccsym	16	Content	cd, name	id	EMPTY
ccv	14	Content	name	id	EMPTY
cpb	7	PresToken		id, xlink:*, type, variant, style, color, background	CDATA
cpbr	10	PresGroup		id, xlink:*	CDATA
cpc	10	PresText			cpt*
cpd	10	PresText			cpt*
cpg	9	PresGroup		id, xlink:*, open, close, separator, indent, breaks	(cpg| cpb| cpo| cpi| cpbr| cptype| cpd| cpc |cpr | cpstyle)*
cpi	7	PresToken		id, xlink:*, variant, style, color, background	CDATA

cpo	7	PresToken		id, xlink:*, type, variant, style, color, background	CDATA
cpt	10	PresText		variant, style, color, background	CDATA
cptype	7	PresToken		id, xlink:*, variant, style, color, background	CDATA
pcode	18	Inter	format	type, href,id, xlink:*, open,close, separator, indent, breaks	(cpg| cpb| cpo| cpi| cpbr| cptype| cpd| cpc |cpr | cpstyle)*
rawcode	18	Inter	format	type, href	PCDATA
semantics	19	Inter			(apply| bind| bvar| ccv| ccb| ccsym, ccdef)*,(pcode| rawcode)*

Appendix B

Quick-Reference Table to the CodeML Attributes

Attribute	*element*	Values
breaks	cpg	bob-bcb, ob-bcb, bo-bcb, o-bcb, bob-cb, ob-cb, bo-cb, o-cb, bob-bc, ob-bc, bo-bc, o-bc, bob-c, ob-c, bo-c, o-c
	SAY SOMETHING HERE, also about prioritized breaks	
cd	ccsym	theory/module name
	The name of a theory or a module that exports this symbol	
close	cpg	string
	The closing bracket of group of CodeML constructs this is inserted into the presentation in place of the opneing tag of its cpg element	
color, background	p-CodeML	#*rgb*,#*rrggbb* aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white, yellow
	The color of the text representation or the background of a presentation-CodeML element	
exports	ccdef	string
	the name of a symbol defined by the ccdef element	
format	pcode,rawcode	
	The format, e.g. the programming language	
id	*	
	a string that identifies the element	
indent	cpg	number

	a factor for the default indentation increment used by the presentation agent	
`name`	`ccsym`	`string`
	The name the symbol	
`open`	`cpg`	`string`
	The opening bracket of group of CodeML constructs; this is inserted into the presentation in place of the opneing tag of its `cpg` element	
`separator`	`cpg`	`string`
	The separator bracket of group of CodeML constructs this is inserted into the presentation between the children of its `cpg` element	
`size`	`p-CodeML`	`small, normal, big,` number, v-unit
	The font size of the text representation of a presentation-CodeML element	
`type`	`cpb, ccb`	`number`
	the type of the basic object	
`type`	`cpo`	`built-in, imported, defined`
	the type of the basic object	
`variant`	`p-CodeML`	`normal, bold, italic, bold-italic, double-struck, bold-fraktur, script, bold-script, fraktur, sans-serif, bold-sans-serif, sans-serif-italic, sans-serif-bold-italic, monospace`
	The font variant of the text of a presentation-CodeML element	
`xlink:*`	`cp*`	
	a crossreference to a semantically equivalent content element; the value of `xlink:xref` (an URIRef) specifies the element, the value of `xlink:type` must be `simple` (it is fixed in the DTD), and the `xlink:role` is a fixed URL to a document that explains this link type.	
`xml:lang`	`cpt`	`ISO 639: en, de, it, ko, ...`
	the language the text is written in	

Appendix C

The CodeML RelaxNG Schema

We reprint the current version of the CodeML RelaxNG schema. The original can be found at `https://svn.omdoc.org/repos/codeml/RelaxNG`

