MusicXML: From DTD specification to Haskell implementation

Samuel Silva [silva.samuel@alumni.uminho.pt]

November 4, 2008

Contents

1 Introdution **1**

2 Implementation **2**
2.1 Attributes . . . 2
2.2 Barline . . . 16
2.3 Common . . . 19
2.4 Container . . . 51
2.5 Direction . . . 53
2.6 Identity . . . 79
2.7 Layout . . . 83
2.8 Link . . . 90
2.9 MusicXML . . . 93
2.10 Note . . . 96
2.11 Opus . . . 137
2.12 Partwise . . . 139
2.13 Score . . . 141
2.14 Timewise . . . 154
2.15 Util . . . 156

3 Test **164**

4 Conclusion **166**

1 Introdution

This document contains Haskell[4, 3] code formatted with lhs2TeX tool. Type definition are conforming specification built by Recordare at second version of MusicXML DTDs, presented on figure 1. At moment of writing this library, Recordare was publishing unstable versions of MusicXML schemas after discussion made by MusicXML comunity to improve its specification[2].

This library are architecture presented on figure 2. Type definition wasn't use `datatype`, using `type` definitions to improve performance. To improve maintainability was built Util module, which contains elementary functions to reading and writing. To minimize code, at reading from MusicXML format is used State Monad. However writing to MusicXML format it is used functional way.

This approach help us maintainability by close to MusicXML DTDs specification. Next section presents implementation using Haskell language of MusicXML DTD specification. Following section presents some use cases of this library into real examples of musicxml documents.

Figure 1: MusicXML architecture

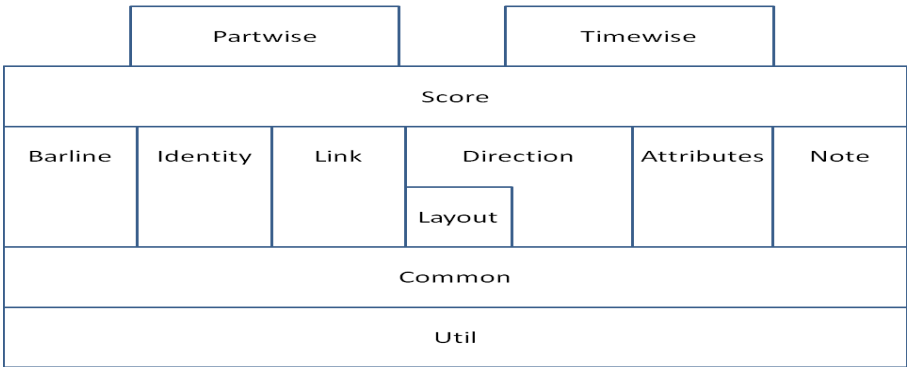

Figure 2: MusicXML modules

2 Implementation

This implementation shows comments from MusicXML specification[5, 1] like *this style*.

2.1 Attributes

```
-- |
-- Maintainer : silva.samuel@alumni.uminho.pt
-- Stability : experimental
-- Portability: HaXML
--
module Text.XML.MusicXML.Attributes where
import Text.XML.MusicXML.Common hiding (Directive, read_Directive, show_Directive)
import Text.XML.HaXml.Types (Content)
import Control.Monad (MonadPlus (..))
import Prelude (Maybe (..), Show, Eq, Monad (..), (.), String, (++))
```

The attributes DTD module contains the attributes element and its children, such as key and time signatures.

The attributes element contains musical information that typically changes on measure boundaries. This includes key and time signatures, clefs, transpositions, and staving.

```
-- * Attributes
-- |
type Attributes = (Editorial, Maybe Divisions, [Key], [Time],
  Maybe Staves, Maybe Part_Symbol, Maybe Instruments, [Clef], [Staff_Details],
  Maybe Transpose, [Directive], [Measure_Style])
-- |
```

```
read_Attributes :: Eq i ⇒ STM Result [Content i] Attributes
read_Attributes = do
  y ← read_ELEMENT "attributes"
  read_12 read_Editorial (read_MAYBE read_Divisions)
    (read_LIST read_Key) (read_LIST read_Time)
    (read_MAYBE read_Staves) (read_MAYBE read_Part_Symbol)
    (read_MAYBE read_Instruments) (read_LIST read_Clef)
    (read_LIST read_Staff_Details) (read_MAYBE read_Transpose)
    (read_LIST read_Directive) (read_LIST read_Measure_Style)
    (childs y)
  -- |
show_Attributes :: Attributes → [Content ()]
show_Attributes (a, b, c, d, e, f, g, h, i, j, k, l) =
  show_ELEMENT "attributes" []
    (show_Editorial a ++ show_MAYBE show_Divisions b ++
      show_LIST show_Key c ++ show_LIST show_Time d ++
      show_MAYBE show_Staves e ++ show_MAYBE show_Part_Symbol f ++
      show_MAYBE show_Instruments g ++
      show_LIST show_Clef h ++ show_LIST show_Staff_Details i ++
      show_MAYBE show_Transpose j ++ show_LIST show_Directive k ++
      show_LIST show_Measure_Style l)
```

Traditional key signatures are represented by the number of flats and sharps, plus an optional mode for major/ minor/mode distinctions. Negative numbers are used for flats and positive numbers for sharps, reflecting the key's placement within the circle of fifths (hence the element name). A cancel element indicates that the old key signature should be cancelled before the new one appears. This will always happen when changing to C major or A minor and need not be specified then. The cancel value matches the fifths value of the cancelled key signature (e.g., a cancel of -2 will provide an explicit cancellation for changing from B flat major to F major). The optional location attribute indicates whether the cancellation appears to the left or the right of the new key signature. It is left by default.

Non-traditional key signatures can be represented using the Humdrum/Scot concept of a list of altered tones. The key-step and key-alter elements are represented the same way as the step and alter elements are in the pitch element in the note.mod file. The different element names indicate the different meaning of altering notes in a scale versus altering a sounding pitch.

Valid mode values include major, minor, dorian, phrygian, lydian, mixolydian, aeolian, ionian, and locrian.

The optional number attribute refers to staff numbers, from top to bottom on the system. If absent, the key signature applies to all staves in the part.

The optional list of key-octave elements is used to specify in which octave each element of the key signature appears. The content specifies the octave value using the same values as the display-octave element. The number attribute is a positive integer that refers to the key signature element in left-to-right order. If the cancel attribute is set to yes, then this number refers to an element specified by the cancel element. It is no by default.

```
  -- ** Key
  -- |
type Key = ((Maybe CDATA, Print_Style, Print_Object),
  (Key_, [Key_Octave]))
  -- |
read_Key :: Eq i ⇒ STM Result [Content i] Key
read_Key = do
  y ← read_ELEMENT "key"
  y1 ← read_3 (read_IMPLIED "number" read_CDATA)
    read_Print_Style read_Print_Object (attributes y)
  y2 ← read_2 read_Key_ (read_LIST read_Key_Octave) (childs y)
  return (y1, y2)
  -- |
show_Key :: Key → [Content ()]
show_Key ((a, b, c), (d, e)) =
```

```
    show_ELEMENT "key" (show_IMPLIED "number" show_CDATA a ++
      show_Print_Style b ++ show_Print_Object c)
      (show_Key_ d ++ show_LIST show_Key_Octave e)
    -- |
data Key_ = Key_1 (Maybe Cancel, Fifths, Maybe Mode)
    | Key_2 [(Key_Step, Key_Alter)]
      deriving (Eq, Show)
    -- |
read_Key_ :: Eq i => STM Result [Content i] Key_
read_Key_ =
    (read_Key_aux1 >>= return . Key_1) `mplus`
    (read_LIST read_Key_aux2 >>= return . Key_2)
    -- |
show_Key_ :: Key_ -> [Content ()]
show_Key_ (Key_1 (a, b, c)) = show_MAYBE show_Cancel a ++ show_Fifths b ++
    show_MAYBE show_Mode c
show_Key_ (Key_2 a) = show_LIST show_Key_aux1 a
    -- |
read_Key_aux1 :: Eq i => STM Result [Content i] (Maybe Cancel, Fifths, Maybe Mode)
read_Key_aux1 = do
    y1 <- read_MAYBE read_Cancel
    y2 <- read_Fifths
    y3 <- read_MAYBE read_Mode
    return (y1, y2, y3)
read_Key_aux2 :: Eq i => STM Result [Content i] (Key_Step, Key_Alter)
read_Key_aux2 = do
    y1 <- read_Key_Step
    y2 <- read_Key_Alter
    return (y1, y2)
    -- |
show_Key_aux1 :: (Key_Step, Key_Alter) -> [Content ()]
show_Key_aux1 (a, b) = show_Key_Step a ++ show_Key_Alter b
    -- |
type Cancel = (Maybe Left_Right, PCDATA)
    -- |
read_Cancel :: STM Result [Content i] Cancel
read_Cancel = do
    y <- read_ELEMENT "cancel"
    y1 <- read_1 (read_IMPLIED "location" read_Left_Right) (attributes y)
    y2 <- read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Cancel :: Cancel -> [Content ()]
show_Cancel (a, b) =
    show_ELEMENT "cancel" (show_IMPLIED "location" show_Left_Right a)
      (show_PCDATA b)
    -- |
type Fifths = PCDATA
    -- |
read_Fifths :: STM Result [Content i] Fifths
read_Fifths = do
    y <- read_ELEMENT "fifths"
    read_1 read_PCDATA (childs y)
    -- |
show_Fifths :: Fifths -> [Content ()]
show_Fifths a = show_ELEMENT "fifths" [] (show_PCDATA a)
    -- |
type Mode = PCDATA
```

```
  -- |
read_Mode :: STM Result [Content i] Mode
read_Mode = do
  y ← read_ELEMENT "mode"
  read_1 read_PCDATA (childs y)
  -- |
show_Mode :: Mode → [Content ()]
show_Mode a = show_ELEMENT "mode" [] (show_PCDATA a)
  -- |
type Key_Step = PCDATA
  -- |
read_Key_Step :: STM Result [Content i] Key_Step
read_Key_Step = do
  y ← read_ELEMENT "key-step"
  read_1 read_PCDATA (childs y)
  -- |
show_Key_Step :: Key_Step → [Content ()]
show_Key_Step a = show_ELEMENT "key-step" [] (show_PCDATA a)
  -- |
type Key_Alter = PCDATA
  -- |
read_Key_Alter :: STM Result [Content i] Key_Alter
read_Key_Alter = do
  y ← read_ELEMENT "key-alter"
  read_1 read_PCDATA (childs y)
  -- |
show_Key_Alter :: Key_Alter → [Content ()]
show_Key_Alter a = show_ELEMENT "key-alter" [] (show_PCDATA a)
  -- |
type Key_Octave = ((CDATA, Maybe Yes_No), PCDATA)
  -- |
read_Key_Octave :: STM Result [Content i] Key_Octave
read_Key_Octave = do
  y ← read_ELEMENT "key-octave"
  y1 ← read_2 (read_REQUIRED "number" read_CDATA)
    (read_IMPLIED "cancel" read_Yes_No) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Key_Octave :: Key_Octave → [Content ()]
show_Key_Octave ((a, b), c) =
  show_ELEMENT "key-octave" (show_REQUIRED "number" show_CDATA a ++
    show_IMPLIED "cancel" show_Yes_No b)
    (show_PCDATA c)
```

Musical notation duration is commonly represented as fractions. The divisions element indicates how many divisions per quarter note are used to indicate a note's duration. For example, if duration = 1 and divisions = 2, this is an eighth note duration. Duration and divisions are used directly for generating sound output, so they must be chosen to take tuplets into account. Using a divisions element lets us use just one number to represent a duration for each note in the score, while retaining the full power of a fractional representation. For maximum compatibility with Standard MIDI Files, the divisions value should not exceed 16383.

```
  -- ** Divisions
  -- |
type Divisions = PCDATA
  -- |
read_Divisions :: STM Result [Content i] Divisions
read_Divisions = do
```

```
  y ← read_ELEMENT "divisions"
  read_1 read_PCDATA (childs y)
  -- |
show_Divisions :: Divisions → [Content ()]
show_Divisions a = show_ELEMENT "divisions" [] (show_PCDATA a)
```

Time signatures are represented by two elements. The beats element indicates the number of beats, as found in the numerator of a time signature. The beat-type element indicates the beat unit, as found in the denominator of a time signature. The symbol attribute is used to indicate another notation beyond a fraction: the common and cut time symbols, as well as a single number with an implied denominator. Normal (a fraction) is the implied symbol type if none is specified. Multiple pairs of beat and beat-type elements are used for composite time signatures with multiple denominators, such as 2/4 + 3/8. A composite such as 3+2/8 requires only one beat/beat-type pair. A senza-misura element explicitly indicates that no time signature is present.

The print-object attribute allows a time signature to be specified but not printed, as is the case for excerpts from the middle of a score. The value is "yes" if not present. The optional number attribute refers to staff numbers within the part, from top to bottom on the system. If absent, the time signature applies to all staves in the part.

```
  -- ** Time
  -- |
type Time = ((Maybe CDATA, Maybe Time_A, Print_Style, Print_Object), Time_B)
  -- |
read_Time :: Eq i ⇒ STM Result [Content i] Time
read_Time = do
  y ← read_ELEMENT "time"
  y1 ← read_4 (read_IMPLIED "number" read_CDATA)
    (read_IMPLIED "symbol" read_Time_A)
    read_Print_Style read_Print_Object (attributes y)
  y2 ← read_1 read_Time_B (childs y)
  return (y1, y2)
  -- |
show_Time :: Time → [Content ()]
show_Time ((a, b, c, d), e) =
  show_ELEMENT "time" (show_IMPLIED "number" show_CDATA a ++
    show_IMPLIED "symbol" show_Time_A b ++
    show_Print_Style c ++ show_Print_Object d)
    (show_Time_B e)
  -- |
data Time_A = Time_1 | Time_2 | Time_3 | Time_4
    deriving (Eq, Show)
  -- |
read_Time_A :: Prelude.String → Result Time_A
read_Time_A "common" = return Time_1
read_Time_A "cut"    = return Time_2
read_Time_A "single-number" = return Time_3
read_Time_A "normal" = return Time_4
read_Time_A x        = fail x
  -- |
show_Time_A :: Time_A → Prelude.String
show_Time_A Time_1 = "common"
show_Time_A Time_2 = "cut"
show_Time_A Time_3 = "single-number"
show_Time_A Time_4 = "normal"
  -- |
data Time_B = Time_5 [(Beats, Beat_Type)]
  | Time_6 Senza_Misura
    deriving (Eq, Show)
  -- |
```

```
read_Time_B :: Eq i ⇒ STM Result [Content i] Time_B
read_Time_B =
  (read_LIST1 read_Time_B_aux1 >>= return · Time_5) `mplus`
  (read_Senza_Misura >>= return · Time_6)
  -- |
show_Time_B :: Time_B → [Content ()]
show_Time_B (Time_5 a) = show_LIST show_Time_B_aux1 a
show_Time_B (Time_6 a) = show_Senza_Misura a
  -- |
read_Time_B_aux1 :: STM Result [Content i] (Beats, Beat_Type)
read_Time_B_aux1 = do
  y1 ← read_Beats
  y2 ← read_Beat_Type
  return (y1, y2)
  -- |
show_Time_B_aux1 :: (Beats, Beat_Type) → [Content ()]
show_Time_B_aux1 (a, b) = show_Beats a ++ show_Beat_Type b
  -- |
type Beats = PCDATA
  -- |
read_Beats :: STM Result [Content i] Beats
read_Beats = do
  y ← read_ELEMENT "beats"
  read_1 read_PCDATA (childs y)
  -- |
show_Beats :: Beats → [Content ()]
show_Beats a = show_ELEMENT "beats" [] (show_PCDATA a)
  -- |
type Beat_Type = PCDATA
  -- |
read_Beat_Type :: STM Result [Content i] Beat_Type
read_Beat_Type = do
  y ← read_ELEMENT "beat-type"
  read_1 read_PCDATA (childs y)
  -- |
show_Beat_Type :: Beat_Type → [Content ()]
show_Beat_Type a = show_ELEMENT "beat-type" [] (show_PCDATA a)
  -- |
type Senza_Misura = ()
  -- |
read_Senza_Misura :: STM Result [Content i] Senza_Misura
read_Senza_Misura = do
  read_ELEMENT "senza-misura" >> return ()
  -- |
show_Senza_Misura :: Senza_Misura → [Content ()]
show_Senza_Misura _ = show_ELEMENT "senza-misura" [] []
```

Staves are used if there is more than one staff represented in the given part (e.g., 2 staves for typical piano parts). If absent, a value of 1 is assumed. Staves are ordered from top to bottom in a part in numerical order, with staff 1 above staff 2.

```
  -- ** Staves
  -- |
type Staves = PCDATA
  -- |
read_Staves :: STM Result [Content i] Staves
read_Staves = do
  y ← read_ELEMENT "staves"
  read_1 read_PCDATA (childs y)
```

```
  -- |
show_Staves :: Staves → [Content ()]
show_Staves a = show_ELEMENT "staves" [] (show_PCDATA a)
```

The part-symbol element indicates how a symbol for a multi-staff part is indicated in the score. Values include none, brace, line, and bracket; brace is the default. The top-staff and bottom-staff elements are used when the brace does not extend across the entire part. For example, in a 3-staff organ part, the top-staff will typically be 1 for the right hand, while the bottom-staff will typically be 2 for the left hand. Staff 3 for the pedals is usually outside the brace.

```
  -- ** Part_Symbol
  -- |
type Part_Symbol = ((Maybe CDATA, Maybe CDATA, Position, Color), PCDATA)
  -- |
read_Part_Symbol :: STM Result [Content i] Part_Symbol
read_Part_Symbol = do
  y ← read_ELEMENT "part-symbol"
  y1 ← read_4 (read_IMPLIED "top-staff" read_CDATA)
    (read_IMPLIED "bottom-staff" read_CDATA)
    read_Position read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Part_Symbol :: Part_Symbol → [Content ()]
show_Part_Symbol ((a, b, c, d), e) =
  show_ELEMENT "part-symbol" (show_IMPLIED "top-staff" show_CDATA a ++
    show_IMPLIED "bottom-staff" show_CDATA b ++
    show_Position c ++ show_Color d)
    (show_PCDATA e)
```

Instruments are only used if more than one instrument is represented in the part (e.g., oboe I and II where they play together most of the time). If absent, a value of 1 is assumed.

```
  -- ** Instruments
  -- |
type Instruments = PCDATA
  -- |
read_Instruments :: STM Result [Content i] Instruments
read_Instruments = do
  y ← read_ELEMENT "instruments"
  read_1 read_PCDATA (childs y)
  -- |
show_Instruments :: Instruments → [Content ()]
show_Instruments a = show_ELEMENT "instruments" [] (show_PCDATA a)
```

Clefs are represented by the sign, line, and clef-octave-change elements. Sign values include G, F, C, percussion, TAB, and none. Line numbers are counted from the bottom of the staff. Standard values are 2 for the G sign (treble clef), 4 for the F sign (bass clef), 3 for the C sign (alto clef) and 5 for TAB (on a 6-line staff). The clef-octave-change element is used for transposing clefs (e.g., a treble clef for tenors would have a clef-octave-change value of -1). The optional number attribute refers to staff numbers within the part, from top to bottom on the system. A value of 1 is assumed if not present.

Sometimes clefs are added to the staff in non-standard line positions, either to indicate cue passages, or when there are multiple clefs present simultaneously on one staff. In this situation, the additional attribute is set to "yes" and the line value is ignored. The size attribute is used for clefs where the additional attribute is "yes". It is typically used to indicate cue clefs.

```
  -- ** Clef
  -- |
type Clef = ((Maybe CDATA, Maybe Yes_No, Maybe Symbol_Size,
  Print_Style, Print_Object),
```

```
    (Sign, Maybe Line, Maybe Clef_Octave_Change))
  -- |
read_Clef :: Eq i ⇒ STM Result [Content i] Clef
read_Clef = do
  y ← read_ELEMENT "clef"
  y1 ← read_5 (read_IMPLIED "number" read_CDATA)
    (read_IMPLIED "additional" read_Yes_No)
    (read_IMPLIED "size" read_Symbol_Size)
    read_Print_Style read_Print_Object (attributes y)
  y2 ← read_3 read_Sign (read_MAYBE read_Line)
    (read_MAYBE read_Clef_Octave_Change) (childs y)
  return (y1, y2)
  -- |
show_Clef :: Clef → [Content ()]
show_Clef ((a, b, c, d, e), (f, g, h)) =
  show_ELEMENT "clef" (show_IMPLIED "number" show_CDATA a ++
      show_IMPLIED "additional" show_Yes_No b ++
      show_IMPLIED "size" show_Symbol_Size c ++
      show_Print_Style d ++ show_Print_Object e)
    (show_Sign f ++ show_MAYBE show_Line g ++
      show_MAYBE show_Clef_Octave_Change h)
  -- |
type Sign = PCDATA
  -- |
read_Sign :: STM Result [Content i] Sign
read_Sign = do
  y ← read_ELEMENT "sign"
  read_1 read_PCDATA (childs y)
  -- |
show_Sign :: Sign → [Content ()]
show_Sign a = show_ELEMENT "sign" [] (show_PCDATA a)
  -- |
type Line = PCDATA
  -- |
read_Line :: STM Result [Content i] Line
read_Line = do
  y ← read_ELEMENT "line"
  read_1 read_PCDATA (childs y)
  -- |
show_Line :: Line → [Content ()]
show_Line a = show_ELEMENT "line" [] (show_PCDATA a)
  -- |
type Clef_Octave_Change = PCDATA
  -- |
read_Clef_Octave_Change :: STM Result [Content i] Clef_Octave_Change
read_Clef_Octave_Change = do
  y ← read_ELEMENT "clef-octave-change"
  read_1 read_PCDATA (childs y)
  -- |
show_Clef_Octave_Change :: Clef_Octave_Change → [Content ()]
show_Clef_Octave_Change a =
  show_ELEMENT "clef-octave-change" [] (show_PCDATA a)
```

The staff-details element is used to indicate different types of staves. The staff-type element can be ossia, cue, editorial, regular, or alternate. An alternate staff indicates one that shares the same musical data as the prior staff, but displayed differently (e.g., treble and bass clef, standard notation and tab). The staff-lines element specifies the number of lines for non 5-line staffs. The staff-tuning and capo elements are used to specify tuning when using tablature notation. The optional number attribute specifies the staff number from top to bottom on the system, as with clef. The optional show-frets attribute indicates

whether to show tablature frets as numbers (0, 1, 2) or letters (a, b, c). The default choice is numbers. The print-object attribute is used to indicate when a staff is not printed in a part, usually in large scores where empty parts are omitted. It is yes by default. If print-spacing is yes while print-object is no, the score is printed in cutaway format where vertical space is left for the empty part.

```
  -- ** Staff_Details
  -- |
type Staff_Details = ((Maybe CDATA, Maybe Staff_Details_,
  Print_Object, Print_Spacing),
  (Maybe Staff_Type, Maybe Staff_Lines, [Staff_Tuning],
    Maybe Capo, Maybe Staff_Size))
  -- |
read_Staff_Details :: Eq i ⇒ STM Result [Content i] Staff_Details
read_Staff_Details = do
  y ← read_ELEMENT "staff-details"
  y1 ← read_4 (read_IMPLIED "number" read_CDATA)
    (read_IMPLIED "show-frets" read_Staff_Details_)
    read_Print_Object read_Print_Spacing (attributes y)
  y2 ← read_5 (read_MAYBE read_Staff_Type) (read_MAYBE read_Staff_Lines)
    (read_LIST read_Staff_Tuning) (read_MAYBE read_Capo)
    (read_MAYBE read_Staff_Size) (childs y)
  return (y1, y2)
  -- |
show_Staff_Details :: Staff_Details → [Content ()]
show_Staff_Details ((a, b, c, d), (e, f, g, h, i)) =
  show_ELEMENT "staff-details" (show_IMPLIED "number" show_CDATA a ++
      show_IMPLIED "show-frets" show_Staff_Details_ b ++
      show_Print_Object c ++ show_Print_Spacing d)
    (show_MAYBE show_Staff_Type e ++
      show_MAYBE show_Staff_Lines f ++
      show_LIST show_Staff_Tuning g ++
      show_MAYBE show_Capo h ++
      show_MAYBE show_Staff_Size i)
  -- |
data Staff_Details_ = Staff_Details_1 | Staff_Details_2
  deriving (Eq, Show)
  -- |
read_Staff_Details_ :: Prelude.String → Result Staff_Details_
read_Staff_Details_ "numbers" = return Staff_Details_1
read_Staff_Details_ "letters" = return Staff_Details_2
read_Staff_Details_ x = fail x
  -- |
show_Staff_Details_ :: Staff_Details_ → Prelude.String
show_Staff_Details_ Staff_Details_1 = "numbers"
show_Staff_Details_ Staff_Details_2 = "letters"
  -- |
type Staff_Type = PCDATA
  -- |
read_Staff_Type :: STM Result [Content i] Staff_Type
read_Staff_Type = do
  y ← read_ELEMENT "staff-type"
  read_1 read_PCDATA (childs y)
  -- |
show_Staff_Type :: Staff_Type → [Content ()]
show_Staff_Type a = show_ELEMENT "staff-type" [] (show_PCDATA a)
  -- |
type Staff_Lines = PCDATA
  -- |
read_Staff_Lines :: STM Result [Content i] Staff_Lines
```

```
read_Staff_Lines = do
  y ← read_ELEMENT "staff-lines"
  read_1 read_PCDATA (childs y)
  -- |
show_Staff_Lines :: Staff_Lines → [Content ()]
show_Staff_Lines a = show_ELEMENT "staff-lines" [] (show_PCDATA a)
```

The tuning-step, tuning-alter, and tuning-octave elements are defined in the common.mod file. Staff lines are numbered from bottom to top.

```
  -- |
type Staff_Tuning = (CDATA, (Tuning_Step, Maybe Tuning_Alter, Tuning_Octave))
  -- |
read_Staff_Tuning :: Eq i ⇒ STM Result [Content i] Staff_Tuning
read_Staff_Tuning = do
  y ← read_ELEMENT "staff-tuning"
  y1 ← read_1 (read_REQUIRED "line" read_CDATA) (attributes y)
  y2 ← read_3 read_Tuning_Step (read_MAYBE read_Tuning_Alter)
    read_Tuning_Octave (childs y)
  return (y1, y2)
  -- |
show_Staff_Tuning :: Staff_Tuning → [Content ()]
show_Staff_Tuning (a, (b, c, d)) =
  show_ELEMENT "staff-tuning" (show_REQUIRED "line" show_CDATA a)
    (show_Tuning_Step b ++
      show_MAYBE show_Tuning_Alter c ++
      show_Tuning_Octave d)
```

The capo element indicates at which fret a capo should be placed on a fretted instrument. This changes the open tuning of the strings specified by staff-tuning by the specified number of half-steps.

```
  -- |
type Capo = PCDATA
  -- |
read_Capo :: STM Result [Content i] Capo
read_Capo = do
  y ← read_ELEMENT "capo"
  read_1 read_PCDATA (childs y)
  -- |
show_Capo :: Capo → [Content ()]
show_Capo a = show_ELEMENT "capo" [] (show_PCDATA a)
```

The staff-size element indicates how large a staff space is on this staff, expressed as a percentage of the work's default scaling. Values less than 100 make the staff space smaller while values over 100 make the staff space larger. A staff-type of cue, ossia, or editorial implies a staff-size of less than 100, but the exact value is implementation-dependent unless specified here. Staff size affects staff height only, not the relationship of the staff to the left and right margins.

```
  -- |
type Staff_Size = PCDATA
  -- |
read_Staff_Size :: STM Result [Content i] Staff_Size
read_Staff_Size = do
  y ← read_ELEMENT "staff-size"
  read_1 read_PCDATA (childs y)
  -- |
show_Staff_Size :: Staff_Size → [Content ()]
show_Staff_Size a = show_ELEMENT "staff-size" [] (show_PCDATA a)
```

If the part is being encoded for a transposing instrument in written vs. concert pitch, the transposition must be encoded in the transpose element. The transpose element represents what must be added to the written pitch to get the correct sounding pitch.

The transposition is represented by chromatic steps (required) and three optional elements: diatonic pitch steps, octave changes, and doubling an octave down. The chromatic and octave-change elements are numeric values added to the encoded pitch data to create the sounding pitch. The diatonic element is also numeric and allows for correct spelling of enharmonic transpositions.

```
  -- ** Transpose
  -- |
type Transpose = (Maybe Diatonic, Chromatic, Maybe Octave_Change,
  Maybe Double)
  -- |
read_Transpose :: Eq i ⇒ STM Result [Content i] Transpose
read_Transpose = do
  y ← read_ELEMENT "transpose"
  read_4 (read_MAYBE read_Diatonic) read_Chromatic
    (read_MAYBE read_Octave_Change)
    (read_MAYBE read_Double) (childs y)
  -- |
show_Transpose :: Transpose → [Content ()]
show_Transpose (a, b, c, d) =
  show_ELEMENT "transpose" [] (show_MAYBE show_Diatonic a ++
    show_Chromatic b ++
    show_MAYBE show_Octave_Change c ++
    show_MAYBE show_Double d)
  -- |
type Diatonic = PCDATA
  -- |
read_Diatonic :: STM Result [Content i] Diatonic
read_Diatonic = do
  y ← read_ELEMENT "diatonic"
  read_1 read_PCDATA (childs y)
  -- |
show_Diatonic :: Diatonic → [Content ()]
show_Diatonic a = show_ELEMENT "diatonic" [] (show_PCDATA a)
  -- |
type Chromatic = PCDATA
  -- |
read_Chromatic :: STM Result [Content i] Chromatic
read_Chromatic = do
  y ← read_ELEMENT "chromatic"
  read_1 read_PCDATA (childs y)
  -- |
show_Chromatic :: Chromatic → [Content ()]
show_Chromatic a = show_ELEMENT "chromatic" [] (show_PCDATA a)
  -- |
type Octave_Change = PCDATA
  -- |
read_Octave_Change :: STM Result [Content i] Octave_Change
read_Octave_Change = do
  y ← read_ELEMENT "octave-change"
  read_1 read_PCDATA (childs y)
  -- |
show_Octave_Change :: Octave_Change → [Content ()]
show_Octave_Change a = show_ELEMENT "octave-change" [] (show_PCDATA a)
  -- |
type Double = ()
  -- |
read_Double :: STM Result [Content i] Double
read_Double = read_ELEMENT "double" ≫ return ()
  -- |
```

```
show_Double :: Double → [Content ()]
show_Double _ = show_ELEMENT "double" [] []
```

Directives are like directions, but can be grouped together with attributes for convenience. This is typically used for tempo markings at the beginning of a piece of music. This element has been deprecated in Version 2.0 in favor of the directive attribute for direction elements. Language names come from ISO 639, with optional country subcodes from ISO 3166.

```
  -- ** Directive
  -- |
type Directive = ((Print_Style, Maybe CDATA), CDATA)
  -- |
read_Directive :: STM Result [Content i] Directive
read_Directive = do
  y ← read_ELEMENT "directive"
  y1 ← read_2 read_Print_Style
    (read_IMPLIED "xml:lang" read_CDATA) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Directive :: Directive → [Content ()]
show_Directive ((a, b), c) =
  show_ELEMENT "directive" (show_Print_Style a ++
    show_IMPLIED "xml:lang" show_CDATA b)
    (show_PCDATA c)
```

A measure-style indicates a special way to print partial to multiple measures within a part. This includes multiple rests over several measures, repeats of beats, single, or multiple measures, and use of slash notation.

The multiple-rest and measure-repeat symbols indicate the number of measures covered in the element content. The beat-repeat and slash elements can cover partial measures. All but the multiple-rest element use a type attribute to indicate starting and stopping the use of the style. The optional number attribute specifies the staff number from top to bottom on the system, as with clef.

```
  -- ** Measure_Style
  -- |
type Measure_Style = ((Maybe CDATA, Font, Color), Measure_Style_)
  -- |
read_Measure_Style :: Eq i ⇒ STM Result [Content i] Measure_Style
read_Measure_Style = do
  y ← read_ELEMENT "measure-style"
  y1 ← read_3 (read_IMPLIED "number" read_CDATA)
    read_Font read_Color (attributes y)
  y2 ← read_1 read_Measure_Style_ (childs y)
  return (y1, y2)
  -- |
show_Measure_Style :: Measure_Style → [Content ()]
show_Measure_Style ((a, b, c), d) =
  show_ELEMENT "measure-style" (show_IMPLIED "number" show_CDATA a ++
    show_Font b ++ show_Color c)
    (show_Measure_Style_ d)
  -- |
data Measure_Style_ = Measure_Style_1 Multiple_Rest
  | Measure_Style_2 Measure_Repeat
  | Measure_Style_3 Beat_Repeat
  | Measure_Style_4 Slash
    deriving (Eq, Show)
  -- |
read_Measure_Style_ :: Eq i ⇒ STM Result [Content i] Measure_Style_
read_Measure_Style_ =
```

```
    (read_Multiple_Rest >>= return . Measure_Style_1) `mplus`
    (read_Measure_Repeat >>= return . Measure_Style_2) `mplus`
    (read_Beat_Repeat >>= return . Measure_Style_3) `mplus`
    (read_Slash >>= return . Measure_Style_4)
  -- |
show_Measure_Style_ :: Measure_Style_ -> [Content ()]
show_Measure_Style_ (Measure_Style_1 a) = show_Multiple_Rest a
show_Measure_Style_ (Measure_Style_2 a) = show_Measure_Repeat a
show_Measure_Style_ (Measure_Style_3 a) = show_Beat_Repeat a
show_Measure_Style_ (Measure_Style_4 a) = show_Slash a
```

The slash-type and slash-dot elements are optional children of the beat-repeat and slash elements. They have the same values as the type and dot elements, and define what the beat is for the display of repetition marks. If not present, the beat is based on the current time signature.

```
  -- |
type Slash_Type = PCDATA
  -- |
read_Slash_Type :: STM Result [Content i] Slash_Type
read_Slash_Type = do
  y <- read_ELEMENT "slash-type"
  read_1 read_PCDATA (childs y)
  -- |
show_Slash_Type :: Slash_Type -> [Content ()]
show_Slash_Type a = show_ELEMENT "slash-type" [] (show_PCDATA a)
  -- |
type Slash_Dot = ()
  -- |
read_Slash_Dot :: STM Result [Content i] Slash_Dot
read_Slash_Dot = read_ELEMENT "slash-dot" >> return ()
  -- |
show_Slash_Dot :: Slash_Dot -> [Content ()]
show_Slash_Dot _ = show_ELEMENT "slash-dot" [] []
```

The text of the multiple-rest element indicates the number of measures in the multiple rest. Multiple rests may use the 1-bar / 2-bar / 4-bar rest symbols, or a single shape. The use-symbols attribute indicates which to use; it is no if not specified.

```
  -- |
type Multiple_Rest = (Maybe Yes_No, PCDATA)
  -- |
read_Multiple_Rest :: STM Result [Content i] Multiple_Rest
read_Multiple_Rest = do
  y <- read_ELEMENT "multiple-rest"
  y1 <- read_1 (read_IMPLIED "use-symbols" read_Yes_No) (attributes y)
  y2 <- read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Multiple_Rest :: Multiple_Rest -> [Content ()]
show_Multiple_Rest (a, b) =
  show_ELEMENT "multiple-rest" (show_IMPLIED "use-symbols" show_Yes_No a)
    (show_PCDATA b)
```

The measure-repeat and beat-repeat element specify a notation style for repetitions. The actual music being repeated needs to be repeated within the MusicXML file. These elements specify the notation that indicates the repeat.

The measure-repeat element is used for both single and multiple measure repeats. The text of the element indicates the number of measures to be repeated in a single pattern. The slashes attribute specifies the number of slashes to use in the repeat sign. It is 1 if not specified. Both the start and the stop of the measure-repeat must be specified.

```
  -- |
type Measure_Repeat = ((Start_Stop, Maybe CDATA), PCDATA)
  -- |
read_Measure_Repeat :: STM Result [Content i] Measure_Repeat
read_Measure_Repeat = do
  y ← read_ELEMENT "measure-repeat"
  y1 ← read_2 (read_REQUIRED "type" read_Start_Stop)
    (read_IMPLIED "slashes" read_CDATA)
    (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Measure_Repeat :: Measure_Repeat → [Content ()]
show_Measure_Repeat ((a, b), c) =
  show_ELEMENT "measure-repeat"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_IMPLIED "slashes" show_CDATA b)
    (show_PCDATA c)
```

The beat-repeat element is used to indicate that a single beat (but possibly many notes) is repeated. Both the start and stop of the beat being repeated should be specified. The slashes attribute specifies the number of slashes to use in the symbol. The use-dots attribute indicates whether or not to use dots as well (for instance, with mixed rhythm patterns). By default, the value for slashes is 1 and the value for use-dots is no.

```
  -- |
type Beat_Repeat = ((Start_Stop, Maybe CDATA, Maybe Yes_No),
  Maybe (Slash_Type, [Slash_Dot]))
  -- |
read_Beat_Repeat :: Eq i ⇒ STM Result [Content i] Beat_Repeat
read_Beat_Repeat = do
  y ← read_ELEMENT "beat-repeat"
  y1 ← read_3 (read_REQUIRED "type" read_Start_Stop)
    (read_IMPLIED "slashes" read_CDATA)
    (read_IMPLIED "use-dots" read_Yes_No)
    (attributes y)
  y2 ← read_1 (read_MAYBE read_Beat_Repeat_aux1) (childs y)
  return (y1, y2)
  -- |
show_Beat_Repeat :: Beat_Repeat → [Content ()]
show_Beat_Repeat ((a, b, c), d) =
  show_ELEMENT "beat-repeat" (show_REQUIRED "type" show_Start_Stop a ++
    show_IMPLIED "slashes" show_CDATA b ++
    show_IMPLIED "use-dots" show_Yes_No c)
    (show_MAYBE show_Beat_Repeat_aux1 d)
  -- |
read_Beat_Repeat_aux1 :: Eq i ⇒ STM Result [Content i] (Slash_Type, [Slash_Dot])
read_Beat_Repeat_aux1 = do
  y1 ← read_Slash_Type
  y2 ← read_LIST read_Slash_Dot
  return (y1, y2)
  -- |
show_Beat_Repeat_aux1 :: (Slash_Type, [Slash_Dot]) → [Content ()]
show_Beat_Repeat_aux1 (a, b) =
  show_Slash_Type a ++ show_LIST show_Slash_Dot b
```

The slash element is used to indicate that slash notation is to be used. If the slash is on every beat, use-stems is no (the default). To indicate rhythms but not pitches, use-stems is set to yes. The type attribute indicates whether this is the start or stop of a slash notation style. The use-dots attribute works as for the beat-repeat element, and only has effect if use-stems is no.

```
  -- |
type Slash = ((Start_Stop, Maybe Yes_No, Maybe Yes_No),
  Maybe (Slash_Type, [Slash_Dot]))
  -- |
read_Slash :: Eq i ⇒ STM Result [Content i] Slash
read_Slash = do
  y ← read_ELEMENT "slash"
  y1 ← read_3 (read_REQUIRED "type" read_Start_Stop)
    (read_IMPLIED "use-dots" read_Yes_No)
    (read_IMPLIED "use-stems" read_Yes_No)
    (attributes y)
  y2 ← read_1 (read_MAYBE read_Slash_aux1) (childs y)
  return (y1, y2)
  -- |
show_Slash :: Slash → [Content ()]
show_Slash ((a, b, c), d) =
  show_ELEMENT "slash" (show_REQUIRED "type" show_Start_Stop a ++
    show_IMPLIED "use-dots" show_Yes_No b ++
    show_IMPLIED "use-stems" show_Yes_No c)
    (show_MAYBE show_Slash_aux1 d)
  -- |
read_Slash_aux1 :: Eq i ⇒ STM Result [Content i] (Slash_Type, [Slash_Dot])
read_Slash_aux1 = do
  y1 ← read_Slash_Type
  y2 ← read_LIST read_Slash_Dot
  return (y1, y2)
  -- |
show_Slash_aux1 :: (Slash_Type, [Slash_Dot]) → [Content ()]
show_Slash_aux1 (a, b) = show_Slash_Type a ++ show_LIST show_Slash_Dot b
```

2.2 Barline

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Barline where
import Text.XML.MusicXML.Common
import Text.XML.HaXml.Types (Content)
import Prelude (Maybe, Show, Eq, Monad (..), String, (++))
```

If a barline is other than a normal single barline, it should be represented by a barline element that describes it. This includes information about repeats and multiple endings, as well as line style. Barline data is on the same level as the other musical data in a score - a child of a measure in a partwise score, or a part in a timewise score. This allows for barlines within measures, as in dotted barlines that subdivide measures in complex meters. The two fermata elements allow for fermatas on both sides of the barline (the lower one inverted).

Barlines have a location attribute to make it easier to process barlines independently of the other musical data in a score. It is often easier to set up measures separately from entering notes. The location attribute must match where the barline element occurs within the rest of the musical data in the score. If location is left, it should be the first element in the measure, aside from the print, bookmark, and link elements. If location is right, it should be the last element, again with the possible exception of the print, bookmark, and link elements. If no location is specified, the right barline is the default. The segno, coda,

and divisions attributes work the same way as in the sound element defined in the direction.mod file. They are used for playback when barline elements contain segno or coda child elements.

```
  -- * Barline
  -- |
type Barline = ((Barline_, Maybe CDATA, Maybe CDATA, Maybe CDATA),
  (Maybe Bar_Style, Editorial, Maybe Wavy_Line,
  Maybe Segno, Maybe Coda, Maybe (Fermata, Maybe Fermata),
  Maybe Ending, Maybe Repeat))
  -- |
read_Barline :: Eq i => STM Result [Content i] Barline
read_Barline = do
  y <- read_ELEMENT "barline"
  y1 <- read_4 (read_DEFAULT "location" read_Barline_ Barline_1)
    (read_IMPLIED "segno" read_CDATA)
    (read_IMPLIED "coda" read_CDATA)
    (read_IMPLIED "divisions" read_CDATA) (attributes y)
  y2 <- read_8 (read_MAYBE read_Bar_Style) read_Editorial
    (read_MAYBE read_Wavy_Line) (read_MAYBE read_Segno)
    (read_MAYBE read_Coda) (read_MAYBE read_Barline_aux1)
    (read_MAYBE read_Ending) (read_MAYBE read_Repeat)
    (childs y)
  return (y1, y2)
  -- |
show_Barline :: Barline -> [Content ()]
show_Barline ((a, b, c, d), (e, f, g, h, i, j, k, l)) =
  show_ELEMENT "barline" (show_DEFAULT "location" show_Barline_ a ++
      show_IMPLIED "segno" show_CDATA b ++
      show_IMPLIED "coda" show_CDATA c ++
      show_IMPLIED "divisions" show_CDATA d)
    (show_MAYBE show_Bar_Style e ++ show_Editorial f ++
      show_MAYBE show_Wavy_Line g ++
      show_MAYBE show_Segno h ++ show_MAYBE show_Coda i ++
      show_MAYBE show_Barline_aux1 j ++
      show_MAYBE show_Ending k ++ show_MAYBE show_Repeat l)
  -- |
read_Barline_aux1 :: STM Result [Content i] (Fermata, Maybe Fermata)
read_Barline_aux1 = do
  y1 <- read_Fermata
  y2 <- read_MAYBE read_Fermata
  return (y1, y2)
  -- |
show_Barline_aux1 :: (Fermata, Maybe Fermata) -> [Content ()]
show_Barline_aux1 (a, b) = show_Fermata a ++ show_MAYBE show_Fermata b
  -- |
data Barline_ = Barline_1 | Barline_2 | Barline_3
  deriving (Eq, Show)
  -- |
read_Barline_ :: Prelude.String -> Result Barline_
read_Barline_ "right" = return Barline_1
read_Barline_ "left"  = return Barline_2
read_Barline_ "middle" = return Barline_3
read_Barline_ x       = fail x
  -- |
show_Barline_ :: Barline_ -> Prelude.String
show_Barline_ Barline_1 = "right"
show_Barline_ Barline_2 = "left"
show_Barline_ Barline_3 = "middle"
```

Bar-style contains style information. Choices are regular, dotted, dashed, heavy, light-light, light-heavy, heavy-light, heavy-heavy, tick (a short stroke through the top line), short (a partial barline between the 2nd and 4th lines), and none.

```
  -- ** Bar_Style
  -- |
type Bar_Style = (Color, PCDATA)
  -- |
read_Bar_Style :: STM Result [Content i] Bar_Style
read_Bar_Style = do
  y ← read_ELEMENT "bar-style"
  y1 ← read_1 read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Bar_Style :: Bar_Style → [Content ()]
show_Bar_Style (a, b) =
  show_ELEMENT "bar-style" (show_Color a) (show_PCDATA b)
```

The voice entity and the wavy-line, segno, and fermata elements are defined in the common.mod file. They can apply to both notes and barlines.

Endings refers to multiple (e.g. first and second) endings. Typically, the start type is associated with the left barline of the first measure in an ending. The stop and discontinue types are associated with the right barline of the last measure in an ending. Stop is used when the ending mark concludes with a downward jog, as is typical for first endings. Discontinue is used when there is no downward jog, as is typical for second endings that do not conclude a piece. The length of the jog can be specified using the end-length attribute. The text-x and text-y attributes are offsets that specify where the baseline of the start of the ending text appears, relative to the start of the ending line.

The number attribute reflects the numeric values of what is under the ending line. Single endings such as "1" or comma-separated multiple endings such as "1, 2" may be used. The ending element text is used when the text displayed in the ending is different than what appears in the number attribute. The print-object element is used to indicate when an ending is present but not printed, as is often the case for many parts in a full score.

```
  -- ** Ending
  -- |
type Ending = ((CDATA, Ending_, Print_Object, Print_Style,
  Maybe Tenths, Maybe Tenths, Maybe Tenths), PCDATA)
  -- |
read_Ending :: Eq i ⇒ STM Result [Content i] Ending
read_Ending = do
  y ← read_ELEMENT "ending"
  y1 ← read_7 (read_REQUIRED "number" read_CDATA)
    (read_REQUIRED "type" read_Ending_)
    read_Print_Object read_Print_Style
    (read_IMPLIED "end-length" read_Tenths)
    (read_IMPLIED "text-x" read_Tenths)
    (read_IMPLIED "text-y" read_Tenths)
    (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Ending :: Ending → [Content ()]
show_Ending ((a, b, c, d, e, f, g), h) =
  show_ELEMENT "ending" (show_REQUIRED "number" show_CDATA a ++
                         show_REQUIRED "type" show_Ending_ b ++
                         show_Print_Object c ++ show_Print_Style d ++
                         show_IMPLIED "end-length" show_Tenths e ++
                         show_IMPLIED "text-x" show_Tenths f ++
                         show_IMPLIED "text-y" show_Tenths g)
```

```
    (show_PCDATA h)
  -- |
data Ending_ = Ending_1 | Ending_2 | Ending_3
  deriving (Eq, Show)
  -- |
read_Ending_ :: Prelude.String → Result Ending_
read_Ending_ "start" = return Ending_1
read_Ending_ "stop"  = return Ending_2
read_Ending_ "discontinue" = return Ending_3
read_Ending_ x           = fail x
  -- |
show_Ending_ :: Ending_ → Prelude.String
show_Ending_ Ending_1 = "start"
show_Ending_ Ending_2 = "stop"
show_Ending_ Ending_3 = "discontinue"
```

Repeat marks. The start of the repeat has a forward direction while the end of the repeat has a backward direction. Backward repeats that are not part of an ending can use the times attribute to indicate the number of times the repeated section is played.

```
  -- ** Repeat
  -- |
type Repeat = ((Repeat_, Maybe CDATA), ())
  -- |
read_Repeat :: STM Result [Content i] Repeat
read_Repeat = do
  y ← read_ELEMENT "repeat"
  y1 ← read_2 (read_REQUIRED "direction" read_Repeat_)
    (read_IMPLIED "times" read_CDATA) (attributes y)
  return (y1, ())
  -- |
show_Repeat :: Repeat → [Content ()]
show_Repeat ((a, b), _) =
  show_ELEMENT "repeat" (show_REQUIRED "direction" show_Repeat_ a ++
    show_IMPLIED "times" show_CDATA b) []
  -- |
data Repeat_ = Repeat_1 | Repeat_2
  deriving (Eq, Show)
  -- |
read_Repeat_ :: Prelude.String → Result Repeat_
read_Repeat_ "backward" = return Repeat_1
read_Repeat_ "forward" = return Repeat_2
read_Repeat_ x           = fail x
  -- |
show_Repeat_ :: Repeat_ → Prelude.String
show_Repeat_ Repeat_1 = "backward"
show_Repeat_ Repeat_2 = "forward"
```

2.3 Common

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
```

```
module Text.XML.MusicXML.Common (
  module Text.XML.MusicXML.Util,
  module Text.XML.MusicXML.Common)
  where
import Text.XML.MusicXML.Util
import Control.Monad (MonadPlus (..))
import Prelude (Maybe (..), Bool (..), Show, Eq,
  Monad (..), Int, (++), (.))
import qualified Data.Char (String)
import Text.XML.HaXml.Types (Attribute, Content (..))
```

This file contains entities and elements that are common across multiple DTD modules. In particular, several elements here are common across both notes and measures.

If greater ASCII compatibility is desired, entity references may be used instead of the direct Unicode characters. Currently we include ISO Latin-1 for Western European characters and ISO Latin-2 for Central European characters. These files are local copies of the W3C entities located at:

http://www.w3.org/2003/entities/

Data type entities. The ones that resolve to strings show intent for how data is formatted and used.