C.1 The CodeML Schema Driver

```
################################################################################
# A RelaxNG schema for CodeML (presentation and content of program code)
# Schema driver
# Id : omdoc.rnc66342007 − 07 − 1306 : 32 : 51Zkohlhase
# HeadURL : https : //svn.omdoc.org/repos/omdoc/trunk/rnc/omdoc.rnc
# See the documentation and examples at https://www.omdoc.org/codeml
# Copyright (c) 2008 Michael Kohlhase, Deyan Ginev, Dimitar Misev, Catalin David
# released under the GNU Public License (GPL)
################################################################################

default namespace = "http://www.mathweb.org/codeml"

include "codeml−cont.rnc"
include "codeml−pres.rnc"
include "codeml−inter.rnc"

common.attrib = attribute id {xsd:ID}?

ISO639 = "aa"|"ab"|"af"|"am"|"ar"|"as"|"ay"|"az"|"ba"|"be"|"bg"|"bh"|"bi"|"bn"|"bo"|"br"|"ca"|"co"|
         "cs"|"cy"|"da"|"de"|"dz"|"el"|"en"|"eo"|"es"|"et"|"eu"|"fa"|" fi "|" fj "|"fo"|" fr "|"fy"|"ga"|
         "gd"|"gl"|"gn"|"gu"|"ha"|"he"|"hi"|"hr"|"hu"|"hy"|"ia"|"ie"|"ik"|"id"|" is "|" it "|"iu"|"ja"|
         "jv"|"ka"|"kk"|"kl"|"km"|"kn"|"ko"|"ks"|"ku"|"ky"|"la"|"ln"|"lo"|"lt"|" lv "|"mg"|"mi"|"mk"|
         "ml"|"mn"|"mo"|"mr"|"ms"|"mt"|"my"|"na"|"ne"|"nl"|"no"|"oc"|"om"|"or"|"pa"|"pl"|"ps"|"pt"|
         "qu"|"rm"|"rn"|"ro"|"ru"|"rw"|"sa"|"sd"|"sg"|"sh"|"si"|"sk"|" sl "|"sm"|"sn"|"so"|"sq"|"sr"|
         "ss"|" st "|" su"|"sv"|"sw"|"ta"|"te"|"tg"|"th"|" ti "|"tk"|" tl "|"tn"|"to"|" tr "|" ts "|" tt "|"tw"|
         "ug"|"uk"|"ur"|"uz"|"vi"|"vo"|"wo"|"xh"|"yi"|"yo"|"za"|"zh"|"zu"
```

```
xml.lang.attrib = attribute xml:lang {ISO639}?

start = code

#More declarations to come, still incomplete...
```

C.2 The CodeML Module PRES

Module PRES introduces the presentation CodeML elements.

```
##############################################################################
# A RelaxNG schema for CodeML (presentation and content of program code)
# Presentation Module
# Id : omdoc.rnc66342007 − 07 − 1306 : 32 : 51Zkohlhase
# HeadURL : https : //svn.omdoc.org/repos/omdoc/trunk/rnc/omdoc.rnc
# See the documentation and examples at https://www.omdoc.org/codeml
# Copyright (c) 2008 Michael Kohlhase, Deyan Ginev, Dimitar Misev, Catalin David
# released under the GNU Public License (GPL)
##############################################################################

default namespace = "http://www.mathweb.org/codeml"

cp.class = cpg | cpb | cpo | cpi | cpbr | cptype | cpd | cpc | cpr | cpstyle

cp.common.attrib = common.attrib, attribute xref {xsd:anyURI}?

cp.group.attrib =
    attribute open { text }?,
    attribute close { text }?,
    attribute separator { text }?,
    attribute indent { text }?,
    attribute Obreak { text }?,
    attribute breakO { text }?,
    attribute Cbreak { text }?,
    attribute breakC { text }?,
    attribute Sbreak { text }?,
    attribute breakS { text }?

cp.style.attrib =
    attribute variant { token }?,
    attribute size {text}?,
    attribute color { text }?,
    attribute background {text}?

cp.token.attrib = cp.common.attrib,cp.style.attrib

#Groups
cpg = element cpg{cp.class*,cp.common.attrib?,cp.group.attrib?}

#basic language objects
cpb = element cpb {text,cp.token.attrib?,attribute type {xsd:NCName}}

#operators
cpo.types = "built−in" | "imported" | "defined" | "definiens" | "recursive−call"
```

```
cpo = element cpo {text,cp.token.attrib, attribute type {cpo.types}?}

# identifiers
cpi = element cpi {text,cp.token.attrib}

#linebreaks
cpbr = element cpbr{cp.common.attrib}

#datatypes
cptype = element cptype {text,cp.common.attrib}

#multilingual text
cpt = element cpt{text,cp.common.attrib,cp.style.attrib,xml.lang.attrib}

#descriptions
cpd = element cpd {cpt+,cp.common.attrib}

#comments
cpc = element cpc {cpt+,cp.common.attrib}

#raw code
cpr = element cpr{text,cp.common.attrib}

#style information
cpstyle = element cpstyle {cp.class,cp.common.attrib,cp.style.attrib}
```