Calendar dates are represented yyyy-mm-dd format, following ISO 8601.

```
  -- * Entities
  -- |
type YYYY_MM_DD = PCDATA
  -- |
read_YYYY_MM_DD :: STM Result [Content i] YYYY_MM_DD
read_YYYY_MM_DD = read_PCDATA
  -- |
show_YYYY_MM_DD :: YYYY_MM_DD -> [Content ()]
show_YYYY_MM_DD = show_PCDATA
```

The tenths entity is a number representing tenths of interline space (positive or negative) for use in attributes. The layout-tenths entity is the same for use in elements. Both integer and decimal values are allowed, such as 5 for a half space and 2.5 for a quarter space. Interline space is measured from the middle of a staff line.

```
  -- |
type Tenths = CDATA
  -- |
read_Tenths :: Data.Char.String -> Result Tenths
read_Tenths = read_CDATA
  -- |
show_Tenths :: Tenths -> Data.Char.String
show_Tenths = show_CDATA
  -- |
type Layout_Tenths = PCDATA
  -- |
read_Layout_Tenths :: STM Result [Content i] Layout_Tenths
read_Layout_Tenths = read_PCDATA
  -- |
show_Layout_Tenths :: Layout_Tenths -> [Content ()]
show_Layout_Tenths = show_PCDATA
```

The start-stop and start-stop-continue entities are used for musical elements that can either start or stop, such as slurs, tuplets, and wedges. The start-stop-continue entity is used when there is a need to refer to an intermediate point in the symbol, as for complex slurs. The start-stop-single entity is used when the same element is used for multi-note and single-note notations, as for tremolos.

```
  -- |
data Start_Stop = Start_Stop_1 | Start_Stop_2
  deriving (Eq, Show)
```

```
  -- |
read_Start_Stop :: Data.Char.String → Result Start_Stop
read_Start_Stop "start" = return Start_Stop_1
read_Start_Stop "stop" = return Start_Stop_2
read_Start_Stop _       =
  fail "wrong value at start-stop entity"
  -- |
show_Start_Stop :: Start_Stop → Data.Char.String
show_Start_Stop Start_Stop_1 = "start"
show_Start_Stop Start_Stop_2 = "stop"
  -- |
data Start_Stop_Continue = Start_Stop_Continue_1
     | Start_Stop_Continue_2
     | Start_Stop_Continue_3
    deriving (Eq, Show)
  -- |
read_Start_Stop_Continue :: Data.Char.String → Result Start_Stop_Continue
read_Start_Stop_Continue "start" = return Start_Stop_Continue_1
read_Start_Stop_Continue "stop"  = return Start_Stop_Continue_2
read_Start_Stop_Continue "continue" = return Start_Stop_Continue_3
read_Start_Stop_Continue _       =
  fail "wrong value at start-stop-continue entity"
  -- |
show_Start_Stop_Continue :: Start_Stop_Continue → Data.Char.String
show_Start_Stop_Continue Start_Stop_Continue_1 = "start"
show_Start_Stop_Continue Start_Stop_Continue_2 = "stop"
show_Start_Stop_Continue Start_Stop_Continue_3 = "continue"
data Start_Stop_Single = Start_Stop_Single_1
   | Start_Stop_Single_2
   | Start_Stop_Single_3
     deriving (Eq, Show)
  -- |
read_Start_Stop_Single :: Data.Char.String → Result Start_Stop_Single
read_Start_Stop_Single "start" = return Start_Stop_Single_1
read_Start_Stop_Single "stop"  = return Start_Stop_Single_2
read_Start_Stop_Single "single" = return Start_Stop_Single_3
read_Start_Stop_Single _       =
  fail "wrong value at start-stop-single entity"
  -- |
show_Start_Stop_Single :: Start_Stop_Single → Data.Char.String
show_Start_Stop_Single Start_Stop_Single_1 = "start"
show_Start_Stop_Single Start_Stop_Single_2 = "stop"
show_Start_Stop_Single Start_Stop_Single_3 = "single"
```

The yes-no entity is used for boolean-like attributes.

```
  -- | The yes-no entity is used for boolean-like attributes.
type Yes_No = Bool
  -- |
read_Yes_No :: Data.Char.String → Result Yes_No
read_Yes_No "yes" = return True
read_Yes_No "no"  = return False
read_Yes_No str   = fail str
  -- |
show_Yes_No :: Yes_No → Data.Char.String
show_Yes_No True = "yes"
show_Yes_No False = "no"
```

The yes-no-number entity is used for attributes that can be either boolean or numeric values. Values can be "yes", "no", or numbers.

```
  -- |
type Yes_No_Number = CDATA
  -- |
read_Yes_No_Number :: Data.Char.String → Result Yes_No_Number
read_Yes_No_Number = read_CDATA
  -- |
show_Yes_No_Number :: Yes_No_Number → Data.Char.String
show_Yes_No_Number = show_CDATA
```

The symbol-size entity is used to indicate full vs. cue-sized vs. oversized symbols. The large value for oversized symbols was added in version 1.1.

```
  -- |
data Symbol_Size = Symbol_Size_1
   | Symbol_Size_2
   | Symbol_Size_3
     deriving (Eq, Show)
  -- |
read_Symbol_Size :: Data.Char.String → Result Symbol_Size
read_Symbol_Size "full" = return Symbol_Size_1
read_Symbol_Size "cue"  = return Symbol_Size_2
read_Symbol_Size "large" = return Symbol_Size_3
read_Symbol_Size _      =
  fail "wrong value at symbol-size entity"
  -- |
show_Symbol_Size :: Symbol_Size → Data.Char.String
show_Symbol_Size Symbol_Size_1 = "full"
show_Symbol_Size Symbol_Size_2 = "cue"
show_Symbol_Size Symbol_Size_3 = "large"
```

The up-down entity is used for arrow direction, indicating which way the tip is pointing.

```
  -- |
data Up_Down = Up_Down_1 | Up_Down_2
  deriving (Eq, Show)
  -- |
read_Up_Down :: Data.Char.String → Result Up_Down
read_Up_Down "up" = return Up_Down_1
read_Up_Down "down" = return Up_Down_2
read_Up_Down _ =
  fail "wrong value at up-down entity"
  -- |
show_Up_Down :: Up_Down → Data.Char.String
show_Up_Down Up_Down_1 = "up"
show_Up_Down Up_Down_2 = "down"
```

The top-bottom entity is used to indicate the top or bottom part of a vertical shape like non-arpeggiate.

```
  -- |
data Top_Bottom = Top_Bottom_1
   | Top_Bottom_2
     deriving (Eq, Show)
  -- |
read_Top_Bottom :: Data.Char.String → Result Top_Bottom
read_Top_Bottom "top" = return Top_Bottom_1
read_Top_Bottom "bottom" = return Top_Bottom_2
read_Top_Bottom _      =
  fail "wrong value at top-bottom entity"
  -- |
```

```
show_Top_Bottom :: Top_Bottom → Data.Char.String
show_Top_Bottom Top_Bottom_1 = "top"
show_Top_Bottom Top_Bottom_2 = "bottom"
```

The left-right entity is used to indicate whether one element appears to the left or the right of another element.

```
  -- |
data Left_Right = Left_Right_1 | Left_Right_2
  deriving (Eq, Show)
  -- |
read_Left_Right :: Data.Char.String → Result Left_Right
read_Left_Right "left" = return Left_Right_1
read_Left_Right "right" = return Left_Right_2
read_Left_Right _       =
  fail "wrong value at left-right entity"
  -- |
show_Left_Right :: Left_Right → Data.Char.String
show_Left_Right Left_Right_1 = "left"
show_Left_Right Left_Right_2 = "right"
```

The number-of-lines entity is used to specify the number of lines in text decoration attributes.

```
  -- |
data Number_Of_Lines = Number_Of_Lines_0
   | Number_Of_Lines_1
   | Number_Of_Lines_2
   | Number_Of_Lines_3
  deriving (Eq, Show)
  -- |
read_Number_Of_Lines :: Data.Char.String → Result Number_Of_Lines
read_Number_Of_Lines "0" = return Number_Of_Lines_0
read_Number_Of_Lines "1" = return Number_Of_Lines_1
read_Number_Of_Lines "2" = return Number_Of_Lines_2
read_Number_Of_Lines "3" = return Number_Of_Lines_3
read_Number_Of_Lines _ =
  fail "wrong value at number-of-lines entity"
  -- |
show_Number_Of_Lines :: Number_Of_Lines → Data.Char.String
show_Number_Of_Lines Number_Of_Lines_0 = "0"
show_Number_Of_Lines Number_Of_Lines_1 = "1"
show_Number_Of_Lines Number_Of_Lines_2 = "2"
show_Number_Of_Lines Number_Of_Lines_3 = "3"
```

Slurs, tuplets, and many other features can be concurrent and overlapping within a single musical part. The number-level attribute distinguishes up to six concurrent objects of the same type. A reading program should be prepared to handle cases where the number-levels stop in an arbitrary order. Different numbers are needed when the features overlap in MusicXML file order. When a number-level value is implied, the value is 1 by default.

```
  -- |
data Number_Level = Number_Level_1
   | Number_Level_2
   | Number_Level_3
   | Number_Level_4
   | Number_Level_5
   | Number_Level_6
  deriving (Eq, Show)
  -- |
read_Number_Level :: Data.Char.String → Result Number_Level
```

```
read_Number_Level "1" = return Number_Level_1
read_Number_Level "2" = return Number_Level_2
read_Number_Level "3" = return Number_Level_3
read_Number_Level "4" = return Number_Level_4
read_Number_Level "5" = return Number_Level_5
read_Number_Level "6" = return Number_Level_6
read_Number_Level _ =
  fail "wrong value at number-level entity"
  -- |
show_Number_Level :: Number_Level → Data.Char.String
show_Number_Level Number_Level_1 = "1"
show_Number_Level Number_Level_2 = "2"
show_Number_Level Number_Level_3 = "3"
show_Number_Level Number_Level_4 = "4"
show_Number_Level Number_Level_5 = "5"
show_Number_Level Number_Level_6 = "6"
```

The MusicXML format supports six levels of beaming, up to 256th notes. Unlike the number-level attribute, the beam-level attribute identifies concurrent beams in a beam group. It does not distinguish overlapping beams such as grace notes within regular notes, or beams used in different voices.

```
  -- |
data Beam_Level = Beam_Level_1
  | Beam_Level_2
  | Beam_Level_3
  | Beam_Level_4
  | Beam_Level_5
  | Beam_Level_6
  deriving (Eq, Show)
  -- |
read_Beam_Level :: Data.Char.String → Result Beam_Level
read_Beam_Level "1" = return Beam_Level_1
read_Beam_Level "2" = return Beam_Level_2
read_Beam_Level "3" = return Beam_Level_3
read_Beam_Level "4" = return Beam_Level_4
read_Beam_Level "5" = return Beam_Level_5
read_Beam_Level "6" = return Beam_Level_6
read_Beam_Level _ =
  fail "wrong value at beam-level entity"
  -- |
show_Beam_Level :: Beam_Level → Data.Char.String
show_Beam_Level Beam_Level_1 = "1"
show_Beam_Level Beam_Level_2 = "2"
show_Beam_Level Beam_Level_3 = "3"
show_Beam_Level Beam_Level_4 = "4"
show_Beam_Level Beam_Level_5 = "5"
show_Beam_Level Beam_Level_6 = "6"
```

Common structures for formatting attribute definitions.

The position attributes are based on MuseData print suggestions. For most elements, any program will compute a default x and y position. The position attributes let this be changed two ways.

The default-x and default-y attributes change the computation of the default position. For most elements, the origin is changed relative to the left-hand side of the note or the musical position within the bar (x) and the top line of the staff (y).

For the following elements, the default-x value changes the origin relative to the start of the current measure:

- note - figured-bass - harmony - link - directive - measure-numbering - all descendants of the part-list element - all children of the direction-type element

When the part-name and part-abbreviation elements are used in the print element, the default-x value changes the origin relative to the start of the first measure on the system. These values are used when the current measure or a succeeding measure starts a new system.

For the note, figured-bass, and harmony elements, the default-x value is considered to have adjusted the musical position within the bar for its descendant elements.

Since the credit-words and credit-image elements are not related to a measure, in these cases the default-x and default-y attributes adjust the origin relative to the bottom left-hand corner of the specified page.

The relative-x and relative-y attributes change the position relative to the default position, either as computed by the individual program, or as overridden by the default-x and default-y attributes.

Positive x is right, negative x is left; positive y is up, negative y is down. All units are in tenths of interline space. For stems, positive relative-y lengthens a stem while negative relative-y shortens it.

The default-x and default-y position attributes provide higher-resolution positioning data than related features such as the placement attribute and the offset element. Applications reading a MusicXML file that can understand both features should generally rely on the default-x and default-y attributes for their greater accuracy. For the relative-x and relative-y attributes, the offset element, placement attribute, and directive attribute provide context for the relative position information, so the two features should be interpreted together.

As elsewhere in the MusicXML format, tenths are the global tenths defined by the scaling element, not the local tenths of a staff resized by the staff-size element.

```
  -- * Attributes
  -- |
type Position = (Maybe Tenths, Maybe Tenths, Maybe Tenths, Maybe Tenths)
  -- |
read_Position :: STM Result [Attribute] Position
read_Position = do
  y1 ← read_IMPLIED "default-x" read_Tenths
  y2 ← read_IMPLIED "default-y" read_Tenths
  y3 ← read_IMPLIED "relative-x" read_Tenths
  y4 ← read_IMPLIED "relative-y" read_Tenths
  return (y1, y2, y3, y4)
  -- |
show_Position :: Position → [Attribute]
show_Position (a, b, c, d) =
  show_IMPLIED "default-x" show_Tenths a ++
  show_IMPLIED "default-y" show_Tenths b ++
  show_IMPLIED "relative-x" show_Tenths c ++
  show_IMPLIED "relative-y" show_Tenths d
```

The placement attribute indicates whether something is above or below another element, such as a note or a notation.

```
  -- |
type Placement = Maybe Placement_
  -- |
read_Placement :: STM Result [Attribute] Placement
read_Placement = read_IMPLIED "placement" read_Placement_
  -- |
show_Placement :: Placement → [Attribute]
show_Placement = show_IMPLIED "placement" show_Placement_
  -- |
data Placement_ = Placement_1
  | Placement_2
    deriving (Eq, Show)
  -- |
read_Placement_ :: Data.Char.String → Result Placement_
read_Placement_ "above" = return Placement_1
read_Placement_ "below" = return Placement_2
```

```
read_Placement_ _ =
  fail "wrong value at placement attribute"
  -- |
show_Placement_ :: Placement_ → Data.Char.String
show_Placement_ Placement_1 = "above"
show_Placement_ Placement_2 = "below"
```

The orientation attribute indicates whether slurs and ties are overhand (tips down) or underhand (tips up). This is distinct from the placement entity used by any notation type.

```
  -- |
type Orientation = Maybe Orientation_
  -- |
read_Orientation :: STM Result [Attribute] Orientation
read_Orientation = read_IMPLIED "orientation" read_Orientation_
  -- |
show_Orientation :: Orientation → [Attribute]
show_Orientation = show_IMPLIED "orientation" show_Orientation_
  -- |
data Orientation_ = Orientation_1 | Orientation_2
  deriving (Eq, Show)
  -- |
read_Orientation_ :: Data.Char.String → Result Orientation_
read_Orientation_ "over" = return Orientation_1
read_Orientation_ "under" = return Orientation_2
read_Orientation_ _       =
  fail "wrong value at orientation attribute"
  -- |
show_Orientation_ :: Orientation_ → Data.Char.String
show_Orientation_ Orientation_1 = "over"
show_Orientation_ Orientation_2 = "under"
```

The directive entity changes the default-x position of a direction. It indicates that the left-hand side of the direction is aligned with the left-hand side of the time signature. If no time signature is present, it is aligned with the left-hand side of the first music notational element in the measure. If a default-x, justify, or halign attribute is present, it overrides the directive entity.

```
  -- |
type Directive = Maybe Yes_No
  -- |
read_Directive :: STM Result [Attribute] Directive
read_Directive = read_IMPLIED "directive" read_Yes_No
  -- |
show_Directive :: Directive → [Attribute]
show_Directive = show_IMPLIED "directive" show_Yes_No
```

The bezier entity is used to indicate the curvature of slurs and ties, representing the control points for a cubic bezier curve. For ties, the bezier entity is used with the tied element.

Normal slurs, S-shaped slurs, and ties need only two bezier points: one associated with the start of the slur or tie, the other with the stop. Complex slurs and slurs divided over system breaks can specify additional bezier data at slur elements with a continue type.

The bezier-offset, bezier-x, and bezier-y attributes describe the outgoing bezier point for slurs and ties with a start type, and the incoming bezier point for slurs and ties with types of stop or continue. The attributes bezier-offset2, bezier-x2, and bezier-y2 are only valid with slurs of type continue, and describe the outgoing bezier point.

The bezier-offset and bezier-offset2 attributes are measured in terms of musical divisions, like the offset element. These are the recommended attributes for specifying horizontal position. The other attributes are specified in tenths, relative to any position settings associated with the slur or tied element.

```
  -- |
type Bezier = (Maybe CDATA, Maybe CDATA,
```

```
    Maybe Tenths, Maybe Tenths, Maybe Tenths, Maybe Tenths)
    -- |
read_Bezier :: STM Result [Attribute] Bezier
read_Bezier = do
    y1 ← read_IMPLIED "bezier-offset" read_CDATA
    y2 ← read_IMPLIED "bezier-offset2" read_CDATA
    y3 ← read_IMPLIED "bezier-x" read_Tenths
    y4 ← read_IMPLIED "bezier-y" read_Tenths
    y5 ← read_IMPLIED "bezier-x2" read_Tenths
    y6 ← read_IMPLIED "bezier-y2" read_Tenths
    return (y1, y2, y3, y4, y5, y6)
    -- |
show_Bezier :: Bezier → [Attribute]
show_Bezier (a, b, c, d, e, f) =
    show_IMPLIED "bezier-offset" show_CDATA a ++
    show_IMPLIED "bezier-offset2" show_CDATA b ++
    show_IMPLIED "bezier-x" show_CDATA c ++
    show_IMPLIED "bezier-y" show_CDATA d ++
    show_IMPLIED "bezier-x2" show_CDATA e ++
    show_IMPLIED "bezier-y2" show_CDATA f
```

The font entity gathers together attributes for determining the font within a directive or direction. They are based on the text styles for Cascading Style Sheets. The font-family is a comma-separated list of font names. These can be specific font styles such as Maestro or Opus, or one of several generic font styles: music, serif, sans-serif, handwritten, cursive, fantasy, and monospace. The music and handwritten values refer to music fonts; the rest refer to text fonts. The fantasy style refers to decorative text such as found in older German-style printing. The font-style can be normal or italic. The font-size can be one of the CSS sizes (xx-small, x-small, small, medium, large, x-large, xx-large) or a numeric point size. The font-weight can be normal or bold. The default is application-dependent, but is a text font vs. a music font.

```
    -- |
type Font = (Maybe CDATA, Maybe CDATA, Maybe CDATA, Maybe CDATA)
    -- |
read_Font :: STM Result [Attribute] Font
read_Font = do
    y1 ← read_IMPLIED "font-family" read_CDATA
    y2 ← read_IMPLIED "font-style" read_CDATA
    y3 ← read_IMPLIED "font-size" read_CDATA
    y4 ← read_IMPLIED "font-weight" read_CDATA
    return (y1, y2, y3, y4)
    -- |
show_Font :: Font → [Attribute]
show_Font (a, b, c, d) =
    show_IMPLIED "font-family" show_CDATA a ++
    show_IMPLIED "font-style" show_CDATA b ++
    show_IMPLIED "font-size" show_CDATA c ++
    show_IMPLIED "font-weight" show_CDATA d
```

The color entity indicates the color of an element. Color may be represented as hexadecimal RGB triples, as in HTML, or as hexadecimal ARGB tuples, with the A indicating alpha of transparency. An alpha value of 00 is totally transparent; FF is totally opaque. If RGB is used, the A value is assumed to be FF.

For instance, the RGB value "#800080" represents purple. An ARGB value of "#40800080" would be a transparent purple.

As in SVG 1.1, colors are defined in terms of the sRGB color space (IEC 61966).

```
    -- |
type Color = Maybe CDATA
```

```
  -- |
read_Color :: STM Result [Attribute] Color
read_Color = read_IMPLIED "color" read_CDATA
  -- |
show_Color :: Color → [Attribute]
show_Color = show_IMPLIED "color" show_CDATA
```

The text-decoration entity is based on the similar feature in XHTML and CSS. It allows for text to be underlined, overlined, or struck-through. It extends the CSS version by allow double or triple lines instead of just being on or off.

```
  -- |
type Text_Decoration = (Maybe Number_Of_Lines,
  Maybe Number_Of_Lines,
  Maybe Number_Of_Lines)
  -- |
read_Text_Decoration :: STM Result [Attribute] Text_Decoration
read_Text_Decoration = do
  y1 ← read_IMPLIED "underline" read_Number_Of_Lines
  y2 ← read_IMPLIED "overline" read_Number_Of_Lines
  y3 ← read_IMPLIED "line-through" read_Number_Of_Lines
  return (y1, y2, y3)
  -- |
show_Text_Decoration :: Text_Decoration → [Attribute]
show_Text_Decoration (a, b, c) =
  show_IMPLIED "underline" show_Number_Of_Lines a ++
  show_IMPLIED "overline" show_Number_Of_Lines b ++
  show_IMPLIED "line-through" show_Number_Of_Lines c
```

The justify entity is used to indicate left, center, or right justification. The default value varies for different elements.

```
  -- |
type Justify = Maybe Justify_
  -- |
read_Justify :: STM Result [Attribute] Justify
read_Justify = read_IMPLIED "justify" read_Justify_
  -- |
show_Justify :: Justify → [Attribute]
show_Justify = show_IMPLIED "justify" show_Justify_
  -- |
data Justify_ = Justify_1 | Justify_2 | Justify_3
  deriving (Eq, Show)
  -- |
read_Justify_ :: Data.Char.String → Result Justify_
read_Justify_ "left"   = return Justify_1
read_Justify_ "center" = return Justify_2
read_Justify_ "right"  = return Justify_3
read_Justify_ _        =
  fail "wrong value at justify attribute"
  -- |
show_Justify_ :: Justify_ → Data.Char.String
show_Justify_ Justify_1 = "left"
show_Justify_ Justify_2 = "center"
show_Justify_ Justify_3 = "right"
```

In cases where text extends over more than one line, horizontal alignment and justify values can be different. The most typical case is for credits, such as:

Words and music by Pat Songwriter

Typically this type of credit is aligned to the right, so that the position information refers to the rightmost part of the text. But in this example, the text is center-justified, not right-justified.

The halign attribute is used in these situations. If it is not present, its value is the same as for the justify attribute.

```
  -- |
type Halign = Maybe Halign_
  -- |
read_Halign :: STM Result [Attribute] Halign
read_Halign = read_IMPLIED "halign" read_Halign_
  -- |
show_Halign :: Halign → [Attribute]
show_Halign = show_IMPLIED "halign" show_Halign_
  -- |
data Halign_ = Halign_1 | Halign_2 | Halign_3
  deriving (Eq, Show)
  -- |
read_Halign_ :: Data.Char.String → Result Halign_
read_Halign_ "left"   = return Halign_1
read_Halign_ "center" = return Halign_2
read_Halign_ "right"  = return Halign_3
read_Halign_ _        =
  fail "wrong value at halign attribute"
  -- |
show_Halign_ :: Halign_ → Data.Char.String
show_Halign_ Halign_1 = "left"
show_Halign_ Halign_2 = "center"
show_Halign_ Halign_3 = "right"
```

The valign entity is used to indicate vertical alignment to the top, middle, bottom, or baseline of the text. Defaults are implementation-dependent.

```
  -- |
type Valign = Maybe Valign_
  -- |
read_Valign :: STM Result [Attribute] Valign
read_Valign = read_IMPLIED "valign" read_Valign_
  -- |
show_Valign :: Valign → [Attribute]
show_Valign = show_IMPLIED "valign" show_Valign_
  -- |
data Valign_ = Valign_1 | Valign_2 | Valign_3 | Valign_4
  deriving (Eq, Show)
  -- |
read_Valign_ :: Data.Char.String → Result Valign_
read_Valign_ "top"    = return Valign_1
read_Valign_ "middle" = return Valign_2
read_Valign_ "bottom" = return Valign_3
read_Valign_ "baseline" = return Valign_4
read_Valign_ _        =
  fail "wrong value at valign attribute"
  -- |
show_Valign_ :: Valign_ → Data.Char.String
show_Valign_ Valign_1 = "top"
show_Valign_ Valign_2 = "middle"
show_Valign_ Valign_3 = "bottom"
show_Valign_ Valign_4 = "baseline"
```

The valign-image entity is used to indicate vertical alignment for images and graphics, so it removes the baseline value. Defaults are implementation-dependent.

```
  -- |
type Valign_Image = Maybe Valign_Image_
  -- |
read_Valign_Image :: STM Result [Attribute] Valign_Image
read_Valign_Image = read_IMPLIED "valign-image" read_Valign_Image_
  -- |
show_Valign_Image :: Valign_Image → [Attribute]
show_Valign_Image = show_IMPLIED "valign-image" show_Valign_Image_
  -- |
data Valign_Image_ = Valign_Image_1 | Valign_Image_2 | Valign_Image_3
  deriving (Eq, Show)
  -- |
read_Valign_Image_ :: Data.Char.String → Result Valign_Image_
read_Valign_Image_ "top"    = return Valign_Image_1
read_Valign_Image_ "middle" = return Valign_Image_2
read_Valign_Image_ "bottom" = return Valign_Image_3
read_Valign_Image_ _        =
  fail "wrong value at valign-image attribute"
  -- |
show_Valign_Image_ :: Valign_Image_ → Data.Char.String
show_Valign_Image_ Valign_Image_1 = "top"
show_Valign_Image_ Valign_Image_2 = "middle"
show_Valign_Image_ Valign_Image_3 = "bottom"
```

The letter-spacing entity specifies text tracking. Values are either "normal" or a number representing the number of ems to add between each letter. The number may be negative in order to subtract space. The default is normal, which allows flexibility of letter-spacing for purposes of text justification.

```
  -- |
type Letter_Spacing = Maybe CDATA
  -- |
read_Letter_Spacing :: STM Result [Attribute] Letter_Spacing
read_Letter_Spacing = read_IMPLIED "letter-spacing" read_CDATA
  -- |
show_Letter_Spacing :: Letter_Spacing → [Attribute]
show_Letter_Spacing = show_IMPLIED "letter-spacing" show_CDATA
```

The line-height entity specified text leading. Values are either "normal" or a number representing the percentage of the current font height to use for leading. The default is "normal". The exact normal value is implementation-dependent, but values between 100 and 120 are recommended.

```
  -- |
type Line_Height = Maybe CDATA
  -- |
read_Line_Height :: STM Result [Attribute] Line_Height
read_Line_Height = read_IMPLIED "line-height" read_CDATA
  -- |
show_Line_Height :: Line_Height → [Attribute]
show_Line_Height = show_IMPLIED "line-height" show_CDATA
```

The text-direction entity is used to adjust and override the Unicode bidirectional text algorithm, similar to the W3C Internationalization Tag Set recommendation. Values are ltr (left-to-right embed), rtl (right-to-left embed), lro (left-to-right bidi-override), and rlo (right-to-left bidi-override). The default value is ltr. This entity is typically used by applications that store text in left-to-right visual order rather than logical order. Such applications can use the lro value to better communicate with other applications that more fully support bidirectional text.

```
  -- |
type Text_Direction = Maybe Text_Direction_
  -- |
```

```
read_Text_Direction :: STM Result [Attribute] Text_Direction
read_Text_Direction = read_IMPLIED "dir" read_Text_Direction_
  -- |
show_Text_Direction :: Text_Direction → [Attribute]
show_Text_Direction = show_IMPLIED "dir" show_Text_Direction_
  -- |
data Text_Direction_ = Text_Direction_1
  | Text_Direction_2
  | Text_Direction_3
  | Text_Direction_4
  deriving (Eq, Show)
  -- |
read_Text_Direction_ :: Data.Char.String → Result Text_Direction_
read_Text_Direction_ "ltr" = return Text_Direction_1
read_Text_Direction_ "rtl" = return Text_Direction_2
read_Text_Direction_ "rlo" = return Text_Direction_3
read_Text_Direction_ "lro" = return Text_Direction_4
read_Text_Direction_ _ =
  fail "wrong value at text-direction attribute"
  -- |
show_Text_Direction_ :: Text_Direction_ → Data.Char.String
show_Text_Direction_ Text_Direction_1 = "ltr"
show_Text_Direction_ Text_Direction_2 = "rtl"
show_Text_Direction_ Text_Direction_3 = "rlo"
show_Text_Direction_ Text_Direction_4 = "lro"
```

The text-rotation entity is used to rotate text around the alignment point specified by the halign and valign entities. The value is a number ranging from -180 to 180. Positive values are clockwise rotations, while negative values are counter-clockwise rotations.

```
  -- |
type Text_Rotation = Maybe CDATA
  -- |
read_Text_Rotation :: STM Result [Attribute] Text_Rotation
read_Text_Rotation = read_IMPLIED "text-rotation" read_CDATA
  -- |
show_Text_Rotation :: Text_Rotation → [Attribute]
show_Text_Rotation = show_IMPLIED "text-rotation" show_CDATA
```

The print-style entity groups together the most popular combination of printing attributes: position, font, and color.

```
  -- |
type Print_Style = (Position, Font, Color)
  -- |
read_Print_Style :: STM Result [Attribute] Print_Style
read_Print_Style = do
  y1 ← read_Position
  y2 ← read_Font
  y3 ← read_Color
  return (y1, y2, y3)
  -- |
show_Print_Style :: Print_Style → [Attribute]
show_Print_Style (a, b, c) =
  show_Position a ++ show_Font b ++ show_Color c
```

The line-shape entity is used to distinguish between straight and curved lines. The line-type entity distinguishes between solid, dashed, dotted, and wavy lines.

```
type Line_Shape = Maybe Line_Shape_
  -- |
```

```
read_Line_Shape :: STM Result [Attribute] Line_Shape
read_Line_Shape = read_IMPLIED "line-shape" read_Line_Shape_
  -- |
show_Line_Shape :: Line_Shape → [Attribute]
show_Line_Shape = show_IMPLIED "line-shape" show_Line_Shape_
  -- |
data Line_Shape_ = Line_Shape_1 | Line_Shape_2
  deriving (Eq, Show)
  -- |
read_Line_Shape_ :: Data.Char.String → Result Line_Shape_
read_Line_Shape_ "straight" = return Line_Shape_1
read_Line_Shape_ "curved" = return Line_Shape_2
read_Line_Shape_ _          =
  fail "wrong value at line-shape attribute"
  -- |
show_Line_Shape_ :: Line_Shape_ → Data.Char.String
show_Line_Shape_ Line_Shape_1 = "straight"
show_Line_Shape_ Line_Shape_2 = "curved"
  -- |
type Line_Type = Maybe Line_Type_
  -- |
read_Line_Type :: STM Result [Attribute] Line_Type
read_Line_Type = read_IMPLIED "line-type" read_Line_Type_
  -- |
show_Line_Type :: Line_Type → [Attribute]
show_Line_Type = show_IMPLIED "line-type" show_Line_Type_
  -- |
data Line_Type_ = Line_Type_1 | Line_Type_2 | Line_Type_3 | Line_Type_4
  deriving (Eq, Show)
  -- |
read_Line_Type_ :: Data.Char.String → Result Line_Type_
read_Line_Type_ "solid" = return Line_Type_1
read_Line_Type_ "dashed" = return Line_Type_2
read_Line_Type_ "dotted" = return Line_Type_3
read_Line_Type_ "wavy"  = return Line_Type_4
read_Line_Type_ _         =
  fail "wrong value at line-type attribute"
show_Line_Type_ :: Line_Type_ → Data.Char.String
show_Line_Type_ Line_Type_1 = "solid"
show_Line_Type_ Line_Type_2 = "dashed"
show_Line_Type_ Line_Type_3 = "dotted"
show_Line_Type_ Line_Type_4 = "wavy"
```

The printout entity is based on MuseData print suggestions. They allow a way to specify not to print print an object (e.g. note or rest), its augmentation dots, or its lyrics. This is especially useful for notes that overlap in different voices, or for chord sheets that contain lyrics and chords but no melody. For wholly invisible notes, such as those providing sound-only data, the attribute for print-spacing may be set to no so that no space is left for this note. The print-spacing value is only used if no note, dot, or lyric is being printed.

By default, all these attributes are set to yes. If print-object is set to no, print-dot and print-lyric are interpreted to also be set to no if they are not present.

```
  -- |
type Print_Object = Maybe Yes_No
  -- |
read_Print_Object :: STM Result [Attribute] Print_Object
read_Print_Object = read_IMPLIED "print-object" read_Yes_No
  -- |
show_Print_Object :: Print_Object → [Attribute]
```

```
show_Print_Object = show_IMPLIED "print-object" show_Yes_No
  -- |
type Print_Spacing = Maybe Yes_No
  -- |
read_Print_Spacing :: STM Result [Attribute] Print_Spacing
read_Print_Spacing = read_IMPLIED "print-spacing" read_Yes_No
  -- |
show_Print_Spacing :: Print_Spacing → [Attribute]
show_Print_Spacing = show_IMPLIED "print-spacing" show_Yes_No
  -- |
type Printout = (Print_Object, Maybe Yes_No, Print_Spacing, Maybe Yes_No)
  -- |
read_Printout :: STM Result [Attribute] Printout
read_Printout = do
  y1 ← read_Print_Object
  y2 ← read_IMPLIED "print-dot" read_Yes_No
  y3 ← read_Print_Spacing
  y4 ← read_IMPLIED "print-lyric" read_Yes_No
  return (y1, y2, y3, y4)
  -- |
show_Printout :: Printout → [Attribute]
show_Printout (a, b, c, d) =
  show_Print_Object a ++
  show_IMPLIED "print-dot" show_Yes_No b ++
  show_Print_Spacing c ++
  show_IMPLIED "print-lyric" show_Yes_No d
```

The text-formatting entity contains the common formatting attributes for text elements. Default values may differ across the elements that use this entity.

```
type Text_Formatting = (Justify, Halign, Valign,
  Print_Style, Text_Decoration, Text_Rotation, Letter_Spacing,
  Line_Height, Maybe CDATA, Text_Direction, Maybe Text_Formatting_)
  -- |
read_Text_Formatting :: STM Result [Attribute] Text_Formatting
read_Text_Formatting = do
  y1 ← read_Justify
  y2 ← read_Halign
  y3 ← read_Valign
  y4 ← read_Print_Style
  y5 ← read_Text_Decoration
  y6 ← read_Text_Rotation
  y7 ← read_Letter_Spacing
  y8 ← read_Line_Height
  y9 ← read_IMPLIED "xml:lang" read_CDATA
  y10 ← read_Text_Direction
  y11 ← read_IMPLIED "enclosure" read_Text_Formatting_
  return (y1, y2, y3, y4, y5, y6, y7, y8, y9, y10, y11)
  -- |
show_Text_Formatting :: Text_Formatting → [Attribute]
show_Text_Formatting (a, b, c, d, e, f, g, h, i, j, k) =
  show_Justify a ++
  show_Halign b ++
  show_Valign c ++
  show_Print_Style d ++
  show_Text_Decoration e ++
  show_Text_Rotation f ++
  show_Letter_Spacing g ++
  show_Line_Height h ++
```

```
    show_IMPLIED "xml:lang" show_CDATA i ++
    show_Text_Direction j ++
    show_IMPLIED "enclosure" show_Text_Formatting_ k
    -- |
  data Text_Formatting_ = Text_Formatting_1
    | Text_Formatting_2
    | Text_Formatting_3
      deriving (Eq, Show)
    -- |
  read_Text_Formatting_ :: Data.Char.String → Result Text_Formatting_
  read_Text_Formatting_ "rectangle" = return Text_Formatting_1
  read_Text_Formatting_ "oval" = return Text_Formatting_2
  read_Text_Formatting_ "none" = return Text_Formatting_3
  read_Text_Formatting_ _      =
    fail "wrong value at enclosure attribute"
    -- |
  show_Text_Formatting_ :: Text_Formatting_ → Data.Char.String
  show_Text_Formatting_ Text_Formatting_1 = "rectangle"
  show_Text_Formatting_ Text_Formatting_2 = "oval"
  show_Text_Formatting_ Text_Formatting_3 = "none"
```

The level-display entity allows specification of three common ways to indicate editorial indications: putting parentheses or square brackets around a symbol, or making the symbol a different size. If not specified, they are left to application defaults. It is used by the level and accidental elements.

```
    -- |
  type Level_Display = (Maybe Yes_No, Maybe Yes_No, Maybe Symbol_Size)
    -- |
  read_Level_Display :: STM Result [Attribute] Level_Display
  read_Level_Display = do    -- return (
    y1 ← read_IMPLIED "parentheses" read_Yes_No
    y2 ← read_IMPLIED "braket" read_Yes_No
    y3 ← read_IMPLIED "size" read_Symbol_Size
    return (y1, y2, y3)
    -- |
  show_Level_Display :: Level_Display → [Attribute]
  show_Level_Display (a, b, c) =
    show_IMPLIED "parentheses" show_Yes_No a ++
    show_IMPLIED "braket" show_Yes_No b ++
    show_IMPLIED "size" show_Symbol_Size c
```

Common structures for playback attribute definitions.

The trill-sound entity includes attributes used to guide the sound of trills, mordents, turns, shakes, and wavy lines, based on MuseData sound suggestions. The default choices are:

start-note = "upper" trill-step = "whole" two-note-turn = "none" accelerate = "no" beats = "4" (minimum of "2").

Second-beat and last-beat are percentages for landing on the indicated beat, with defaults of 25 and 75 respectively.

For mordent and inverted-mordent elements, the defaults are different:

The default start-note is "main", not "upper". The default for beats is "3", not "4". The default for second-beat is "12", not "25". The default for last-beat is "24", not "75".

```
    -- * Attributes
    -- |
  type Trill_Sound = (
    Maybe Trill_Sound_A, Maybe Trill_Sound_B, Maybe Trill_Sound_C,
    Maybe Bool, Maybe CDATA, Maybe CDATA, Maybe CDATA)
    -- |
  read_Trill_Sound :: STM Result [Attribute] Trill_Sound
```

```
read_Trill_Sound = do
  y1 ← read_IMPLIED "start-note" read_Trill_Sound_A
  y2 ← read_IMPLIED "trill-step" read_Trill_Sound_B
  y3 ← read_IMPLIED "two-note-turn" read_Trill_Sound_C
  y4 ← read_IMPLIED "accelerate" read_Yes_No
  y5 ← read_IMPLIED "beats" read_CDATA
  y6 ← read_IMPLIED "second-beat" read_CDATA
  y7 ← read_IMPLIED "last-beat" read_CDATA
  return (y1, y2, y3, y4, y5, y6, y7)
  -- |
show_Trill_Sound :: Trill_Sound → [Attribute]
show_Trill_Sound (a, b, c, d, e, f, g) =
  show_IMPLIED "start-note" show_Trill_Sound_A a ++
  show_IMPLIED "trill-step" show_Trill_Sound_B b ++
  show_IMPLIED "two-note-turn" show_Trill_Sound_C c ++
  show_IMPLIED "accelerate" show_Yes_No d ++
  show_IMPLIED "beats" show_CDATA e ++
  show_IMPLIED "second-beat" show_CDATA f ++
  show_IMPLIED "last-beat" show_CDATA g
  -- |
data Trill_Sound_A = Trill_Sound_1 | Trill_Sound_2 | Trill_Sound_3
  deriving (Eq, Show)
  -- |
read_Trill_Sound_A :: Data.Char.String → Result Trill_Sound_A
read_Trill_Sound_A "upper" = return Trill_Sound_1
read_Trill_Sound_A "main" = return Trill_Sound_2
read_Trill_Sound_A "below" = return Trill_Sound_3
read_Trill_Sound_A _       =
  fail "wrong value at start-note attribute"
  -- |
show_Trill_Sound_A :: Trill_Sound_A → Data.Char.String
show_Trill_Sound_A Trill_Sound_1 = "upper"
show_Trill_Sound_A Trill_Sound_2 = "main"
show_Trill_Sound_A Trill_Sound_3 = "below"
  -- |
data Trill_Sound_B = Trill_Sound_4 | Trill_Sound_5 | Trill_Sound_6
  deriving (Eq, Show)
  -- |
read_Trill_Sound_B :: Data.Char.String → Result Trill_Sound_B
read_Trill_Sound_B "whole" = return Trill_Sound_4
read_Trill_Sound_B "half"  = return Trill_Sound_5
read_Trill_Sound_B "unison" = return Trill_Sound_6
read_Trill_Sound_B _       =
  fail "wrong value at trill-step attribute"
  -- |
show_Trill_Sound_B :: Trill_Sound_B → Data.Char.String
show_Trill_Sound_B Trill_Sound_4 = "whole"
show_Trill_Sound_B Trill_Sound_5 = "half"
show_Trill_Sound_B Trill_Sound_6 = "unison"
  -- |
data Trill_Sound_C = Trill_Sound_7 | Trill_Sound_8 | Trill_Sound_9
  deriving (Eq, Show)
  -- |
read_Trill_Sound_C :: Data.Char.String → Result Trill_Sound_C
read_Trill_Sound_C "whole" = return Trill_Sound_7
read_Trill_Sound_C "half" = return Trill_Sound_8
read_Trill_Sound_C "none" = return Trill_Sound_9
read_Trill_Sound_C _       =
```

```
    fail "wrong value at two-note-turn attribute"
    -- |
show_Trill_Sound_C :: Trill_Sound_C → Data.Char.String
show_Trill_Sound_C Trill_Sound_7 = "whole"
show_Trill_Sound_C Trill_Sound_8 = "half"
show_Trill_Sound_C Trill_Sound_9 = "none"
```

The bend-sound entity is used for bend and slide elements, and is similar to the trill-sound. Here the beats element refers to the number of discrete elements (like MIDI pitch bends) used to represent a continuous bend or slide. The first-beat indicates the percentage of the direction for starting a bend; the last-beat the percentage for ending it. The default choices are:
accelerate = "no" beats = "4" (minimum of "2") first-beat = "25" last-beat = "75"

```
    -- |
type Bend_Sound = (Maybe Yes_No, Maybe CDATA, Maybe CDATA, Maybe CDATA)
    -- |
read_Bend_Sound :: STM Result [Attribute] Bend_Sound
read_Bend_Sound = do
    y1 ← read_IMPLIED "accelerate" read_Yes_No
    y2 ← read_IMPLIED "beats" read_CDATA
    y3 ← read_IMPLIED "first-beat" read_CDATA
    y4 ← read_IMPLIED "last-beat" read_CDATA
    return (y1, y2, y3, y4)
    -- |
show_Bend_Sound :: Bend_Sound → [Attribute]
show_Bend_Sound (a, b, c, d) =
    show_IMPLIED "accelerate" show_Yes_No a ++
    show_IMPLIED "beats" show_CDATA b ++
    show_IMPLIED "first-beat" show_CDATA c ++
    show_IMPLIED "second-beat" show_CDATA d
```

Common structures for other attribute definitions.

The document-attributes entity is used to specify the attributes for an entire MusicXML document. Currently this is used for the version attribute.

The version attribute was added in Version 1.1 for the score-partwise and score-timewise documents, and in Version 2.0 for opus documents. It provides an easier way to get version information than through the MusicXML public ID. The default value is 1.0 to make it possible for programs that handle later versions to distinguish earlier version files reliably. Programs that write MusicXML 1.1 or 2.0 files should set this attribute.

```
    -- * Attributes
    -- |
type Document_Attributes = CDATA
    -- |
read_Document_Attributes :: STM Result [Attribute] Document_Attributes
read_Document_Attributes = read_DEFAULT "version" read_CDATA "1.0"
    -- |
show_Document_Attributes :: Document_Attributes → [Attribute]
show_Document_Attributes = show_DEFAULT "version" show_CDATA
```

Common structures for element definitions.

Two entities for editorial information in notes. These entities, and their elements defined below, are used across all the different component DTD modules.

```
    -- * Elements
    -- |
type Editorial = (Maybe Footnote, Maybe Level)
    -- |
read_Editorial :: STM Result [Content i] (Editorial)
read_Editorial = do
```

```
  y1 ← read_MAYBE read_Footnote
  y2 ← read_MAYBE read_Level
  return (y1, y2)
  -- |
show_Editorial :: Editorial → [Content ()]
show_Editorial (a, b) =
  show_MAYBE show_Footnote a ++
  show_MAYBE show_Level b
  -- |
type Editorial_Voice = (Maybe Footnote, Maybe Level, Maybe Voice)
  -- |
read_Editorial_Voice :: STM Result [Content i] Editorial_Voice
read_Editorial_Voice = do
  y1 ← read_MAYBE read_Footnote
  y2 ← read_MAYBE read_Level
  y3 ← read_MAYBE read_Voice
  return (y1, y2, y3)
  -- |
show_Editorial_Voice :: Editorial_Voice → [Content ()]
show_Editorial_Voice (a, b, c) =
  show_MAYBE show_Footnote a ++
  show_MAYBE show_Level b ++
  show_MAYBE show_Voice c
```

Footnote and level are used to specify editorial information, while voice is used to distinguish between multiple voices (what MuseData calls tracks) in individual parts. These elements are used throughout the different MusicXML DTD modules. If the reference attribute for the level element is yes, this indicates editorial information that is for display only and should not affect playback. For instance, a modern edition of older music may set reference="yes" on the attributes containing the music's original clef, key, and time signature. It is no by default.

```
  -- * Elements
  -- |
type Footnote = (Text_Formatting, PCDATA)
  -- |
read_Footnote :: STM Result [Content i] Footnote
read_Footnote = do
  y ← read_ELEMENT "footnote"
  y1 ← read_1 read_Text_Formatting (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Footnote :: Footnote → [Content ()]
show_Footnote (a, b) =
  show_ELEMENT "footnote"
    (show_Text_Formatting a)
    (show_PCDATA b)
  -- |
type Level = ((Maybe Yes_No, Level_Display), PCDATA)
  -- |
read_Level :: STM Result [Content i] Level
read_Level = do
  y ← read_ELEMENT "level"
  y1 ← read_2 (read_IMPLIED "reference" read_Yes_No)
    read_Level_Display (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Level :: Level → [Content ()]
```

```
show_Level ((a, b), c) =
  show_ELEMENT "level"
    (show_IMPLIED "reference" show_Yes_No a ++
    show_Level_Display b)
    (show_PCDATA c)
  -- |
type Voice = PCDATA
  -- |
read_Voice :: STM Result [Content i] Voice
read_Voice = do
  y ← read_ELEMENT "voice"
  read_1 read_PCDATA (childs y)
  -- |
show_Voice :: Voice → [Content ()]
show_Voice x = show_ELEMENT "voice" [] (show_PCDATA x)
```

Fermata and wavy-line elements can be applied both to notes and to measures, so they are defined here. Wavy lines are one way to indicate trills; when used with a measure element, they should always have type="continue" set. The fermata text content represents the shape of the fermata sign and may be normal, angled, or square. An empty fermata element represents a normal fermata. The fermata type is upright if not specified.

```
  -- |
type Fermata = ((Maybe Fermata_, Print_Style), PCDATA)
data Fermata_ = Fermata_1 | Fermata_2
  deriving (Eq, Show)
  -- |
read_Fermata_ :: Data.Char.String → Result Fermata_
read_Fermata_ "upright" = return Fermata_1
read_Fermata_ "inverted" = return Fermata_2
read_Fermata_ _          =
  fail "I expect type attribute"
  -- |
show_Fermata_ :: Fermata_ → Data.Char.String
show_Fermata_ Fermata_1 = "upright"
show_Fermata_ Fermata_2 = "inverted"
  -- |
read_Fermata :: STM Result [Content i] Fermata
read_Fermata = do
  y ← read_ELEMENT "fermata"
  y1 ← read_2 (read_IMPLIED "type" read_Fermata_)
    read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Fermata :: Fermata → [Content ()]
show_Fermata ((a, b), c) =
  show_ELEMENT "fermata"
    (show_IMPLIED "type" show_Fermata_ a ++
      show_Print_Style b)
    (show_PCDATA c)
  -- |
type Wavy_Line = ((Start_Stop_Continue, Maybe Number_Level,
    Position, Placement, Color, Trill_Sound), ())
  -- |
read_Wavy_Line :: STM Result [Content i] Wavy_Line
read_Wavy_Line = do
  y ← read_ELEMENT "wavy-line"
  y1 ← read_6 (read_REQUIRED "type" read_Start_Stop_Continue)
```

```
      (read_IMPLIED "number" read_Number_Level)
      read_Position read_Placement read_Color
      read_Trill_Sound (attributes y)
    return (y1, ())
  -- |
show_Wavy_Line :: Wavy_Line → [Content ()]
show_Wavy_Line ((a, b, c, d, e, f), ()) =
    show_ELEMENT "wavy-line"
      (show_REQUIRED "type" show_Start_Stop_Continue a ++
        show_IMPLIED "number" show_Number_Level b ++
        show_Position c ++
        show_Placement d ++
        show_Color e ++
        show_Trill_Sound f
      )
      []
```

Staff assignment is only needed for music notated on multiple staves. Used by both notes and directions. Staff values are numbers, with 1 referring to the top-most staff in a part.

```
  -- |
type Staff = PCDATA
  -- |
read_Staff :: STM Result [Content i] Staff
read_Staff = do
  y ← read_ELEMENT "staff"
  read_1 read_PCDATA (childs y)
  -- |
show_Staff :: Staff → [Content ()]
show_Staff x = show_ELEMENT "staff" [] (show_PCDATA x)
```

Segno and coda signs can be associated with a measure or a general musical direction. These are visual indicators only; a sound element is needed to guide playback applications reliably.

```
  -- |
type Segno = (Print_Style, ())
  -- |
read_Segno :: STM Result [Content i] Segno
read_Segno = do
  y ← read_ELEMENT "segno"
  y1 ← read_1 read_Print_Style (attributes y)
  return (y1, ())
  -- |
show_Segno :: Segno → [Content ()]
show_Segno (x, _) = show_ELEMENT "segno" (show_Print_Style x) []
  -- |
type Coda = (Print_Style, ())
  -- |
read_Coda :: STM Result [Content i] Coda
read_Coda = do
  y ← read_ELEMENT "coda"
  y1 ← read_1 read_Print_Style (attributes y)
  return (y1, ())
  -- |
show_Coda :: Coda → [Content ()]
show_Coda (x, _) = show_ELEMENT "coda" (show_Print_Style x) []
```

These elements are used both in the time-modification and metronome-tuplet elements. The actual-notes element describes how many notes are played in the time usually occupied by the number of normal-notes. If the normal-notes type is different than the current note type (e.g., a quarter note within an

eighth note triplet), then the normal-notes type (e.g. eighth) is specified in the normal-type and normal-dot elements.

```
  -- |
type Actual_Notes = PCDATA
  -- |
read_Actual_Notes :: STM Result [Content i] Actual_Notes
read_Actual_Notes = do
  y ← read_ELEMENT "actual-notes"
  read_1 read_PCDATA (childs y)
  -- |
show_Actual_Notes :: Actual_Notes → [Content ()]
show_Actual_Notes x = show_ELEMENT "actual-notes" [] (show_PCDATA x)
  -- |
type Normal_Notes = PCDATA
  -- |
read_Normal_Notes :: STM Result [Content i] Normal_Notes
read_Normal_Notes = do
  y ← read_ELEMENT "normal-notes"
  read_1 read_PCDATA (childs y)
  -- |
show_Normal_Notes :: Normal_Notes → [Content ()]
show_Normal_Notes x = show_ELEMENT "normal-notes" [] (show_PCDATA x)
  -- |
type Normal_Type = PCDATA
  -- |
read_Normal_Type :: STM Result [Content i] Normal_Type
read_Normal_Type = do
  y ← read_ELEMENT "normal-type"
  read_1 read_PCDATA (childs y)
  -- |
show_Normal_Type :: Normal_Type → [Content ()]
show_Normal_Type x = show_ELEMENT "normal-type" [] (show_PCDATA x)
  -- |
type Normal_Dot = ()
  -- |
read_Normal_Dot :: STM Result [Content i] Normal_Dot
read_Normal_Dot = read_ELEMENT "normal-dot" ≫ return ()
  -- |
show_Normal_Dot :: Normal_Dot → [Content ()]
show_Normal_Dot _ = show_ELEMENT "normal-dot" [] []
```

Dynamics can be associated either with a note or a general musical direction. To avoid inconsistencies between and amongst the letter abbreviations for dynamics (what is sf vs. sfz, standing alone or with a trailing dynamic that is not always piano), we use the actual letters as the names of these dynamic elements. The other-dynamics element allows other dynamic marks that are not covered here, but many of those should perhaps be included in a more general musical direction element. Dynamics may also be combined as in <sf/><mp/>.