C.3 The CodeML Module CONT

Module CONT introduces the content CodeML elements.

```
################################################################################
# A RelaxNG schema for CodeML (presentation and content of program code)
# Content Markup Module
# Id : omdoc.rnc66342007 − 07 − 1306 : 32 : 51Zkohlhase
# HeadURL : https : //svn.omdoc.org/repos/omdoc/trunk/rnc/omdoc.rnc
# See the documentation and examples at https://www.omdoc.org/codeml
# Copyright (c) 2008 Michael Kohlhase, Deyan Ginev, Dimitar Misev, Catalin David
# released under the GNU Public License (GPL)
################################################################################

default namespace = "http://www.mathweb.org/codeml"

cc.class = apply | bind | bvar | ccv | ccb | ccsym
cc.top.class = cc.class | ccdef
cc.common.attrib = common.attrib

# application
apply = element apply {cc.class*, cc.common.attrib}

# binding
bind = element bind {ccsym, bvar,cc.class,cc.common.attrib}

# bound varibale declarations
bvar = element bvar {ccv*,cc.common.attrib}
```

```
25
    # variables
    ccv = element ccv {attribute name { text },cc.common.attrib}

    # basic language objects
30  ccb = element ccb {text,attribute type { xsd:NMTOKEN }?,cc.common.attrib?}

    # symbols (reserved names)
    ccsym = element ccsym {text,attribute cd { text },attribute name { text },cc.common.attrib}

35  # definitions
    ccdef = element ccdef {ccsym,cc.class*,attribute export { text },cc.common.attrib}

    # symbol declaration
    symbol = element symbol {type,cc.common.attrib}
40
    # types
    type = element type {cc.class,cc.common.attrib}
```

C.4 The CodeML Module INTER

Module INTER introduces elements for the interaction of content and presentation CodeML elements. The top-level `code` element is among then.

```
    ##################################################################
    # A RelaxNG schema for CodeML (presentation and content of program code)
3   # Interface Module
    # Id : omdoc.rnc66342007 − 07 − 1306 : 32 : 51Zkohlhase
    # HeadURL : https : //svn.omdoc.org/repos/omdoc/trunk/rnc/omdoc.rnc
    # See the documentation and examples at https://www.omdoc.org/codeml
    # Copyright (c) 2008 Michael Kohlhase, Deyan Ginev, Dimitar Misev, Catalin David
8   # released under the GNU Public License (GPL)
    ##################################################################

    default namespace = "http://www.mathweb.org/codeml"

13  xcode.attrib = attribute format { text },
                   attribute type { text }?,
                   attribute href { text }?

    semantics = element semantics {cc.top.class,(pcode | rawcode)*}
18
    dublincore = grammar {include "MARCRelators.rnc"
          include "dublincore.rnc"
                {dc.date = attribute action {xsd:NMTOKEN}?,attribute who {xsd:anyURI}?,xsd:dateTime
                 dc. identifier  = attribute scheme {xsd:NMTOKEN},text
23               dc.type = "Dataset" | "Text" | "Collection"
                 dc.text = parent common.attrib, parent xml.lang.attrib,text
                 dc.person = attribute role {MARCRelators}?,text}}

    metadata = element metadata {common.attrib,dublincore*}
28
    pcode.content = metadata?,cp.class*
```

```
pcode = element pcode {pcode.content,xcode.attrib,cp.group.attrib}

rawcode.content = text

rawcode = element rawcode {rawcode.content,xcode.attrib}

code = element code {common.attrib,cp.group.attrib,xcode.attrib,
                     metadata?,(cp.class| cc.top.class | semantics)*}
```