These letter dynamic symbols are separated from crescendo, decrescendo, and wedge indications. Dynamic representation is inconsistent in scores. Many things are assumed by the composer and left out, such as returns to original dynamics. Systematic representations are quite complex: for example, Humdrum has at least 3 representation formats related to dynamics. The MusicXML format captures what is in the score, but does not try to be optimal for analysis or synthesis of dynamics.

```
  -- |
type Dynamics = ((Print_Style, Placement), [Dynamics_])
  -- |
read_Dynamics :: Eq i ⇒ STM Result [Content i] Dynamics
read_Dynamics = do
  y ← read_ELEMENT "dynamics"
```

```
    y1 ← read_2 read_Print_Style read_Placement (attributes y)
    y2 ← read_1 (read_LIST read_Dynamics_) (childs y)
    return (y1, y2)
  -- |
show_Dynamics :: Dynamics → [Content ()]
show_Dynamics ((a, b), c) =
  show_ELEMENT "dynamics"
      (show_Print_Style a ++ show_Placement b)
      (show_LIST show_Dynamics_ c)
  -- |
data Dynamics_ = Dynamics_1 P
    | Dynamics_2 PP
    | Dynamics_3 PPP
    | Dynamics_4 PPPP
    | Dynamics_5 PPPPP
    | Dynamics_6 PPPPPP
    | Dynamics_7 F
    | Dynamics_8 FF
    | Dynamics_9 FFF
    | Dynamics_10 FFFF
    | Dynamics_11 FFFFF
    | Dynamics_12 FFFFFF
    | Dynamics_13 MP
    | Dynamics_14 MF
    | Dynamics_15 SF
    | Dynamics_16 SFP
    | Dynamics_17 SFPP
    | Dynamics_18 FP
    | Dynamics_19 RF
    | Dynamics_20 RFZ
    | Dynamics_21 SFZ
    | Dynamics_22 SFFZ
    | Dynamics_23 FZ
    | Dynamics_24 Other_Dynamics
      deriving (Eq, Show)
  -- |
read_Dynamics_ :: STM Result [Content i] Dynamics_
read_Dynamics_ =
  (read_P >>= return · Dynamics_1) `mplus`
  (read_PP >>= return · Dynamics_2) `mplus`
  (read_PPP >>= return · Dynamics_3) `mplus`
  (read_PPPP >>= return · Dynamics_4) `mplus`
  (read_PPPPP >>= return · Dynamics_5) `mplus`
  (read_PPPPPP >>= return · Dynamics_6) `mplus`
  (read_F >>= return · Dynamics_7) `mplus`
  (read_FF >>= return · Dynamics_8) `mplus`
  (read_FFF >>= return · Dynamics_9) `mplus`
  (read_FFFF >>= return · Dynamics_10) `mplus`
  (read_FFFFF >>= return · Dynamics_11) `mplus`
  (read_FFFFFF >>= return · Dynamics_12) `mplus`
  (read_MP >>= return · Dynamics_13) `mplus`
  (read_MF >>= return · Dynamics_14) `mplus`
  (read_SF >>= return · Dynamics_15) `mplus`
  (read_SFP >>= return · Dynamics_16) `mplus`
  (read_SFPP >>= return · Dynamics_17) `mplus`
  (read_FP >>= return · Dynamics_18) `mplus`
  (read_RF >>= return · Dynamics_19) `mplus`
  (read_RFZ >>= return · Dynamics_20) `mplus`
```

```
    (read_SFZ >>= return . Dynamics_21) `mplus`
    (read_SFFZ >>= return . Dynamics_22) `mplus`
    (read_FZ >>= return . Dynamics_23) `mplus`
    (read_Other_Dynamics >>= return . Dynamics_24)
    -- |
show_Dynamics_ :: Dynamics_ -> [Content ()]
show_Dynamics_ (Dynamics_1 x) = show_P x
show_Dynamics_ (Dynamics_2 x) = show_PP x
show_Dynamics_ (Dynamics_3 x) = show_PPP x
show_Dynamics_ (Dynamics_4 x) = show_PPPP x
show_Dynamics_ (Dynamics_5 x) = show_PPPPP x
show_Dynamics_ (Dynamics_6 x) = show_PPPPPP x
show_Dynamics_ (Dynamics_7 x) = show_F x
show_Dynamics_ (Dynamics_8 x) = show_FF x
show_Dynamics_ (Dynamics_9 x) = show_FFF x
show_Dynamics_ (Dynamics_10 x) = show_FFFF x
show_Dynamics_ (Dynamics_11 x) = show_FFFFF x
show_Dynamics_ (Dynamics_12 x) = show_FFFFFF x
show_Dynamics_ (Dynamics_13 x) = show_MP x
show_Dynamics_ (Dynamics_14 x) = show_MF x
show_Dynamics_ (Dynamics_15 x) = show_SF x
show_Dynamics_ (Dynamics_16 x) = show_SFP x
show_Dynamics_ (Dynamics_17 x) = show_SFPP x
show_Dynamics_ (Dynamics_18 x) = show_FP x
show_Dynamics_ (Dynamics_19 x) = show_RF x
show_Dynamics_ (Dynamics_20 x) = show_RFZ x
show_Dynamics_ (Dynamics_21 x) = show_SFZ x
show_Dynamics_ (Dynamics_22 x) = show_SFFZ x
show_Dynamics_ (Dynamics_23 x) = show_FZ x
show_Dynamics_ (Dynamics_24 x) = show_Other_Dynamics x
    -- |
type P = ()
    -- |
read_P :: STM Result [Content i] P
read_P = read_ELEMENT "p" >> return ()
    -- |
show_P :: P -> [Content ()]
show_P _ = show_ELEMENT "p" [] []
    -- |
type PP = ()
    -- |
read_PP :: STM Result [Content i] PP
read_PP = read_ELEMENT "pp" >> return ()
    -- |
show_PP :: PP -> [Content ()]
show_PP _ = show_ELEMENT "pp" [] []
    -- |
type PPP = ()
    -- |
read_PPP :: STM Result [Content i] PPP
read_PPP = read_ELEMENT "ppp" >> return ()
    -- |
show_PPP :: PPP -> [Content ()]
show_PPP _ = show_ELEMENT "ppp" [] []
    -- |
type PPPP = ()
    -- |
read_PPPP :: STM Result [Content i] PPPP
```

```
read_PPPP = read_ELEMENT "pppp" ≫ return ()
  -- |
show_PPPP :: PPPP → [Content ()]
show_PPPP _ = show_ELEMENT "pppp" [] []
  -- |
type PPPPP = ()
  -- |
read_PPPPP :: STM Result [Content i] PPPPP
read_PPPPP = read_ELEMENT "ppppp" ≫ return ()
  -- |
show_PPPPP :: PPPPP → [Content ()]
show_PPPPP _ = show_ELEMENT "ppppp" [] []
  -- |
type PPPPPP = ()
  -- |
read_PPPPPP :: STM Result [Content i] PPPPPP
read_PPPPPP = read_ELEMENT "pppppp" ≫ return ()
  -- |
show_PPPPPP :: PPPPPP → [Content ()]
show_PPPPPP _ = show_ELEMENT "pppppp" [] []
  -- |
type FFFFFF = ()
  -- |
read_FFFFFF :: STM Result [Content i] FFFFFF
read_FFFFFF = read_ELEMENT "ffffff" ≫ return ()
  -- |
show_FFFFFF :: FFFFFF → [Content ()]
show_FFFFFF _ = show_ELEMENT "ffffff" [] []
  -- |
type FFFFF = ()
  -- |
read_FFFFF :: STM Result [Content i] FFFFF
read_FFFFF = read_ELEMENT "fffff" ≫ return ()
  -- |
show_FFFFF :: FFFFF → [Content ()]
show_FFFFF _ = show_ELEMENT "fffff" [] []
  -- |
type FFFF = ()
  -- |
read_FFFF :: STM Result [Content i] FFFF
read_FFFF = read_ELEMENT "ffff" ≫ return ()
  -- |
show_FFFF :: FFFF → [Content ()]
show_FFFF _ = show_ELEMENT "ffff" [] []
  -- |
type FFF = ()
  -- |
read_FFF :: STM Result [Content i] FFF
read_FFF = read_ELEMENT "fff" ≫ return ()
  -- |
show_FFF :: FFF → [Content ()]
show_FFF _ = show_ELEMENT "fff" [] []
  -- |
type FF = ()
  -- |
read_FF :: STM Result [Content i] FF
read_FF = read_ELEMENT "ff" ≫ return ()
  -- |
```

```
show_FF :: FF → [Content ()]
show_FF _ = show_ELEMENT "ff" [] []
  -- |
type F = ()
  -- |
read_F :: STM Result [Content i] F
read_F = read_ELEMENT "f" ≫ return ()
  -- |
show_F :: F → [Content ()]
show_F _ = show_ELEMENT "f" [] []
  -- |
type MP = ()
  -- |
read_MP :: STM Result [Content i] MP
read_MP = read_ELEMENT "mp" ≫ return ()
  -- |
show_MP :: MP → [Content ()]
show_MP _ = show_ELEMENT "mp" [] []
  -- |
type MF = ()
  -- |
read_MF :: STM Result [Content i] MF
read_MF = read_ELEMENT "mf" ≫ return ()
  -- |
show_MF :: MF → [Content ()]
show_MF _ = show_ELEMENT "mf" [] []
  -- |
type SF = ()
  -- |
read_SF :: STM Result [Content i] SF
read_SF = read_ELEMENT "sf" ≫ return ()
  -- |
show_SF :: SF → [Content ()]
show_SF _ = show_ELEMENT "sf" [] []
  -- |
type SFP = ()
  -- |
read_SFP :: STM Result [Content i] SFP
read_SFP = read_ELEMENT "sfp" ≫ return ()
  -- |
show_SFP :: SFP → [Content ()]
show_SFP _ = show_ELEMENT "sfp" [] []
  -- |
type SFPP = ()
  -- |
read_SFPP :: STM Result [Content i] SFPP
read_SFPP = read_ELEMENT "sfpp" ≫ return ()
  -- |
show_SFPP :: SFPP → [Content ()]
show_SFPP _ = show_ELEMENT "sfpp" [] []
  -- |
type FP = ()
  -- |
read_FP :: STM Result [Content i] FP
read_FP = read_ELEMENT "fp" ≫ return ()
  -- |
show_FP :: FP → [Content ()]
show_FP _ = show_ELEMENT "fp" [] []
```

```
  -- |
type RF = ()
  -- |
read_RF :: STM Result [Content i] RF
read_RF = read_ELEMENT "rf" ≫ return ()
  -- |
show_RF :: RF → [Content ()]
show_RF _ = show_ELEMENT "rf" [] []
  -- |
type RFZ = ()
  -- |
read_RFZ :: STM Result [Content i] RFZ
read_RFZ = read_ELEMENT "rfz" ≫ return ()
  -- |
show_RFZ :: RFZ → [Content ()]
show_RFZ _ = show_ELEMENT "rfz" [] []
  -- |
type SFZ = ()
  -- |
read_SFZ :: STM Result [Content i] SFZ
read_SFZ = read_ELEMENT "sfz" ≫ return ()
  -- |
show_SFZ :: SFZ → [Content ()]
show_SFZ _ = show_ELEMENT "sfz" [] []
  -- |
type SFFZ = ()
  -- |
read_SFFZ :: STM Result [Content i] SFFZ
read_SFFZ = read_ELEMENT "sffz" ≫ return ()
  -- |
show_SFFZ :: SFFZ → [Content ()]
show_SFFZ _ = show_ELEMENT "sffz" [] []
  -- |
type FZ = ()
  -- |
read_FZ :: STM Result [Content i] FZ
read_FZ = read_ELEMENT "fz" ≫ return ()
  -- |
show_FZ :: FZ → [Content ()]
show_FZ _ = show_ELEMENT "fz" [] []
  -- |
type Other_Dynamics = PCDATA
  -- |
read_Other_Dynamics :: STM Result [Content i] Other_Dynamics
read_Other_Dynamics = do
  y ← read_ELEMENT "other-dynamics"
  read_1 read_PCDATA (childs y)
  -- |
show_Other_Dynamics :: Other_Dynamics → [Content ()]
show_Other_Dynamics x = show_ELEMENT "other-dynamics" [] (show_PCDATA x)
```

The fret, string, and fingering elements can be used either in a technical element for a note or in a frame element as part of a chord symbol.

Fingering is typically indicated 1,2,3,4,5. Multiple fingerings may be given, typically to substitute fingerings in the middle of a note. The substitution and alternate values are "no" if the attribute is not present. For guitar and other fretted instruments, the fingering element represents the fretting finger; the pluck element represents the plucking finger.

```
  -- |
type Fingering = ((Maybe Yes_No, Maybe Yes_No, Print_Style, Placement), PCDATA)
```

```
  -- |
read_Fingering :: STM Result [Content i] Fingering
read_Fingering = do
  y ← read_ELEMENT "fingering"
  y1 ← read_4 (read_IMPLIED "substitution" read_Yes_No)
    (read_IMPLIED "alternate" read_Yes_No)
    read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Fingering :: Fingering → [Content ()]
show_Fingering ((a, b, c, d), e) =
  show_ELEMENT "fingering"
    (show_IMPLIED "substitution" show_Yes_No a ++
      show_IMPLIED "alternate" show_Yes_No b ++
      show_Print_Style c ++
      show_Placement d)
    (show_PCDATA e)
```

Fret and string are used with tablature notation and chord symbols. Fret numbers start with 0 for an open string and 1 for the first fret. String numbers start with 1 for the highest string. The string element can also be used in regular notation.

```
  -- |
type Fret = ((Font, Color), PCDATA)
  -- |
read_Fret :: STM Result [Content i] Fret
read_Fret = do
  y ← read_ELEMENT "fret"
  y1 ← read_2 read_Font read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Fret :: Fret → [Content ()]
show_Fret ((a, b), c) =
  show_ELEMENT "fret"
    (show_Font a ++ show_Color b)
    (show_PCDATA c)
  -- |
type String = ((Print_Style, Placement), PCDATA)
  -- |
read_String :: STM Result [Content i] String
read_String = do
  y ← read_ELEMENT "string"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_String :: String → [Content ()]
show_String ((a, b), c) =
  show_ELEMENT "string"
    (show_Print_Style a ++ show_Placement b)
    (show_PCDATA c)
```

The tuning-step, tuning-alter, and tuning-octave elements are represented like the step, alter, and octave elements, with different names to reflect their different function. They are used in the staff-tuning and accord elements.

```
  -- |
type Tuning_Step = PCDATA
```

```
  -- |
read_Tuning_Step :: STM Result [Content i] Tuning_Step
read_Tuning_Step = do
  y ← read_ELEMENT "tuning-step"
  read_1 read_PCDATA (childs y)
  -- |
show_Tuning_Step :: Tuning_Step → [Content ()]
show_Tuning_Step x = show_ELEMENT "tuning-step" [] (show_PCDATA x)
  -- |
type Tuning_Alter = PCDATA
  -- |
read_Tuning_Alter :: STM Result [Content i] Tuning_Alter
read_Tuning_Alter = do
  y ← read_ELEMENT "tuning-alter"
  read_1 read_PCDATA (childs y)
  -- |
show_Tuning_Alter :: Tuning_Alter → [Content ()]
show_Tuning_Alter x = show_ELEMENT "tuning-alter" [] (show_PCDATA x)
  -- |
type Tuning_Octave = PCDATA
  -- |
read_Tuning_Octave :: STM Result [Content i] Tuning_Octave
read_Tuning_Octave = do
  y ← read_ELEMENT "tuning-octave"
  read_1 read_PCDATA (childs y)
  -- |
show_Tuning_Octave :: Tuning_Octave → [Content ()]
show_Tuning_Octave x = show_ELEMENT "tuning-octave" [] (show_PCDATA x)
```

The display-text element is used for exact formatting of multi-font text in element in display elements such as part-name-display. Language is Italian ("it") by default. Enclosure is none by default.

```
  -- |
type Display_Text = (Text_Formatting, PCDATA)
  -- |
read_Display_Text :: STM Result [Content i] Display_Text
read_Display_Text = do
  y ← read_ELEMENT "display-text"
  y1 ← read_1 read_Text_Formatting (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Display_Text :: Display_Text → [Content ()]
show_Display_Text (a, b) =
  show_ELEMENT "display-text"
    (show_Text_Formatting a)
    (show_PCDATA b)
```

The accidental-text element is used for exact formatting of accidentals in display elements such as part-name-display. Values are the same as for the accidental element. Enclosure is none by default.

```
  -- |
type Accidental_Text = (Text_Formatting, PCDATA)
  -- |
read_Accidental_Text :: STM Result [Content i] Accidental_Text
read_Accidental_Text = do
  y ← read_ELEMENT "accidental-text"
  y1 ← read_1 read_Text_Formatting (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
```

```
  -- |
show_Accidental_Text :: Accidental_Text → [Content ()]
show_Accidental_Text (a, b) =
  show_ELEMENT "accidental-text"
    (show_Text_Formatting a)
    (show_PCDATA b)
```

The part-name-display and part-abbreviation-display elements are used in both the score.mod and direction.mod files. They allow more precise control of how part names and abbreviations appear throughout a score. The print-object attributes can be used to determine what, if anything, is printed at the start of each system. Formatting specified in the part-name-display and part-abbreviation-display elements override the formatting specified in the part-name and part-abbreviation elements, respectively.

```
type Part_Name_Display = (Print_Object, [Part_Name_Display_])
  -- |
read_Part_Name_Display :: Eq i ⇒ STM Result [Content i] Part_Name_Display
read_Part_Name_Display = do
  y ← read_ELEMENT "part-name-display"
  y1 ← read_1 read_Print_Object (attributes y)
  y2 ← read_1 (read_LIST read_Part_Name_Display_) (childs y)
  return (y1, y2)
  -- |
show_Part_Name_Display :: Part_Name_Display → [Content ()]
show_Part_Name_Display (a, b) =
  show_ELEMENT "part-name-display"
    (show_Print_Object a)
    (show_LIST show_Part_Name_Display_ b)
  -- |
data Part_Name_Display_ = Part_Name_Display_1 Display_Text
  | Part_Name_Display_2 Accidental_Text
  deriving (Eq, Show)
  -- |
read_Part_Name_Display_ :: STM Result [Content i] Part_Name_Display_
read_Part_Name_Display_ =
  (read_Display_Text ≫= (return · Part_Name_Display_1)) `mplus`
  (read_Accidental_Text ≫= (return · Part_Name_Display_2)) `mplus`
  fail "part-name-display"
  -- |
show_Part_Name_Display_ :: Part_Name_Display_ → [Content ()]
show_Part_Name_Display_ (Part_Name_Display_1 x) = show_Display_Text x
show_Part_Name_Display_ (Part_Name_Display_2 x) = show_Accidental_Text x
  -- |
type Part_Abbreviation_Display = (Print_Object, [Part_Abbreviation_Display_])
  -- |
read_Part_Abbreviation_Display :: Eq i ⇒
  STM Result [Content i] Part_Abbreviation_Display
read_Part_Abbreviation_Display = do
  y ← read_ELEMENT "part-abbreviation-display"
  y1 ← read_1 read_Print_Object (attributes y)
  y2 ← read_1 (read_LIST read_Part_Abbreviation_Display_) (childs y)
  return (y1, y2)
  -- |
show_Part_Abbreviation_Display :: Part_Abbreviation_Display → [Content ()]
show_Part_Abbreviation_Display (a, b) =
  show_ELEMENT "part-abbreviation-display"
    (show_Print_Object a)
    (show_LIST show_Part_Abbreviation_Display_ b)
  -- |
data Part_Abbreviation_Display_ =
```

```
    Part_Abbreviation_Display_1 Display_Text
  | Part_Abbreviation_Display_2 Accidental_Text
    deriving (Eq, Show)
  -- |
read_Part_Abbreviation_Display_ ::
  STM Result [Content i] Part_Abbreviation_Display_
read_Part_Abbreviation_Display_ =
  (read_Display_Text >>= (return . Part_Abbreviation_Display_1)) `mplus`
  (read_Accidental_Text >>= (return . Part_Abbreviation_Display_2)) `mplus`
  fail "part-name-display"
  -- |
show_Part_Abbreviation_Display_ :: Part_Abbreviation_Display_ -> [Content ()]
show_Part_Abbreviation_Display_
  (Part_Abbreviation_Display_1 x) = show_Display_Text x
show_Part_Abbreviation_Display_
  (Part_Abbreviation_Display_2 x) = show_Accidental_Text x
  -- |
```

The midi-instrument element can be a part of either the score-instrument element at the start of a part, or the sound element within a part. The id attribute refers to the score-instrument affected by the change.

```
  -- |
type Midi_Instrument = (ID, (Maybe Midi_Channel, Maybe Midi_Name,
  Maybe Midi_Bank, Maybe Midi_Program, Maybe Midi_Unpitched,
  Maybe Volume, Maybe Pan, Maybe Elevation))
  -- |
read_Midi_Instrument :: STM Result [Content i] Midi_Instrument
read_Midi_Instrument = do
  y <- read_ELEMENT "midi-instrument"
  y1 <- read_1 (read_REQUIRED "id" read_ID) (attributes y)
  y2 <- read_8 (read_MAYBE read_Midi_Channel) (read_MAYBE read_Midi_Name)
    (read_MAYBE read_Midi_Bank) (read_MAYBE read_Midi_Program)
    (read_MAYBE read_Midi_Unpitched) (read_MAYBE read_Volume)
    (read_MAYBE read_Pan) (read_MAYBE read_Elevation)
    (childs y)
  return (y1, y2)
  -- |
show_Midi_Instrument :: Midi_Instrument -> [Content ()]
show_Midi_Instrument (a, (b, c, d, e, f, g, h, i)) =
  show_ELEMENT "midi-instrument"
    (show_REQUIRED "id" show_ID a)
    (show_MAYBE show_Midi_Channel b ++ show_MAYBE show_Midi_Name c ++
      show_MAYBE show_Midi_Bank d ++ show_MAYBE show_Midi_Program e ++
      show_MAYBE show_Midi_Unpitched f ++ show_MAYBE show_Volume g ++
      show_MAYBE show_Pan h ++ show_MAYBE show_Elevation i)
```

MIDI 1.0 channel numbers range from 1 to 16.

```
  -- |
type Midi_Channel = PCDATA
  -- |
read_Midi_Channel :: STM Result [Content i] Midi_Channel
read_Midi_Channel = do
  y <- read_ELEMENT "midi-channel"
  read_1 read_PCDATA (childs y)
  -- |
show_Midi_Channel :: Midi_Channel -> [Content ()]
show_Midi_Channel x =
  show_ELEMENT "midi-channel" [] (show_PCDATA x)
```

MIDI names correspond to ProgramName meta-events within a Standard MIDI File.

```
  -- |
type Midi_Name = PCDATA
  -- |
read_Midi_Name :: STM Result [Content i] Midi_Name
read_Midi_Name = do
  y ← read_ELEMENT "midi-name"
  read_1 read_PCDATA (childs y)
  -- |
show_Midi_Name :: Midi_Name → [Content ()]
show_Midi_Name x =
  show_ELEMENT "midi-name" [] (show_PCDATA x)
```

MIDI 1.0 bank numbers range from 1 to 16,384.

```
  -- |
type Midi_Bank = PCDATA
  -- |
read_Midi_Bank :: STM Result [Content i] Midi_Bank
read_Midi_Bank = do
  y ← read_ELEMENT "midi-bank"
  read_1 read_PCDATA (childs y)
  -- |
show_Midi_Bank :: Midi_Bank → [Content ()]
show_Midi_Bank x =
  show_ELEMENT "midi-bank" [] (show_PCDATA x)
```

MIDI 1.0 program numbers range from 1 to 128.

```
  -- |
type Midi_Program = PCDATA
  -- |
read_Midi_Program :: STM Result [Content i] Midi_Program
read_Midi_Program = do
  y ← read_ELEMENT "midi-program"
  read_1 read_PCDATA (childs y)
  -- |
show_Midi_Program :: Midi_Program → [Content ()]
show_Midi_Program x =
  show_ELEMENT "midi-program" [] (show_PCDATA x)
```

For unpitched instruments, specify a MIDI 1.0 note number ranging from 1 to 128. Usually used with MIDI banks for percussion.

```
  -- |
type Midi_Unpitched = PCDATA
  -- |
read_Midi_Unpitched :: STM Result [Content i] Midi_Unpitched
read_Midi_Unpitched = do
  y ← read_ELEMENT "midi-unpitched"
  read_1 read_PCDATA (childs y)
  -- |
show_Midi_Unpitched :: Midi_Unpitched → [Content ()]
show_Midi_Unpitched x =
  show_ELEMENT "midi-unpitched" [] (show_PCDATA x)
```

The volume value is a percentage of the maximum ranging from 0 to 100, with decimal values allowed. This corresponds to a scaling value for the MIDI 1.0 channel volume controller.

```
  -- |
type Volume = PCDATA
```

```
  -- |
read_Volume :: STM Result [Content i] Volume
read_Volume = do
  y ← read_ELEMENT "volume"
  read_1 read_PCDATA (childs y)
  -- |
show_Volume :: Volume → [Content ()]
show_Volume x =
  show_ELEMENT "volume" [] (show_PCDATA x)
```

Pan and elevation allow placing of sound in a 3-D space relative to the listener. Both are expressed in degrees ranging from -180 to 180. For pan, 0 is straight ahead, -90 is hard left, 90 is hard right, and -180 and 180 are directly behind the listener. For elevation, 0 is level with the listener, 90 is directly above, and -90 is directly below.

```
  -- |
type Pan = PCDATA
  -- |
read_Pan :: STM Result [Content i] Pan
read_Pan = do
  y ← read_ELEMENT "pan"
  read_1 read_PCDATA (childs y)
  -- |
show_Pan :: Pan → [Content ()]
show_Pan x =
  show_ELEMENT "pan" [] (show_PCDATA x)
  -- |
type Elevation = PCDATA
  -- |
read_Elevation :: STM Result [Content i] Elevation
read_Elevation = do
  y ← read_ELEMENT "elevation"
  read_1 read_PCDATA (childs y)
  -- |
show_Elevation :: Elevation → [Content ()]
show_Elevation x =
  show_ELEMENT "elevation" [] (show_PCDATA x)
```

2.4 Container

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Container where
import Text.XML.MusicXML.Common
import Text.XML.HaXml.Types (Content,
  DocTypeDecl (..), ExternalID (..), PubidLiteral (..), SystemLiteral (..))
import Prelude (FilePath, Maybe (..), Eq, Monad (..), (++), map)
```

Starting with Version 2.0, the MusicXML format includes a standard zip compressed version. These zip files can contain multiple MusicXML files as well as other media files for images and sound. The container DTD describes the contents of the META-INF/container.xml file. The container describes the starting point for the MusicXML version of the file, as well as alternate renditions such as PDF and audio versions of the musical score.

The MusicXML 2.0 zip file format is compatible with the zip format used by the java.util.zip package and Java JAR files. It is based on the Info-ZIP format described at:

ftp://ftp.uu.net/pub/archiving/zip/doc/appnote-970311-iz.zip

The JAR file format is specified at:

http://java.sun.com/javase/6/docs/technotes/guides/jar/jar.html

Note that, compatible with JAR files, file names should be encoded in UTF-8 format.

Files with the zip container are compressed the DEFLATE algorithm. The DEFLATE Compressed Data Format (RFC 1951) is specified at:

http://www.ietf.org/rfc/rfc1951.txt

The recommended file suffix for compressed MusicXML 2.0 files is .mxl. The recommended media type for a compressed MusicXML file is:

application/vnd.recordare.musicxml

The recommended media type for an uncompressed MusicXML file is:

application/vnd.recordare.musicxml+xml

Suggested use:

```
<!DOCTYPE container PUBLIC
            "-//Recordare//DTD MusicXML 2.0 Container//EN"
            "http://www.musicxml.org/dtds/container.dtd">
```

```
  -- |
doctype :: DocTypeDecl
doctype = DTD "container"
  (Just (PUBLIC (PubidLiteral "-//Recordare//DTD MusicXML 2.0 Container//EN")
    (SystemLiteral "http://www.musicxml.org/dtds/container.dtd")))
    []
  -- |
getFiles :: Container → [FilePath]
getFiles = map (λ((a, _), _) → a)
```

Container is the document element.

```
  -- * Container
  -- |
type Container = Rootfiles
  -- |
read_Container :: Eq i ⇒ STM Result [Content i] Container
read_Container = do
  y ← read_ELEMENT "container"
  read_1 read_Rootfiles (childs y)
  -- |
show_Container :: Container → [Content ()]
show_Container a =
  show_ELEMENT "container" [] (show_Rootfiles a)
```

Rootfiles include the starting points for the different representations of a MusicXML score. The MusicXML root must be described in the first rootfile element. Additional rootfile elements can describe alternate versions such as PDF and audio files.

```
  -- |
type Rootfiles = [Rootfile]
  -- |
read_Rootfiles :: Eq i ⇒ STM Result [Content i] Rootfiles
read_Rootfiles = do
  y ← read_ELEMENT "rootfiles"
  read_1 (read_LIST1 read_Rootfile) (childs y)
  -- |
show_Rootfiles :: Rootfiles → [Content ()]
show_Rootfiles a =
  show_ELEMENT "rootfiles" [] (show_LIST1 show_Rootfile a)
```

The rootfile element describes each top-level file in the MusicXML container. The full-path attribute provides the path relative to the root folder of the zip file. The media-type identifies the type of different top-level root files. It is an error to have a non-MusicXML media-type value in the first rootfile in a rootfiles element. If no media-type value is present, a MusicXML file is assumed. A MusicXML file used as a rootfile may have score-partwise, score-timewise, or opus as its document element.

```
  -- |
type Rootfile = ((CDATA, Maybe CDATA), ())
  -- |
read_Rootfile :: Eq i ⇒ STM Result [Content i] Rootfile
read_Rootfile = do
  y ← read_ELEMENT "rootfile"
  y1 ← read_2 (read_REQUIRED "full-path" read_CDATA)
    (read_IMPLIED "media-type" read_CDATA) (attributes y)
  return (y1, ())
  -- |
show_Rootfile :: Rootfile → [Content ()]
show_Rootfile ((a, b), _) =
  show_ELEMENT "rootfile"
    (show_REQUIRED "full-path" show_CDATA a ++
      show_IMPLIED "media-type" show_CDATA b) []
```

2.5 Direction

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Direction where
import Text.XML.MusicXML.Common
import Text.XML.MusicXML.Layout hiding (Tenths, read_Tenths, show_Tenths)
import Text.XML.HaXml.Types (Content)
import Control.Monad (MonadPlus (..))
import Prelude (Maybe (..), Show, Eq, Monad (..), (++), (.))
import qualified Data.Char (String)
```

This direction DTD module contains the direction element and its children. Directions are not note-specific, but instead are associated with a part or the overall score.

Harmony indications and general print and sound suggestions are likewise not necessarily attached to particular note elements, and are included here as well.

A direction is a musical indication that is not attached to a specific note. Two or more may be combined to indicate starts and stops of wedges, dashes, etc.

By default, a series of direction-type elements and a series of child elements of a direction-type within a single direction element follow one another in sequence visually. For a series of direction-type children, non- positional formatting attributes are carried over from the previous element by default.

```
  -- * Direction
  -- |
type Direction = ((Placement, Directive),
  ([Direction_Type], Maybe Offset, Editorial_Voice,
  Maybe Staff, Maybe Sound))
  -- |
read_Direction :: Eq i ⇒ STM Result [Content i] Direction
read_Direction = do
  y ← read_ELEMENT "direction"
  y1 ← read_2 read_Placement read_Directive (attributes y)
```

```
    y2 ← read_5 (read_LIST1 read_Direction_Type) (read_MAYBE read_Offset)
      (read_Editorial_Voice) (read_MAYBE read_Staff)
      (read_MAYBE read_Sound) (childs y)
    return (y1, y2)
    -- |
show_Direction :: Direction → [Content ()]
show_Direction ((a, b), (c, d, e, f, g)) =
    show_ELEMENT "direction" (show_Placement a ++ show_Directive b)
      (show_LIST show_Direction_Type c ++
      show_MAYBE show_Offset d ++
      show_Editorial_Voice e ++
      show_MAYBE show_Staff f ++
      show_MAYBE show_Sound g)
```

Textual direction types may have more than 1 component due to multiple fonts. The dynamics element may also be used in the notations element, and is defined in the common.mod file.

```
    -- ** Direction_Type
    -- |
type Direction_Type = Direction_Type_
    -- |
read_Direction_Type :: Eq i ⇒ STM Result [Content i] Direction_Type
read_Direction_Type = do
    y ← read_ELEMENT "direction-type"
    read_1 read_Direction_Type_ (childs y)
    -- |
show_Direction_Type :: Direction_Type → [Content ()]
show_Direction_Type a =
    show_ELEMENT "direction-type" [] (show_Direction_Type_ a)
    -- |
data Direction_Type_ = Direction_Type_1 [Rehearsal]
    | Direction_Type_2 [Segno]
    | Direction_Type_3 [Words]
    | Direction_Type_4 [Coda]
    | Direction_Type_5 Wedge
    | Direction_Type_6 [Dynamics]
    | Direction_Type_7 Dashes
    | Direction_Type_8 Bracket
    | Direction_Type_9 Pedal
    | Direction_Type_10 Metronome
    | Direction_Type_11 Octave_Shift
    | Direction_Type_12 Harp_Pedals
    | Direction_Type_13 Damp
    | Direction_Type_14 Damp_All
    | Direction_Type_15 Eyeglasses
    | Direction_Type_16 Scordatura
    | Direction_Type_17 Image
    | Direction_Type_18 Accordion_Registration
    | Direction_Type_19 Other_Direction
      deriving (Eq, Show)
    -- |
read_Direction_Type_ :: Eq i ⇒ STM Result [Content i] Direction_Type_
read_Direction_Type_ =
    (read_LIST1 read_Rehearsal >>= return · Direction_Type_1) `mplus`
    (read_LIST1 read_Segno >>= return · Direction_Type_2) `mplus`
    (read_LIST1 read_Words >>= return · Direction_Type_3) `mplus`
    (read_LIST1 read_Coda >>= return · Direction_Type_4) `mplus`
    (read_Wedge >>= return · Direction_Type_5) `mplus`
    (read_LIST1 read_Dynamics >>= return · Direction_Type_6) `mplus`
```

```
    (read_Dashes >>= return . Direction_Type_7) `mplus`
    (read_Bracket >>= return . Direction_Type_8) `mplus`
    (read_Pedal >>= return . Direction_Type_9) `mplus`
    (read_Metronome >>= return . Direction_Type_10) `mplus`
    (read_Octave_Shift >>= return . Direction_Type_11) `mplus`
    (read_Harp_Pedals >>= return . Direction_Type_12) `mplus`
    (read_Damp >>= return . Direction_Type_13) `mplus`
    (read_Damp_All >>= return . Direction_Type_14) `mplus`
    (read_Eyeglasses >>= return . Direction_Type_15) `mplus`
    (read_Scordatura >>= return . Direction_Type_16) `mplus`
    (read_Image >>= return . Direction_Type_17) `mplus`
    (read_Accordion_Registration >>= return . Direction_Type_18) `mplus`
    (read_Other_Direction >>= return . Direction_Type_19)
    -- |
  show_Direction_Type_ :: Direction_Type_ -> [Content ()]
  show_Direction_Type_ (Direction_Type_1 a) = show_LIST1 show_Rehearsal a
  show_Direction_Type_ (Direction_Type_2 a) = show_LIST1 show_Segno a
  show_Direction_Type_ (Direction_Type_3 a) = show_LIST1 show_Words a
  show_Direction_Type_ (Direction_Type_4 a) = show_LIST1 show_Coda a
  show_Direction_Type_ (Direction_Type_5 a) = show_Wedge a
  show_Direction_Type_ (Direction_Type_6 a) = show_LIST1 show_Dynamics a
  show_Direction_Type_ (Direction_Type_7 a) = show_Dashes a
  show_Direction_Type_ (Direction_Type_8 a) = show_Bracket a
  show_Direction_Type_ (Direction_Type_9 a) = show_Pedal a
  show_Direction_Type_ (Direction_Type_10 a) = show_Metronome a
  show_Direction_Type_ (Direction_Type_11 a) = show_Octave_Shift a
  show_Direction_Type_ (Direction_Type_12 a) = show_Harp_Pedals a
  show_Direction_Type_ (Direction_Type_13 a) = show_Damp a
  show_Direction_Type_ (Direction_Type_14 a) = show_Damp_All a
  show_Direction_Type_ (Direction_Type_15 a) = show_Eyeglasses a
  show_Direction_Type_ (Direction_Type_16 a) = show_Scordatura a
  show_Direction_Type_ (Direction_Type_17 a) = show_Image a
  show_Direction_Type_ (Direction_Type_18 a) = show_Accordion_Registration a
  show_Direction_Type_ (Direction_Type_19 a) = show_Other_Direction a
```

Entities related to print suggestions apply to the individual direction-type, not to the overall direction. Language is Italian ("it") by default. Enclosure is square by default.

```
    -- |
  type Rehearsal = ((Print_Style, Text_Decoration,
    Maybe CDATA, Text_Direction, Text_Rotation,
    Maybe Rehearsal_), PCDATA)
    -- |
  read_Rehearsal :: STM Result [Content i] Rehearsal
  read_Rehearsal = do
    y <- read_ELEMENT "rehearsal"
    y1 <- read_6 read_Print_Style read_Text_Decoration
      (read_IMPLIED "xml:lang" read_CDATA)
      read_Text_Direction read_Text_Rotation
      (read_IMPLIED "enclosure" read_Rehearsal_)
      (attributes y)
    y2 <- read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
  show_Rehearsal :: Rehearsal -> [Content ()]
  show_Rehearsal ((a, b, c, d, e, f), g) =
    show_ELEMENT "rehearsal" (show_Print_Style a ++ show_Text_Decoration b ++
      show_IMPLIED "xml:lang" show_CDATA c ++
      show_Text_Direction d ++ show_Text_Rotation e ++
```

```
      show_IMPLIED "enclosure" show_Rehearsal_ f)
      (show_PCDATA g)
  -- |
data Rehearsal_ = Rehearsal_1 | Rehearsal_2 | Rehearsal_3
  deriving (Eq, Show)
  -- |
read_Rehearsal_ :: Data.Char.String → Result Rehearsal_
read_Rehearsal_ "square" = return Rehearsal_1
read_Rehearsal_ "circle" = return Rehearsal_2
read_Rehearsal_ "none" = return Rehearsal_3
read_Rehearsal_ x = fail x
  -- |
show_Rehearsal_ :: Rehearsal_ → Data.Char.String
show_Rehearsal_ Rehearsal_1 = "square"
show_Rehearsal_ Rehearsal_2 = "circle"
show_Rehearsal_ Rehearsal_3 = "none"
```

Left justification is assumed if not specified. Language is Italian ("it") by default. Enclosure is none by default.

```
  -- |
type Words = (Text_Formatting, PCDATA)
  -- |
read_Words :: STM Result [Content i] Words
read_Words = do
  y ← read_ELEMENT "words"
  y1 ← read_1 read_Text_Formatting (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Words :: Words → [Content ()]
show_Words (a, b) =
  show_ELEMENT "words" (show_Text_Formatting a) (show_PCDATA b)
```

Wedge spread is measured in tenths of staff line space. The type is crescendo for the start of a wedge that is closed at the left side, and diminuendo for the start of a wedge that is closed on the right side. Spread values at the start of a crescendo wedge or end of a diminuendo wedge are ignored.

```
type Wedge = ((Wedge_, Maybe Number_Level, Maybe CDATA,
  Position, Color), ())
  -- |
read_Wedge :: STM Result [Content i] Wedge
read_Wedge = do
  y ← read_ELEMENT "wedge"
  y1 ← read_5 (read_REQUIRED "type" read_Wedge_)
    (read_IMPLIED "number" read_Number_Level)
    (read_IMPLIED "spread" read_CDATA)
    read_Position read_Color (attributes y)
  return (y1, ())
  -- |
show_Wedge :: Wedge → [Content ()]
show_Wedge ((a, b, c, d, e), _) =
  show_ELEMENT "wedge" (show_REQUIRED "type" show_Wedge_ a ++
    show_IMPLIED "number" show_Number_Level b ++
    show_IMPLIED "spread" show_CDATA c ++
    show_Position d ++ show_Color e) []
  -- |
data Wedge_ = Wedge_1 | Wedge_2 | Wedge_3
  deriving (Eq, Show)
  -- |
```

```
read_Wedge_ :: Data.Char.String → Result Wedge_
read_Wedge_ "crescendo" = return Wedge_1
read_Wedge_ "diminuendo" = return Wedge_2
read_Wedge_ "stop" = return Wedge_3
read_Wedge_ x = fail x
  -- |
show_Wedge_ :: Wedge_ → Data.Char.String
show_Wedge_ Wedge_1 = "crescendo"
show_Wedge_ Wedge_2 = "diminuendo"
show_Wedge_ Wedge_3 = "stop"
```

Dashes, used for instance with cresc. and dim. marks.

```
  -- |
type Dashes = ((Start_Stop, Maybe Number_Level,
  Position, Color), ())
  -- |
read_Dashes :: STM Result [Content i] Dashes
read_Dashes = do
  y ← read_ELEMENT "dashes"
  y1 ← read_4 (read_REQUIRED "type" read_Start_Stop)
    (read_IMPLIED "number" read_Number_Level)
    read_Position read_Color (attributes y)
  return (y1, ())
  -- |
show_Dashes :: Dashes → [Content ()]
show_Dashes ((a, b, c, d), _) =
  show_ELEMENT "dashes" (show_REQUIRED "type" show_Start_Stop a ++
    show_IMPLIED "number" show_Number_Level b ++
    show_Position c ++ show_Color d) []
```

Brackets are combined with words in a variety of modern directions. The line-end attribute specifies if there is a jog up or down (or both), an arrow, or nothing at the start or end of the bracket. If the line-end is up or down, the length of the jog can be specified using the end-length attribute. The line-type is solid by default.

```
  -- |
type Bracket = ((Start_Stop, Maybe Number_Level,
  Bracket_, Maybe Tenths, Line_Type, Position, Color), ())
  -- |
read_Bracket :: STM Result [Content i] Bracket
read_Bracket = do
  y ← read_ELEMENT "bracket"
  y1 ← read_7 (read_REQUIRED "type" read_Start_Stop)
    (read_IMPLIED "number" read_Number_Level)
    (read_REQUIRED "line-end" read_Bracket_)
    (read_IMPLIED "end-length" read_Tenths)
    read_Line_Type read_Position read_Color
    (attributes y)
  return (y1, ())
  -- |
show_Bracket :: Bracket → [Content ()]
show_Bracket ((a, b, c, d, e, f, g), _) =
  show_ELEMENT "bracket" (show_REQUIRED "type" show_Start_Stop a ++
    show_IMPLIED "number" show_Number_Level b ++
    show_REQUIRED "line-end" show_Bracket_ c ++
    show_IMPLIED "end-length" show_Tenths d ++
    show_Line_Type e ++ show_Position f ++
    show_Color g) []
  -- |
```

```
data Bracket_ = Bracket_1 | Bracket_2 | Bracket_3 | Bracket_4 | Bracket_5
  deriving (Eq, Show)
  -- |
read_Bracket_ :: Data.Char.String → Result Bracket_
read_Bracket_ "up" = return Bracket_1
read_Bracket_ "down" = return Bracket_2
read_Bracket_ "both" = return Bracket_3
read_Bracket_ "arrow" = return Bracket_4
read_Bracket_ "none" = return Bracket_5
read_Bracket_ x = fail x
  -- |
show_Bracket_ :: Bracket_ → Data.Char.String
show_Bracket_ Bracket_1 = "up"
show_Bracket_ Bracket_2 = "down"
show_Bracket_ Bracket_3 = "both"
show_Bracket_ Bracket_4 = "arrow"
show_Bracket_ Bracket_5 = "none"
```

*Piano pedal marks. The line attribute is yes if pedal lines are used, no if Ped and * signs are used. The change type is used with line set to yes.*

```
  -- |
type Pedal = ((Pedal_, Maybe Yes_No, Print_Style), ())
  -- |
read_Pedal :: STM Result [Content i] Pedal
read_Pedal = do
  y ← read_ELEMENT "pedal"
  y1 ← read_3 (read_REQUIRED "type" read_Pedal_)
    (read_IMPLIED "line" read_Yes_No)
    read_Print_Style (attributes y)
  return (y1, ())
  -- |
show_Pedal :: Pedal → [Content ()]
show_Pedal ((a, b, c), _) =
  show_ELEMENT "pedal" (show_REQUIRED "type" show_Pedal_ a ++
    show_IMPLIED "line" show_Yes_No b ++
    show_Print_Style c) []
  -- |
data Pedal_ = Pedal_1 | Pedal_2 | Pedal_3
  deriving (Eq, Show)
read_Pedal_ :: Data.Char.String → Result Pedal_
read_Pedal_ "start" = return Pedal_1
read_Pedal_ "stop" = return Pedal_2
read_Pedal_ "change" = return Pedal_3
read_Pedal_ x = fail x
  -- |
show_Pedal_ :: Pedal_ → Data.Char.String
show_Pedal_ Pedal_1 = "start"
show_Pedal_ Pedal_2 = "stop"
show_Pedal_ Pedal_3 = "change"
  -- |
```

Metronome marks and other metric relationships.

The beat-unit values are the same as for a type element, and the beat-unit-dot works like the dot element. The per-minute element can be a number, or a text description including numbers. The parentheses attribute indicates whether or not to put the metronome mark in parentheses; its value is no if not specified. If a font is specified for the per-minute element, it overrides the font specified for the overall metronome element. This allows separate specification of a music font for beat-unit and a text font for the numeric value in cases where a single metronome font is not used.

The metronome-note and metronome-relation elements allow for the specification of more complicated metric relationships, such as swing tempo marks where two eighths are equated to a quarter note / eighth note triplet. The metronome-type, metronome-beam, and metronome-dot elements work like the type, beam, and dot elements. The metronome-tuplet element uses the same element structure as the time-modification element along with some attributes from the tuplet element. The metronome-relation element describes the relationship symbol that goes between the two sets of metronome-note elements. The currently allowed value is equals, but this may expand in future versions. If the element is empty, the equals value is used. The metronome-relation and the following set of metronome-note elements are optional to allow display of an isolated Grundschlagnote.

```
  -- |
type Metronome = ((Print_Style, Maybe Yes_No), Metronome_A)
  -- |
read_Metronome :: Eq i => STM Result [Content i] Metronome
read_Metronome = do
  y <- read_ELEMENT "metronome"
  y1 <- read_2 read_Print_Style (read_IMPLIED "parentheses" read_Yes_No)
    (attributes y)
  y2 <- read_1 read_Metronome_A (childs y)
  return (y1, y2)
  -- |
show_Metronome :: Metronome -> [Content ()]
show_Metronome ((a, b), c) =
  show_ELEMENT "metronome" (show_Print_Style a ++
    show_IMPLIED "parentheses" show_Yes_No b)
    (show_Metronome_A c)
  -- |
data Metronome_A = Metronome_1 (Beat_Unit, [Beat_Unit_Dot], Metronome_B)
    | Metronome_2 ([Metronome_Note],
      Maybe (Metronome_Relation, [Metronome_Note]))
    deriving (Eq, Show)
  -- |
read_Metronome_A :: Eq i => STM Result [Content i] Metronome_A
read_Metronome_A =
  (read_Metronome_A_aux1 >>= return . Metronome_1) `mplus`
  (read_Metronome_A_aux2 >>= return . Metronome_2)
  -- |
show_Metronome_A :: Metronome_A -> [Content ()]
show_Metronome_A (Metronome_1 (a, b, c)) = show_Beat_Unit a ++
  show_LIST show_Beat_Unit_Dot b ++
  show_Metronome_B c
show_Metronome_A (Metronome_2 (a, b)) = show_LIST show_Metronome_Note a ++
  show_MAYBE show_Metronome_A_aux1 b
  -- |
read_Metronome_A_aux1 :: Eq i =>
  STM Result [Content i] (Beat_Unit, [Beat_Unit_Dot], Metronome_B)
read_Metronome_A_aux1 = do
  y1 <- read_Beat_Unit
  y2 <- read_LIST read_Beat_Unit_Dot
  y3 <- read_Metronome_B
  return (y1, y2, y3)
  -- |
read_Metronome_A_aux2 :: Eq i => STM Result [Content i]
  ([Metronome_Note], Maybe (Metronome_Relation, [Metronome_Note]))
read_Metronome_A_aux2 = do
  y1 <- read_LIST1 read_Metronome_Note
  y2 <- read_MAYBE read_Metronome_A_aux3
  return (y1, y2)
  -- |
```

```
read_Metronome_A_aux3 :: Eq i ⇒
  STM Result [Content i] (Metronome_Relation, [Metronome_Note])
read_Metronome_A_aux3 = do
  y1 ← read_Metronome_Relation
  y2 ← read_LIST1 read_Metronome_Note
  return (y1, y2)
  -- |
show_Metronome_A_aux1 :: (Metronome_Relation, [Metronome_Note]) → [Content ()]
show_Metronome_A_aux1 (a, b) = show_Metronome_Relation a ++
    show_LIST show_Metronome_Note b
  -- |
data Metronome_B = Metronome_3 Per_Minute
     | Metronome_4 (Beat_Unit, [Beat_Unit_Dot])
     deriving (Eq, Show)
  -- |
read_Metronome_B :: Eq i ⇒ STM Result [Content i] Metronome_B
read_Metronome_B =
  (read_Per_Minute >>= return · Metronome_3) `mplus`
  (read_Metronome_B_aux1 >>= return · Metronome_4)
  -- |
show_Metronome_B :: Metronome_B → [Content ()]
show_Metronome_B (Metronome_3 a) = show_Per_Minute a
show_Metronome_B (Metronome_4 (a, b)) = show_Beat_Unit a ++
  show_LIST show_Beat_Unit_Dot b
  -- |
read_Metronome_B_aux1 :: Eq i ⇒
  STM Result [Content i] (Beat_Unit, [Beat_Unit_Dot])
read_Metronome_B_aux1 = do
  y1 ← read_Beat_Unit
  y2 ← read_LIST read_Beat_Unit_Dot
  return (y1, y2)
  -- |
type Beat_Unit = PCDATA
  -- |
read_Beat_Unit :: STM Result [Content i] Beat_Unit
read_Beat_Unit = do
  y ← read_ELEMENT "beat-unit"
  read_1 read_PCDATA (childs y)
  -- |
show_Beat_Unit :: Beat_Unit → [Content ()]
show_Beat_Unit a = show_ELEMENT "beat-unit" [] (show_PCDATA a)
  -- |
type Beat_Unit_Dot = ()
  -- |
read_Beat_Unit_Dot :: STM Result [Content i] Beat_Unit_Dot
read_Beat_Unit_Dot = read_ELEMENT "beat-unit-dot" >> return ()
  -- |
show_Beat_Unit_Dot :: Beat_Unit_Dot → [Content ()]
show_Beat_Unit_Dot _ = show_ELEMENT "beat-unit-dot" [] []
  -- |
type Per_Minute = (Font, PCDATA)
  -- |
read_Per_Minute :: STM Result [Content i] Per_Minute
read_Per_Minute = do
  y ← read_ELEMENT "per-minute"
  y1 ← read_1 read_Font (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
```

```
  -- |
show_Per_Minute :: Per_Minute → [Content ()]
show_Per_Minute (a, b) =
  show_ELEMENT "per-minute" (show_Font a) (show_PCDATA b)
  -- |
type Metronome_Note = (Metronome_Type, [Metronome_Dot],
  [Metronome_Beam], Maybe Metronome_Tuplet)
  -- |
read_Metronome_Note :: Eq i ⇒ STM Result [Content i] Metronome_Note
read_Metronome_Note = do
  y ← read_ELEMENT "metronome-note"
  read_4 read_Metronome_Type (read_LIST read_Metronome_Dot)
    (read_LIST read_Metronome_Beam)
    (read_MAYBE read_Metronome_Tuplet) (childs y)
  -- |
show_Metronome_Note :: Metronome_Note → [Content ()]
show_Metronome_Note (a, b, c, d) =
  show_ELEMENT "metronome-note" []
    (show_Metronome_Type a ++ show_LIST show_Metronome_Dot b ++
      show_LIST show_Metronome_Beam c ++ show_MAYBE show_Metronome_Tuplet d)
  -- |
type Metronome_Relation = PCDATA
  -- |
read_Metronome_Relation :: STM Result [Content i] Metronome_Relation
read_Metronome_Relation = do
  y ← read_ELEMENT "metronome-relation"
  read_1 read_PCDATA (childs y)
  -- |
show_Metronome_Relation :: Metronome_Relation → [Content ()]
show_Metronome_Relation a =
  show_ELEMENT "metronome-relation" [] (show_PCDATA a)
  -- |
type Metronome_Type = PCDATA
  -- |
read_Metronome_Type :: STM Result [Content i] Metronome_Type
read_Metronome_Type = do
  y ← read_ELEMENT "metronome-type"
  read_1 read_PCDATA (childs y)
  -- |
show_Metronome_Type :: Metronome_Type → [Content ()]
show_Metronome_Type a =
  show_ELEMENT "metronome-type" [] (show_PCDATA a)
  -- |
type Metronome_Dot = ()
  -- |
read_Metronome_Dot :: STM Result [Content i] Metronome_Dot
read_Metronome_Dot = read_ELEMENT "metronome-dot" ≫ return ()
  -- |
show_Metronome_Dot :: Metronome_Dot → [Content ()]
show_Metronome_Dot _ =
  show_ELEMENT "metronome-dot" [] []
  -- |
type Metronome_Beam = (Beam_Level, PCDATA)
  -- |
read_Metronome_Beam :: STM Result [Content i] Metronome_Beam
read_Metronome_Beam = do
  y ← read_ELEMENT "metronome-beam"
  y1 ← read_1 (read_DEFAULT "number" read_Beam_Level Beam_Level_1)
```

```
    (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Metronome_Beam :: Metronome_Beam → [Content ()]
show_Metronome_Beam (a, b) =
  show_ELEMENT "metronome-beam" (show_DEFAULT "number" show_Beam_Level a)
    (show_PCDATA b)
  -- |
type Metronome_Tuplet = ((Start_Stop, Maybe Yes_No, Maybe Metronome_Tuplet_),
  (Actual_Notes, Normal_Notes, Maybe (Normal_Type, [Normal_Dot])))
  -- |
read_Metronome_Tuplet :: Eq i ⇒ STM Result [Content i] Metronome_Tuplet
read_Metronome_Tuplet = do
  y ← read_ELEMENT "metronome-tuplet"
  y1 ← read_3 (read_REQUIRED "type" read_Start_Stop)
    (read_IMPLIED "bracket" read_Yes_No)
    (read_IMPLIED "show-number" read_Metronome_Tuplet_)
    (attributes y)
  y2 ← read_3 read_Actual_Notes read_Normal_Notes
    (read_MAYBE read_Metronome_Tuplet_aux1) (childs y)
  return (y1, y2)
  -- |
show_Metronome_Tuplet :: Metronome_Tuplet → [Content ()]
show_Metronome_Tuplet ((a, b, c), (d, e, f)) =
  show_ELEMENT "metronome-tuplet"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_IMPLIED "bracket" show_Yes_No b ++
      show_IMPLIED "show-number" show_Metronome_Tuplet_ c)
    (show_Actual_Notes d ++ show_Normal_Notes e ++
      show_MAYBE show_Metronome_Tuplet_aux1 f)
  -- |
read_Metronome_Tuplet_aux1 :: Eq i ⇒
  STM Result [Content i] (Normal_Type, [Normal_Dot])
read_Metronome_Tuplet_aux1 = do
  y1 ← read_Normal_Type
  y2 ← read_LIST read_Normal_Dot
  return (y1, y2)
  -- |
show_Metronome_Tuplet_aux1 :: (Normal_Type, [Normal_Dot]) → [Content ()]
show_Metronome_Tuplet_aux1 (a, b) =
  show_ELEMENT "metronome-tuplet" []
    (show_Normal_Type a ++ show_LIST show_Normal_Dot b)
  -- |
data Metronome_Tuplet_ = Metronome_Tuplet_1
  | Metronome_Tuplet_2
  | Metronome_Tuplet_3
    deriving (Eq, Show)
  -- |
read_Metronome_Tuplet_ :: Data.Char.String → Result Metronome_Tuplet_
read_Metronome_Tuplet_ "actual" = return Metronome_Tuplet_1
read_Metronome_Tuplet_ "both" = return Metronome_Tuplet_2
read_Metronome_Tuplet_ "none" = return Metronome_Tuplet_3
read_Metronome_Tuplet_ x = fail x
  -- |
show_Metronome_Tuplet_ :: Metronome_Tuplet_ → Data.Char.String
show_Metronome_Tuplet_ Metronome_Tuplet_1 = "actual"
show_Metronome_Tuplet_ Metronome_Tuplet_2 = "both"
```

```
show_Metronome_Tuplet_ Metronome_Tuplet_3 = "none"
```