This schema includes the generic schema for Dublin Core Metadata and for the MARC relator set, we include both of them for reference

C.4.1 The Dublin Core Schema

```
# A RelaxNG schema for the Dublin Core elements
# Id : omdocdc.rnc63992007 − 05 − 2515 : 07 : 45Zkohlhase
# HeadURL : https : //svn.omdoc.org/repos/omdoc/branches/omdoc − 1.2/rnc/omdocdc.rnc
# See the documentation and examples at http://www.omdoc.org
# Copyright (c) 2004−2008 Michael Kohlhase, released under the GNU Public License (GPL)

default namespace dc = "http://purl.org/dc/elements/1.1/"

## the various content models, specialize for integration
dc.person = text
dc.publisher = text
dc.text = text
dc.format = text
dc.source = text
dc.language = text
dc.rights = text
dc.relation = text
dc.date = xsd:dateTime
dc.type = text
dc.identifier = text

# the model of the Dublin Metadata initiative (http://purl.org/dc)
start = contributor | creator | rights | subject | title | description | publisher
                  | date | type | format | identifier | source | language | relation

contributor = element contributor {dc.person}
creator = element creator {dc.person}
title = element title {dc.text}
subject = element subject {dc.text}
description = element description {dc.text}
publisher = element publisher {dc.publisher}
type = element type {dc.type}
format = element format {dc.format}
source = element source {dc.source}
language = element language {dc.language}
relation = element relation {dc.relation}
rights = element rights {dc.rights}
date = element date {dc.date}
identifier = element identifier {dc.identifier}
```

C.4.2 The MARC Relators Schema

```
# the MARC relator set; see http://www.loc.gov/marc/relators
MARCRelators =
        "act" | "adp" | "aft" | "ann" | "ant" | "app" | "aqt" |
        "arc" | "arr" | "art" | "asg" | "asn" | "att" | "auc" | "aud" | "aui" |
        "aus" | "aut" | "bdd" | "bjd" | "bkd" | "bkp" | "bnd" | "bpd" | "bsl" |
        "ccp" | "chr" | "clb" | "cli" | "cll" | "clt" | "cmm" | "cmp" | "cmt" |
        "cnd" | "cns" | "coe" | "col" | "com" | "cos" | "cot" | "cov" | "cpc" |
        "cpe" | "cph" | "cpl" | "cpt" | "cre" | "crp" | "crr" | "csl" | "csp" |
        "cst" | "ctb" | "cte" | "ctg" | "ctr" | "cts" | "ctt" | "cur" | "cwt" |
        "dfd" | "dfe" | "dft" | "dgg" | "dis" | "dln" | "dnc" | "dnr" | "dpc" |
        "dpt" | "drm" | "drt" | "dsr" | "dst" | "dte" | "dto" | "dub" | "edt" |
        "egr" | "elt" | "eng" | "etr" | "exp" | "fac" | "flm" | "fmo" | "fnd" |
        "fpy" | "frg" | "hnr" | "hst" | "ill" | "ilu" | "ins" | "inv" | "itr" |
        "ive" | "ivr" | "lbt" | "lee" | "lel" | "len" | "let" | "lie" | "lil" |
        "lit" | "lsa" | "lse" | "lso" | "ltg" | "lyr" | "mdc" | "mod" | "mon" |
        "mrk" | "mte" | "mus" | "nrt" | "opn" | "org" | "orm" | "oth" | "own" |
        "pat" | "pbd" | "pbl" | "pfr" | "pht" | "plt" | "pop" | "ppm" | "prc" |
        "prd" | "prf" | "prg" | "prm" | "pro" | "prt" | "pta" | "pte" | "ptf" |
        "pth" | "ptt" | "rbr" | "rce" | "rcp" | "red" | "ren" | "res" | "rev" |
        "rpt" | "rpy" | "rse" | "rsp" | "rst" | "rth" | "rtm" | "sad" | "sce" |
        "scl" | "scr" | "sec" | "sgn" | "sng" | "spk" | "spn" | "spy" | "srv" |
        "stl" | "stn" | "str" | "ths" | "trc" | "trl" | "tyd" | "tyg" | "voc" |
        "wam" | "wdc" | "wde" | "wit"
```