Octave shifts indicate where notes are shifted up or down from their true pitched values because of printing difficulty. Thus a treble clef line noted with 8va will be indicated with an octave-shift down from the pitch data indicated in the notes. A size of 8 indicates one octave; a size of 15 indicates two octaves.

```
  -- |
type Octave_Shift = ((Octave_Shift_, Maybe Number_Level,
  CDATA, Print_Style), ())
  -- |
read_Octave_Shift :: STM Result [Content i] Octave_Shift
read_Octave_Shift = do
  y ← read_ELEMENT "octave-shift"
  y1 ← read_4 (read_REQUIRED "type" read_Octave_Shift_)
    (read_IMPLIED "number" read_Number_Level)
    (read_DEFAULT "size" read_CDATA "8")
    read_Print_Style (attributes y)
  return (y1, ())
  -- |
show_Octave_Shift :: Octave_Shift → [Content ()]
show_Octave_Shift ((a, b, c, d), _) =
  show_ELEMENT "octave-shift"
    (show_REQUIRED "type" show_Octave_Shift_ a ++
      show_IMPLIED "number" show_Number_Level b ++
      show_DEFAULT "size" show_CDATA c ++ show_Print_Style d) []
  -- |
data Octave_Shift_ = Octave_Shift_1 | Octave_Shift_2 | Octave_Shift_3
  deriving (Eq, Show)
  -- |
read_Octave_Shift_ :: Data.Char.String → Result Octave_Shift_
read_Octave_Shift_ "up" = return Octave_Shift_1
read_Octave_Shift_ "down" = return Octave_Shift_2
read_Octave_Shift_ "stop" = return Octave_Shift_3
read_Octave_Shift_ x = fail x
  -- |
show_Octave_Shift_ :: Octave_Shift_ → Data.Char.String
show_Octave_Shift_ Octave_Shift_1 = "up"
show_Octave_Shift_ Octave_Shift_2 = "down"
show_Octave_Shift_ Octave_Shift_3 = "stop"
```

The harp-pedals element is used to create harp pedal diagrams. The pedal-step and pedal-alter elements use the same values as the step and alter elements. For easiest reading, the pedal-tuning elements should follow standard harp pedal order, with pedal-step values of D, C, B, E, F, G, and A.

```
  -- |
type Harp_Pedals = (Print_Style, [Pedal_Tuning])
  -- |
read_Harp_Pedals :: Eq i ⇒ STM Result [Content i] Harp_Pedals
read_Harp_Pedals = do
  y ← read_ELEMENT "harp-pedals"
  y1 ← read_1 read_Print_Style (attributes y)
  y2 ← read_1 (read_LIST1 read_Pedal_Tuning) (childs y)
  return (y1, y2)
  -- |
show_Harp_Pedals :: Harp_Pedals → [Content ()]
show_Harp_Pedals (a, b) =
  show_ELEMENT "harp-pedals" (show_Print_Style a)
    (show_LIST show_Pedal_Tuning b)
  -- |
type Pedal_Tuning = (Pedal_Step, Pedal_Alter)
```

```
  -- |
read_Pedal_Tuning :: STM Result [Content i] Pedal_Tuning
read_Pedal_Tuning = do
  y ← read_ELEMENT "pedal-tuning"
  read_2 read_Pedal_Step read_Pedal_Alter (childs y)
  -- |
show_Pedal_Tuning :: Pedal_Tuning → [Content ()]
show_Pedal_Tuning (a, b) =
  show_ELEMENT "pedal-tuning" []
    (show_Pedal_Step a ++ show_Pedal_Alter b)
  -- |
type Pedal_Step = PCDATA
  -- |
read_Pedal_Step :: STM Result [Content i] Pedal_Step
read_Pedal_Step = do
  y ← read_ELEMENT "pedal-step"
  read_1 read_PCDATA (childs y)
  -- |
show_Pedal_Step :: Pedal_Step → [Content ()]
show_Pedal_Step a = show_ELEMENT "pedal-step" [] (show_Pedal_Step a)
  -- |
type Pedal_Alter = PCDATA
  -- |
read_Pedal_Alter :: STM Result [Content i] Pedal_Alter
read_Pedal_Alter = do
  y ← read_ELEMENT "pedal-alter"
  read_1 read_PCDATA (childs y)
  -- |
show_Pedal_Alter :: Pedal_Alter → [Content ()]
show_Pedal_Alter a = show_ELEMENT "pedal-alter" [] (show_Pedal_Alter a)
  -- |
type Damp = (Print_Style, ())
  -- |
read_Damp :: STM Result [Content i] Damp
read_Damp = do
  y ← read_ELEMENT "damp"
  y1 ← read_1 read_Print_Style (attributes y)
  return (y1, ())
  -- |
show_Damp :: Damp → [Content ()]
show_Damp (a, _) = show_ELEMENT "damp" (show_Print_Style a) []
  -- |
type Damp_All = (Print_Style, ())
  -- |
read_Damp_All :: STM Result [Content i] Damp_All
read_Damp_All = do
  y ← read_ELEMENT "damp-all"
  y1 ← read_1 read_Print_Style (attributes y)
  return (y1, ())
  -- |
show_Damp_All :: Damp_All → [Content ()]
show_Damp_All (a, _) = show_ELEMENT "damp-all" (show_Print_Style a) []
  -- |
type Eyeglasses = (Print_Style, ())
  -- |
read_Eyeglasses :: STM Result [Content i] Eyeglasses
read_Eyeglasses = do
  y ← read_ELEMENT "eyeglasses"
```

```
  y1 ← read_1 read_Print_Style (attributes y)
  return (y1, ())
  -- |
show_Eyeglasses :: Eyeglasses → [Content ()]
show_Eyeglasses (a, _) = show_ELEMENT "eyeglasses" (show_Print_Style a) []
```

Scordatura string tunings are represented by a series of accord elements. The tuning-step, tuning-alter, and tuning-octave elements are also used with the staff-tuning element, and are defined in the common.mod file. Strings are numbered from high to low.

```
  -- |
type Scordatura = [Accord]
  -- |
read_Scordatura :: Eq i ⇒ STM Result [Content i] Scordatura
read_Scordatura = do
  y ← read_ELEMENT "scordatura"
  read_1 (read_LIST read_Accord) (childs y)
  -- |
show_Scordatura :: Scordatura → [Content ()]
show_Scordatura a =
  show_ELEMENT "scordatura" [] (show_LIST show_Accord a)
  -- |
type Accord = (CDATA, (Tuning_Step, Maybe Tuning_Alter, Tuning_Octave))
  -- |
read_Accord :: STM Result [Content i] Accord
read_Accord = do
  y ← read_ELEMENT "accord"
  y1 ← read_1 (read_REQUIRED "string" read_CDATA) (attributes y)
  y2 ← read_3 read_Tuning_Step (read_MAYBE read_Tuning_Alter)
    read_Tuning_Octave (childs y)
  return (y1, y2)
  -- |
show_Accord :: Accord → [Content ()]
show_Accord (a, (b, c, d)) =
  show_ELEMENT "accord"
    (show_REQUIRED "string" show_CDATA a)
    (show_Tuning_Step b ++ show_MAYBE show_Tuning_Alter c ++
      show_Tuning_Octave d)
```

The image element is used to include graphical images in a score. The required source attribute is the URL for the image file. The required type attribute is the MIME type for the image file format. Typical choices include application/postscript, image/gif, image/jpeg, image/png, and image/tiff.

```
  -- |
type Image = ((CDATA, CDATA, Position, Halign, Valign_Image), ())
  -- |
read_Image :: STM Result [Content i] Image
read_Image = do
  y ← read_ELEMENT "image"
  y1 ← read_5 (read_REQUIRED "source" read_CDATA)
    (read_REQUIRED "type" read_CDATA) read_Position
    read_Halign read_Valign_Image (attributes y)
  return (y1, ())
  -- |
show_Image :: Image → [Content ()]
show_Image ((a, b, c, d, e), _) =
  show_ELEMENT "image"
    (show_REQUIRED "source" show_CDATA a ++
      show_REQUIRED "type" show_CDATA b ++
      show_Position c ++ show_Halign d ++ show_Valign_Image e) []
```

The accordion-registration element is use for accordion registration symbols. These are circular symbols divided horizontally into high, middle, and low sections that correspond to 4', 8', and 16' pipes. Each accordion-high, accordion-middle, and accordion-low element represents the presence of one or more dots in the registration diagram. The accordion-middle element may have text values of 1, 2, or 3, corresponding to have 1 to 3 dots in the middle section. An accordion-registration element needs to have at least one of the child elements present.

```
  -- |
type Accordion_Registration = (Print_Style,
  (Maybe Accordion_High, Maybe Accordion_Middle, Maybe Accordion_Low))
  -- |
read_Accordion_Registration :: STM Result [Content i] Accordion_Registration
read_Accordion_Registration = do
  y ← read_ELEMENT "accordion-registration"
  y1 ← read_1 read_Print_Style (attributes y)
  y2 ← read_3 (read_MAYBE read_Accordion_High)
    (read_MAYBE read_Accordion_Middle)
    (read_MAYBE read_Accordion_Low) (childs y)
  return (y1, y2)
  -- |
show_Accordion_Registration :: Accordion_Registration → [Content ()]
show_Accordion_Registration (a, (b, c, d)) =
  show_ELEMENT "accordion-registration"
    (show_Print_Style a)
    (show_MAYBE show_Accordion_High b ++
      show_MAYBE show_Accordion_Middle c ++
      show_MAYBE show_Accordion_Low d)
  -- |
type Accordion_High = ()
  -- |
read_Accordion_High :: STM Result [Content i] Accordion_High
read_Accordion_High = read_ELEMENT "accordion-high" ≫ return ()
  -- |
show_Accordion_High :: Accordion_High → [Content ()]
show_Accordion_High _ = show_ELEMENT "accordion-high" [] []
  -- |
type Accordion_Middle = PCDATA
  -- |
read_Accordion_Middle :: STM Result [Content i] Accordion_Middle
read_Accordion_Middle = do
  y ← read_ELEMENT "accordion-middle"
  read_1 read_PCDATA (childs y)
  -- |
show_Accordion_Middle :: Accordion_Middle → [Content ()]
show_Accordion_Middle a = show_ELEMENT "accordion-middle" [] (show_PCDATA a)
  -- |
type Accordion_Low = ()
  -- |
read_Accordion_Low :: STM Result [Content i] Accordion_Low
read_Accordion_Low = read_ELEMENT "accordion-low" ≫ return ()
  -- |
show_Accordion_Low :: Accordion_Low → [Content ()]
show_Accordion_Low _ = show_ELEMENT "accordion-low" [] []
```

The other-direction element is used to define any direction symbols not yet in the current version of the MusicXML format. This allows extended representation, though without application interoperability.

```
  -- |
type Other_Direction = ((Print_Object, Print_Style), PCDATA)
  -- |
```

```
read_Other_Direction :: STM Result [Content i] Other_Direction
read_Other_Direction = do
  y ← read_ELEMENT "other-direction"
  y1 ← read_2 read_Print_Object read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Other_Direction :: Other_Direction → [Content ()]
show_Other_Direction ((a, b), c) =
  show_ELEMENT "other-direction"
    (show_Print_Object a ++ show_Print_Style b)
    (show_PCDATA c)
```

An offset is represented in terms of divisions, and indicates where the direction will appear relative to the current musical location. This affects the visual appearance of the direction. If the sound attribute is "yes", then the offset affects playback too. If the sound attribute is "no", then any sound associated with the direction takes effect at the current location. The sound attribute is "no" by default for compatibility with earlier versions of the MusicXML format. If an element within a direction includes a default-x attribute, the offset value will be ignored when determining the appearance of that element.

```
  -- |
type Offset = (Maybe Yes_No, PCDATA)
  -- |
read_Offset :: STM Result [Content i] Offset
read_Offset = do
  y ← read_ELEMENT "offset"
  y1 ← read_1 (read_IMPLIED "sound" read_Yes_No) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Offset :: Offset → [Content ()]
show_Offset (a, b) =
  show_ELEMENT "offset" (show_IMPLIED "sound" show_Yes_No a)
    (show_PCDATA b)
```

*The harmony elements are based on Humdrum's **harm encoding, extended to support chord symbols in popular music as well as functional harmony analysis in classical music.*

If there are alternate harmonies possible, this can be specified using multiple harmony elements differentiated by type. Explicit harmonies have all note present in the music; implied have some notes missing but implied; alternate represents alternate analyses.

The harmony object may be used for analysis or for chord symbols. The print-object attribute controls whether or not anything is printed due to the harmony element. The print-frame attribute controls printing of a frame or fretboard diagram. The print-style entity sets the default for the harmony, but individual elements can override this with their own print-style values.

A harmony element can contain many stacked chords (e.g. V of II). A sequence of harmony-chord entities is used for this type of secondary function, where V of II would be represented by a harmony-chord with a V function followed by a harmony-chord with a II function.

```
  -- |
type Harmony_Chord = (Harmony_Chord_, Kind, Maybe Inversion,
    Maybe Bass, [Degree])
  -- |
read_Harmony_Chord :: Eq i ⇒ STM Result [Content i] Harmony_Chord
read_Harmony_Chord = do
  y1 ← read_Harmony_Chord_
  y2 ← read_Kind
  y3 ← read_MAYBE read_Inversion
  y4 ← read_MAYBE read_Bass
  y5 ← read_LIST read_Degree
  return (y1, y2, y3, y4, y5)
```

```
  -- |
show_Harmony_Chord :: Harmony_Chord → [Content ()]
show_Harmony_Chord (a, b, c, d, e) =
  (show_Harmony_Chord_ a ++ show_Kind b ++
    show_MAYBE show_Inversion c ++ show_MAYBE show_Bass d ++
    show_LIST show_Degree e)
  -- |
data Harmony_Chord_ = Harmony_Chord_1 Root
  | Harmony_Chord_2 Function
    deriving (Eq, Show)
  -- |
read_Harmony_Chord_ :: STM Result [Content i] Harmony_Chord_
read_Harmony_Chord_ =
  (read_Root >>= return · Harmony_Chord_1) `mplus`
  (read_Function >>= return · Harmony_Chord_2)
  -- |
show_Harmony_Chord_ :: Harmony_Chord_ → [Content ()]
show_Harmony_Chord_ (Harmony_Chord_1 a) = show_Root a
show_Harmony_Chord_ (Harmony_Chord_2 a) = show_Function a
  -- |
type Harmony = ((Maybe Harmony_, Print_Object, Maybe Yes_No,
    Print_Style, Placement),
  ([Harmony_Chord], Maybe Frame,
  Maybe Offset, Editorial, Maybe Staff))
  -- |
read_Harmony :: Eq i ⇒ STM Result [Content i] Harmony
read_Harmony = do
  y ← read_ELEMENT "harmony"
  y1 ← read_5 (read_IMPLIED "type" read_Harmony_) read_Print_Object
    (read_IMPLIED "print-frame" read_Yes_No)
    read_Print_Style read_Placement (attributes y)
  y2 ← read_5 (read_LIST read_Harmony_Chord) (read_MAYBE read_Frame)
    (read_MAYBE read_Offset) read_Editorial
    (read_MAYBE read_Staff) (childs y)
  return (y1, y2)
  -- fail 'harmony'
show_Harmony :: Harmony → [Content ()]
show_Harmony ((a, b, c, d, e), (f, g, h, i, j)) =
  show_ELEMENT "harmony"
    (show_IMPLIED "type" show_Harmony_ a ++ show_Print_Object b ++
      show_IMPLIED "print-frame" show_Yes_No c ++ show_Print_Style d ++
      show_Placement e)
    (show_LIST show_Harmony_Chord f ++ show_MAYBE show_Frame g ++
      show_MAYBE show_Offset h ++ show_Editorial i ++
      show_MAYBE show_Staff j)
  -- |
data Harmony_ = Harmony_1 | Harmony_2 | Harmony_3
  deriving (Eq, Show)
  -- |
read_Harmony_ :: Data.Char.String → Result Harmony_
read_Harmony_ "explicit" = return Harmony_1
read_Harmony_ "implied"  = return Harmony_2
read_Harmony_ "alternate" = return Harmony_3
read_Harmony_ x          = fail x
  -- |
show_Harmony_ :: Harmony_ → Data.Char.String
show_Harmony_ Harmony_1 = "explicit"
show_Harmony_ Harmony_2 = "implied"
```

```
show_Harmony_ Harmony_3 = "alternate"
```

A root is a pitch name like C, D, E, where a function is an indication like I, II, III. Root is generally used with pop chord symbols, function with classical functional harmony. It is an either/or choice to avoid data inconsistency. Function requires that the key be specified in the encoding.

The root element has a root-step and optional root-alter similar to the step and alter elements in a pitch, but renamed to distinguish the different musical meanings. The root-step text element indicates how the root should appear on the page if not using the element contents. In some chord styles, this will include the root-alter information as well. In that case, the print-object attribute of the root-alter element can be set to no. The root-alter location attribute indicates whether the alteration should appear to the left or the right of the root-step; it is right by default.

```
  -- |
type Root = (Root_Step, Maybe Root_Alter)
  -- |
read_Root :: STM Result [Content i] Root
read_Root = do
  y ← read_ELEMENT "root"
  read_2 read_Root_Step (read_MAYBE read_Root_Alter) (childs y)
  -- |
show_Root :: Root → [Content ()]
show_Root (a, b) =
  show_ELEMENT "root" []
    (show_Root_Step a ++ show_MAYBE show_Root_Alter b)
  -- |
type Root_Step = ((Maybe CDATA, Print_Style), PCDATA)
  -- |
read_Root_Step :: STM Result [Content i] Root_Step
read_Root_Step = do
  y ← read_ELEMENT "root-step"
  y1 ← read_2 (read_IMPLIED "text" read_CDATA)
    read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Root_Step :: Root_Step → [Content ()]
show_Root_Step ((a, b), c) =
  show_ELEMENT "root-step"
    (show_IMPLIED "text" show_CDATA a ++ show_Print_Style b)
    (show_PCDATA c)
  -- |
type Root_Alter = ((Print_Object, Print_Style, Maybe Left_Right), PCDATA)
  -- |
read_Root_Alter :: STM Result [Content i] Root_Alter
read_Root_Alter = do
  y ← read_ELEMENT "root-alter"
  y1 ← read_3 read_Print_Object read_Print_Style
    (read_IMPLIED "location" read_Left_Right)
    (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Root_Alter :: Root_Alter → [Content ()]
show_Root_Alter ((a, b, c), d) =
  show_ELEMENT "root-alter"
    (show_Print_Object a ++ show_Print_Style b ++
      show_IMPLIED "location" show_Left_Right c)
    (show_PCDATA d)
  -- |
```

```
type Function = (Print_Style, PCDATA)
  -- |
read_Function :: STM Result [Content i] Function
read_Function = do
  y ← read_ELEMENT "function"
  y1 ← read_1 read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Function :: Function → [Content ()]
show_Function (a, b) =
  show_ELEMENT "function" (show_Print_Style a) (show_PCDATA b)
```

Kind indicates the type of chord. Degree elements can then add, subtract, or alter from these starting points. Values include:

- *Triads: major (major third, perfect fifth) minor (minor third, perfect fifth) augmented (major third, augmented fifth) diminished (minor third, diminished fifth)*
- *Sevenths: dominant (major triad, minor seventh) major-seventh (major triad, major seventh) minor-seventh (minor triad, minor seventh) diminished-seventh (diminished triad, diminished seventh) augmented-seventh (augmented triad, minor seventh) half-diminished (diminished triad, minor seventh) major-minor (minor triad, major seventh)*
- *Sixths: major-sixth (major triad, added sixth) minor-sixth (minor triad, added sixth)*
- *Ninths: dominant-ninth (dominant-seventh, major ninth) major-ninth (major-seventh, major ninth) minor-ninth (minor-seventh, major ninth)*
- *11ths (usually as the basis for alteration): dominant-11th (dominant-ninth, perfect 11th) major-11th (major-ninth, perfect 11th) minor-11th (minor-ninth, perfect 11th)*
- *13ths (usually as the basis for alteration): dominant-13th (dominant-11th, major 13th) major-13th (major-11th, major 13th) minor-13th (minor-11th, major 13th)*
- *Suspended: suspended-second (major second, perfect fifth) suspended-fourth (perfect fourth, perfect fifth)*
- *Functional sixths: Neapolitan Italian French German*
- *Other: pedal (pedal-point bass) power (perfect fifth) Tristan*

The "other" kind is used when the harmony is entirely composed of add elements. The "none" kind is used to explicitly encode absence of chords or functional harmony.

The attributes are used to indicate the formatting of the symbol. Since the kind element is the constant in all the harmony-chord entities that can make up a polychord, many formatting attributes are here.

The use-symbols attribute is yes if the kind should be represented when possible with harmony symbols rather than letters and numbers. These symbols include:

major: a triangle, like Unicode 25B3 minor: -, like Unicode 002D augmented: +, like Unicode 002B diminished: $\hat{A}^{\circ}$, like Unicode 00B0 half-diminished: $\tilde{A}_{,}$, like Unicode 00F8

The text attribute describes how the kind should be spelled if not using symbols; it is ignored if use-symbols is yes. The stack-degrees attribute is yes if the degree elements should be stacked above each other. The parentheses-degrees attribute is yes if all the degrees should be in parentheses. The bracket-degrees attribute is yes if all the degrees should be in a bracket. If not specified, these values are implementation-specific. The alignment attributes are for the entire harmony-chord entity of which this kind element is a part.

```
type Kind = ((Maybe Yes_No, Maybe CDATA,
    Maybe Yes_No, Maybe Yes_No, Maybe Yes_No,
    Print_Style, Halign, Valign), PCDATA)
  -- |
read_Kind :: STM Result [Content i] Kind
read_Kind = do
  y ← read_ELEMENT "kind"
  y1 ← read_8 (read_IMPLIED "use-symbols" read_Yes_No)
```

```
      (read_IMPLIED "text" read_CDATA)
      (read_IMPLIED "stack-degrees" read_Yes_No)
      (read_IMPLIED "parentheses-degrees" read_Yes_No)
      (read_IMPLIED "bracket-degrees" read_Yes_No)
      read_Print_Style read_Halign read_Valign (attributes y)
    y2 ← read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Kind :: Kind → [Content ()]
show_Kind ((a, b, c, d, e, f, g, h), i) =
    show_ELEMENT "kind"
      (show_IMPLIED "use-symbols" show_Yes_No a ++
        show_IMPLIED "text" show_CDATA b ++
        show_IMPLIED "stack-degrees" show_Yes_No c ++
        show_IMPLIED "parentheses-degrees" show_Yes_No d ++
        show_IMPLIED "bracket-degrees" show_Yes_No e ++
        show_Print_Style f ++ show_Halign g ++ show_Valign h) (show_PCDATA i)
```

Inversion is a number indicating which inversion is used: 0 for root position, 1 for first inversion, etc.

```
type Inversion = (Print_Style, PCDATA)
    -- |
read_Inversion :: STM Result [Content i] Inversion
read_Inversion = do
    y ← read_ELEMENT "inversion"
    y1 ← read_1 read_Print_Style (attributes y)
    y2 ← read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Inversion :: Inversion → [Content ()]
show_Inversion (a, b) =
    show_ELEMENT "inversion" (show_Print_Style a) (show_PCDATA b)
```

Bass is used to indicate a bass note in popular music chord symbols, e.g. G/C. It is generally not used in functional harmony, as inversion is generally not used in pop chord symbols. As with root, it is divided into step and alter elements, similar to pitches. The attributes for bass-step and bass-alter work the same way as the corresponding root-step and root-alter attributes.

```
    -- |
type Bass = (Bass_Step, Maybe Bass_Alter)
    -- |
read_Bass :: STM Result [Content i] Bass
read_Bass = do
    y ← read_ELEMENT "bass"
    read_2 read_Bass_Step (read_MAYBE read_Bass_Alter) (childs y)
    -- |
show_Bass :: Bass → [Content ()]
show_Bass (a, b) =
    show_ELEMENT "bass" []
      (show_Bass_Step a ++ show_MAYBE show_Bass_Alter b)
    -- |
type Bass_Step = ((Maybe CDATA, Print_Style), PCDATA)
    -- |
read_Bass_Step :: STM Result [Content i] Bass_Step
read_Bass_Step = do
    y ← read_ELEMENT "bass-step"
    y1 ← read_2 (read_IMPLIED "text" read_CDATA)
      read_Print_Style (attributes y)
    y2 ← read_1 read_PCDATA (childs y)
    return (y1, y2)
```

```
  -- |
show_Bass_Step :: Bass_Step → [Content ()]
show_Bass_Step ((a, b), c) =
  show_ELEMENT "bass-step"
    (show_IMPLIED "text" show_CDATA a ++ show_Print_Style b)
    (show_PCDATA c)
  -- |
type Bass_Alter = ((Print_Object, Print_Style, Maybe Bass_Alter_), PCDATA)
  -- |
read_Bass_Alter :: STM Result [Content i] Bass_Alter
read_Bass_Alter = do
  y ← read_ELEMENT "bass-alter"
  y1 ← read_3 read_Print_Object read_Print_Style
    (read_IMPLIED "location" read_Bass_Alter_) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Bass_Alter :: Bass_Alter → [Content ()]
show_Bass_Alter ((a, b, c), d) =
  show_ELEMENT "bass-alter"
    (show_Print_Object a ++ show_Print_Style b ++
      show_IMPLIED "location" show_Bass_Alter_ c)
    (show_PCDATA d)
  -- | This is equivalent to left-right entity
data Bass_Alter_ = Bass_Alter_1 | Bass_Alter_2
    deriving (Eq, Show)
  -- |
read_Bass_Alter_ :: Data.Char.String → Result Bass_Alter_
read_Bass_Alter_ "left" = return Bass_Alter_1
read_Bass_Alter_ "right" = return Bass_Alter_2
read_Bass_Alter_ x      = fail x
  -- |
show_Bass_Alter_ :: Bass_Alter_ → Data.Char.String
show_Bass_Alter_ Bass_Alter_1 = "left"
show_Bass_Alter_ Bass_Alter_2 = "right"
```

The degree element is used to add, alter, or subtract individual notes in the chord. The degree-value element is a number indicating the degree of the chord (1 for the root, 3 for third, etc). The degree-alter element is like the alter element in notes: 1 for sharp, -1 for flat, etc. The degree-type element can be add, alter, or subtract. If the degree-type is alter or subtract, the degree-alter is relative to the degree already in the chord based on its kind element. If the degree-type is add, the degree-alter is relative to a dominant chord (major and perfect intervals except for a minor seventh). The print-object attribute can be used to keep the degree from printing separately when it has already taken into account in the text attribute of the kind element. The plus-minus attribute is used to indicate if plus and minus symbols should be used instead of sharp and flat symbols to display the degree alteration; it is no by default. The degree-value and degree-type text attributes specify how the value and type of the degree should be displayed.

A harmony of kind "other" can be spelled explicitly by using a series of degree elements together with a root.

```
  -- |
type Degree = (Print_Object, (Degree_Value, Degree_Alter, Degree_Type))
  -- |
read_Degree :: STM Result [Content i] Degree
read_Degree = do
  y ← read_ELEMENT "degree"
  y1 ← read_1 read_Print_Object (attributes y)
  y2 ← read_3 read_Degree_Value read_Degree_Alter
    read_Degree_Type (childs y)
  return (y1, y2)
```

```
  -- |
show_Degree :: Degree → [Content ()]
show_Degree (a, (b, c, d)) =
  show_ELEMENT "degree"
    (show_Print_Object a)
    (show_Degree_Value b ++ show_Degree_Alter c ++
      show_Degree_Type d)
  -- |
type Degree_Value = ((Maybe CDATA, Print_Style), PCDATA)
  -- |
read_Degree_Value :: STM Result [Content i] Degree_Value
read_Degree_Value = do
  y ← read_ELEMENT "degree-value"
  y1 ← read_2 (read_IMPLIED "text" read_CDATA)
    read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Degree_Value :: Degree_Value → [Content ()]
show_Degree_Value ((a, b), c) =
  show_ELEMENT "degree-value"
    (show_IMPLIED "type" show_CDATA a ++ show_Print_Style b)
    (show_PCDATA c)
  -- |
type Degree_Alter = ((Print_Style, Maybe Yes_No), PCDATA)
  -- |
read_Degree_Alter :: STM Result [Content i] Degree_Alter
read_Degree_Alter = do
  y ← read_ELEMENT "degree-alter"
  y1 ← read_2 read_Print_Style
    (read_IMPLIED "plus-minus" read_Yes_No) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Degree_Alter :: Degree_Alter → [Content ()]
show_Degree_Alter ((a, b), c) =
  show_ELEMENT "degree-alter"
    (show_Print_Style a ++ show_IMPLIED "plus-minus" show_Yes_No b)
    (show_PCDATA c)
  -- |
type Degree_Type = ((Maybe CDATA, Print_Style), PCDATA)
  -- |
read_Degree_Type :: STM Result [Content i] Degree_Type
read_Degree_Type = do
  y ← read_ELEMENT "degree-type"
  y1 ← read_2 (read_IMPLIED "text" read_CDATA)
    read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Degree_Type :: Degree_Type → [Content ()]
show_Degree_Type ((a, b), c) =
  show_ELEMENT "degree-type"
    (show_IMPLIED "type" show_CDATA a ++ show_Print_Style b)
    (show_PCDATA c)
```

The frame element represents a frame or fretboard diagram used together with a chord symbol. The representation is based on the NIFF guitar grid with additional information. The frame-strings and frame-frets elements give the overall size of the frame in vertical lines (strings) and horizontal spaces

(frets). The first-fret indicates which fret is shown in the top space of the frame; it is fret 1 if the element is not present. The optional text attribute indicates how this is represented in the fret diagram, while the location attribute indicates whether the text appears to the left or right of the frame. The frame-note element represents each note included in the frame. The definitions for string, fret, and fingering are found in the common.mod file. An open string will have a fret value of 0, while a muted string will not be associated with a frame-note element.

```
  -- |
type Frame =
  ((Position, Color, Halign, Valign, Maybe Tenths, Maybe Tenths),
  (Frame_Strings, Frame_Frets, Maybe First_Fret, [Frame_Note]))
  -- |
read_Frame :: Eq i ⇒ STM Result [Content i] Frame
read_Frame = do
  y ← read_ELEMENT "frame"
  y1 ← read_6 read_Position read_Color read_Halign read_Valign
    (read_IMPLIED "height" read_Tenths)
    (read_IMPLIED "width" read_Tenths) (attributes y)
  y2 ← read_4 read_Frame_Strings read_Frame_Frets
    (read_MAYBE read_First_Fret)
    (read_LIST read_Frame_Note) (childs y)
  return (y1, y2)
  -- |
show_Frame :: Frame → [Content ()]
show_Frame ((a, b, c, d, e, f), (g, h, i, j)) =
  show_ELEMENT "frame"
    (show_Position a ++ show_Color b ++ show_Halign c ++
      show_Valign d ++ show_IMPLIED "height" show_Tenths e ++
      show_IMPLIED "width" show_Tenths f)
    (show_Frame_Strings g ++ show_Frame_Frets h ++
      show_MAYBE show_First_Fret i ++ show_LIST show_Frame_Note j)
  -- |
type Frame_Strings = PCDATA
  -- |
read_Frame_Strings :: STM Result [Content i] Frame_Strings
read_Frame_Strings = do
  y ← read_ELEMENT "frame-strings"
  read_1 read_PCDATA (childs y)
  -- |
show_Frame_Strings :: Frame_Strings → [Content ()]
show_Frame_Strings a = show_ELEMENT "frame-strings" [] (show_PCDATA a)
  -- |
type Frame_Frets = PCDATA
  -- |
read_Frame_Frets :: STM Result [Content i] Frame_Frets
read_Frame_Frets = do
  y ← read_ELEMENT "frame-frets"
  read_1 read_PCDATA (childs y)
  -- |
show_Frame_Frets :: Frame_Frets → [Content ()]
show_Frame_Frets a = show_ELEMENT "frame-frets" [] (show_PCDATA a)
  -- |
type First_Fret = ((Maybe CDATA, Maybe Left_Right), PCDATA)
  -- |
read_First_Fret :: STM Result [Content i] First_Fret
read_First_Fret = do
  y ← read_ELEMENT "first-fret"
  y1 ← read_2 (read_IMPLIED "text" read_CDATA)
    (read_IMPLIED "location" read_Left_Right) (attributes y)
```

```
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_First_Fret :: First_Fret → [Content ()]
show_First_Fret ((a, b), c) =
  show_ELEMENT "first-fret"
    (show_IMPLIED "text" show_CDATA a ++
       show_IMPLIED "location" show_Left_Right b)
    (show_PCDATA c)
  -- |
type Frame_Note = (String, Fret, Maybe Fingering, Maybe Barre)
  -- |
read_Frame_Note :: STM Result [Content i] Frame_Note
read_Frame_Note = do
  y ← read_ELEMENT "frame-note"
  read_4 read_String read_Fret (read_MAYBE read_Fingering)
    (read_MAYBE read_Barre) (childs y)
  -- |
show_Frame_Note :: Frame_Note → [Content ()]
show_Frame_Note (a, b, c, d) =
  show_ELEMENT "frame-note" []
    (show_String a ++ show_Fret b ++
       show_MAYBE show_Fingering c ++ show_MAYBE show_Barre d)
```

The barre element indicates placing a finger over multiple strings on a single fret. The type is "start" for the lowest pitched string (e.g., the string with the highest MusicXML number) and is "stop" for the highest pitched string.

```
  -- |
type Barre = ((Start_Stop, Color), ())
  -- |
read_Barre :: STM Result [Content i] Barre
read_Barre = do
  y ← read_ELEMENT "barre"
  y1 ← read_2 (read_REQUIRED "type" read_Start_Stop)
    read_Color (attributes y)
  return (y1, ())
  -- |
show_Barre :: Barre → [Content ()]
show_Barre ((a, b), _) =
  show_ELEMENT "barre"
    (show_REQUIRED "type" show_Start_Stop a ++ show_Color b) []
```

The grouping element is used for musical analysis. When the element type is "start" or "single", it usually contains one or more feature elements. The number attribute is used for distinguishing between overlapping and hierarchical groupings. The member-of attribute allows for easy distinguishing of what grouping elements are in what hierarchy. Feature elements contained within a "stop" type of grouping may be ignored.

This element is flexible to allow for non-standard analyses. Future versions of the MusicXML format may add elements that can represent more standardized categories of analysis data, allowing for easier data sharing.

```
  -- |
type Grouping = ((Start_Stop_Single, CDATA, Maybe CDATA), [Feature])
  -- |
read_Grouping :: Eq i ⇒ STM Result [Content i] Grouping
read_Grouping = do
  y ← read_ELEMENT "grouping"
  y1 ← read_3 (read_REQUIRED "type" read_Start_Stop_Single)
    (read_DEFAULT "number" read_CDATA "1")
```

```
    (read_IMPLIED "member-of" read_CDATA)
    (attributes y)
  y2 ← read_1 (read_LIST read_Feature) (childs y)
  return (y1, y2)
  -- |
show_Grouping :: Grouping → [Content ()]
show_Grouping ((a, b, c), d) =
  show_ELEMENT "grouping" (show_REQUIRED "type" show_Start_Stop_Single a ++
    show_DEFAULT "number" show_CDATA b ++
    show_IMPLIED "member-of" show_CDATA c)
    (show_LIST show_Feature d)
  -- |
type Feature = (Maybe CDATA, PCDATA)
  -- |
read_Feature :: STM Result [Content i] Feature
read_Feature = do
  y ← read_ELEMENT "feature"
  y1 ← read_1 (read_IMPLIED "type" read_CDATA) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Feature :: Feature → [Content ()]
show_Feature (a, b) =
  show_ELEMENT "feature" (show_IMPLIED "type" show_CDATA a)
    (show_PCDATA b)
```

The print element contains general printing parameters, including the layout elements defined in the layout.mod file. The part-name-display and part-abbreviation-display elements used in the score.mod file may also be used here to change how a part name or abbreviation is displayed over the course of a piece. They take effect when the current measure or a succeeding measure starts a new system.

The new-system and new-page attributes indicate whether to force a system or page break, or to force the current music onto the same system or page as the preceding music. Normally this is the first music data within a measure. If used in multi-part music, they should be placed in the same positions within each part, or the results are undefined. The page-number attribute sets the number of a new page; it is ignored if new-page is not "yes". Version 2.0 adds a blank-page attribute. This is a positive integer value that specifies the number of blank pages to insert before the current measure. It is ignored if new-page is not "yes". These blank pages have no music, but may have text or images specified by the credit element. This is used to allow a combination of pages that are all text, or all text and images, together with pages of music.

Staff spacing between multiple staves is measured in tenths of staff lines (e.g. 100 = 10 staff lines). This is deprecated as of Version 1.1; the staff-layout element should be used instead. If both are present, the staff-layout values take priority.

Layout elements in a print statement only apply to the current page, system, staff, or measure. Music that follows continues to take the default values from the layout included in the defaults element.

```
  -- |
type Print = ((Maybe Tenths, Maybe Yes_No, Maybe Yes_No,
    Maybe CDATA, Maybe CDATA),
  (Maybe Page_Layout, Maybe System_Layout, [Staff_Layout],
  Maybe Measure_Layout, Maybe Measure_Numbering, Maybe Part_Name_Display,
  Maybe Part_Abbreviation_Display))
  -- |
read_Print :: Eq i ⇒ STM Result [Content i] Print
read_Print = do
  y ← read_ELEMENT "print"
  y1 ← read_5 (read_IMPLIED "staff-spacing" read_Tenths)
    (read_IMPLIED "new-system" read_Yes_No)
    (read_IMPLIED "new-page" read_Yes_No)
    (read_IMPLIED "blank-page" read_CDATA)
```

```
    (read_IMPLIED "page-number" read_CDATA) (attributes y)
  y2 ← read_7 (read_MAYBE read_Page_Layout) (read_MAYBE read_System_Layout)
    (read_LIST read_Staff_Layout) (read_MAYBE read_Measure_Layout)
    (read_MAYBE read_Measure_Numbering)
    (read_MAYBE read_Part_Name_Display)
    (read_MAYBE read_Part_Abbreviation_Display) (childs y)
  return (y1, y2)
  -- |
show_Print :: Print → [Content ()]
show_Print ((a, b, c, d, e), (f, g, h, i, j, k, l)) =
  show_ELEMENT "print"
    (show_IMPLIED "staff-spacing" show_Tenths a ++
      show_IMPLIED "new-system" show_Yes_No b ++
      show_IMPLIED "new-page" show_Yes_No c ++
      show_IMPLIED "blank-page" show_CDATA d ++
      show_IMPLIED "page-number" show_CDATA e)
    (show_MAYBE show_Page_Layout f ++ show_MAYBE show_System_Layout g ++
      show_LIST show_Staff_Layout h ++ show_MAYBE show_Measure_Layout i ++
      show_MAYBE show_Measure_Numbering j ++
      show_MAYBE show_Part_Name_Display k ++
      show_MAYBE show_Part_Abbreviation_Display l)
```

The measure-numbering element describes how measure numbers are displayed on this part. Values may be none, measure, or system. The number attribute from the measure element is used for printing. Measures with an implicit attribute set to "yes" never display a measure number, regardless of the measure-numbering setting.

```
  -- |
type Measure_Numbering = (Print_Style, PCDATA)
  -- |
read_Measure_Numbering :: Eq i ⇒ STM Result [Content i] Measure_Numbering
read_Measure_Numbering = do
  y ← read_ELEMENT "measure-numbering"
  y1 ← read_1 read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Measure_Numbering :: Measure_Numbering → [Content ()]
show_Measure_Numbering (a, b) =
  show_ELEMENT "measure-numbering"
    (show_Print_Style a) (show_PCDATA b)
```

The sound element contains general playback parameters. They can stand alone within a part/measure, or be a component element within a direction.

Tempo is expressed in quarter notes per minute. If 0, the sound-generating program should prompt the user at the time of compiling a sound (MIDI) file.

Dynamics (or MIDI velocity) are expressed as a percentage of the default forte value (90 for MIDI 1.0).

Dacapo indicates to go back to the beginning of the movement. When used it always has the value "yes".

Segno and dalsegno are used for backwards jumps to a segno sign; coda and tocoda are used for forward jumps to a coda sign. If there are multiple jumps, the value of these parameters can be used to name and distinguish them. If segno or coda is used, the divisions attribute can also be used to indicate the number of divisions per quarter note. Otherwise sound and MIDI generating programs may have to recompute this.

By default, a dalsegno or dacapo attribute indicates that the jump should occur the first time through, while a tocoda attribute indicates the jump should occur the second time through. The time that jumps occur can be changed by using the time-only attribute.

Forward-repeat is used when a forward repeat sign is implied, and usually follows a bar line. When used it always has the value of "yes".

The fine attribute follows the final note or rest in a movement with a da capo or dal segno direction. If numeric, the value represents the actual duration of the final note or rest, which can be ambiguous in written notation and different among parts and voices. The value may also be "yes" to indicate no change to the final duration.

If the sound element applies only one time through a repeat, the time-only attribute indicates which time to apply the sound element.

Pizzicato in a sound element effects all following notes. Yes indicates pizzicato, no indicates arco.

The pan and elevation attributes are deprecated in Version 2.0. The pan and elevation elements in the midi-instrument element should be used instead. The meaning of the pan and elevation attributes is the same as for the pan and elevation elements. If both are present, the mid-instrument elements take priority.

The damper-pedal, soft-pedal, and sostenuto-pedal attributes effect playback of the three common piano pedals and their MIDI controller equivalents. The yes value indicates the pedal is depressed; no indicates the pedal is released. A numeric value from 0 to 100 may also be used for half pedaling. This value is the percentage that the pedal is depressed. A value of 0 is equivalent to no, and a value of 100 is equivalent to yes.

MIDI instruments are changed using the midi-instrument element defined in the common.mod file.

The offset element is used to indicate that the sound takes place offset from the current score position. If the sound element is a child of a direction element, the sound offset element overrides the direction offset element if both elements are present. Note that the offset reflects the intended musical position for the change in sound. It should not be used to compensate for latency issues in particular hardware configurations.

```
  -- ** Sound
  -- |
type Sound = ((Maybe CDATA, Maybe CDATA, Maybe Yes_No,
  Maybe CDATA, Maybe CDATA, Maybe CDATA,
  Maybe CDATA, Maybe CDATA, Maybe Yes_No,
  Maybe CDATA, Maybe CDATA, Maybe Yes_No,
  Maybe CDATA, Maybe CDATA, Maybe Yes_No_Number,
  Maybe Yes_No_Number, Maybe Yes_No_Number),
  ([Midi_Instrument], Maybe Offset))
  -- |
read_Sound :: Eq i ⇒ STM Result [Content i] Sound
read_Sound = do
  y ← read_ELEMENT "sound"
  y1 ← read_17 (read_IMPLIED "tempo" read_CDATA)
    (read_IMPLIED "dynamics" read_CDATA)
    (read_IMPLIED "dacapo" read_Yes_No)
    (read_IMPLIED "segno" read_CDATA)
    (read_IMPLIED "dalsegno" read_CDATA)
    (read_IMPLIED "coda" read_CDATA)
    (read_IMPLIED "tocoda" read_CDATA)
    (read_IMPLIED "divisions" read_CDATA)
    (read_IMPLIED "forward-repeat" read_Yes_No)
    (read_IMPLIED "fine" read_CDATA)
    (read_IMPLIED "time-only" read_CDATA)
    (read_IMPLIED "pizzicato" read_Yes_No)
    (read_IMPLIED "pan" read_CDATA)
    (read_IMPLIED "elevation" read_CDATA)
    (read_IMPLIED "damper-pedal" read_Yes_No_Number)
    (read_IMPLIED "soft-pedal" read_Yes_No_Number)
    (read_IMPLIED "sostenuto-pedal" read_Yes_No_Number)
    (attributes y)
  y2 ← read_2 (read_LIST read_Midi_Instrument)
    (read_MAYBE read_Offset) (childs y)
  return (y1, y2)
  -- |
show_Sound :: Sound → [Content ()]
```

```
show_Sound ((a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q), (r, s)) =
  show_ELEMENT "sound" (show_IMPLIED "tempo" show_CDATA a ++
      show_IMPLIED "dynamics" show_CDATA b ++
      show_IMPLIED "dacapo" show_Yes_No c ++
      show_IMPLIED "segno" show_CDATA d ++
      show_IMPLIED "dalsegno" show_CDATA e ++
      show_IMPLIED "coda" show_CDATA f ++
      show_IMPLIED "tocoda" show_CDATA g ++
      show_IMPLIED "divisions" show_CDATA h ++
      show_IMPLIED "forward-repeat" show_Yes_No i ++
      show_IMPLIED "fine" show_CDATA j ++
      show_IMPLIED "time-only" show_CDATA k ++
      show_IMPLIED "pizzicato" show_Yes_No l ++
      show_IMPLIED "pan" show_CDATA m ++
      show_IMPLIED "elevation" show_CDATA n ++
      show_IMPLIED "damper-pedal" show_Yes_No_Number o ++
      show_IMPLIED "soft-pedal" show_Yes_No_Number p ++
      show_IMPLIED "sostenuto-pedal" show_Yes_No_Number q)
    (show_LIST show_Midi_Instrument r ++
      show_MAYBE show_Offset s)
```

2.6 Identity

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Identity where
import Text.XML.MusicXML.Common
import Text.XML.HaXml.Types (Content)
import Control.Monad (MonadPlus (..))
import Prelude (Maybe, Monad (..), Functor (..), Show, Eq, (·), (++))
```

The identify DTD module contains the identification element and its children, containing metadata about a score.

Identification contains basic metadata about the score. It includes the information in MuseData headers that may apply at a score-wide, movement-wide, or part-wide level. The creator, rights, source, and relation elements are based on Dublin Core.

```
  -- * Identification
  -- |
type Identification = ([Creator], [Rights], Maybe Encoding,
  Maybe Source, [Relation], Maybe Miscellaneous)
  -- |
read_Identification :: Eq i ⇒ STM Result [Content i] Identification
read_Identification = do
  y ← read_ELEMENT "identification"
  read_6 (read_LIST read_Creator) (read_LIST read_Rights)
    (read_MAYBE read_Encoding) (read_MAYBE read_Source)
    (read_LIST read_Relation) (read_MAYBE read_Miscellaneous)
    (childs y)
  -- |
show_Identification :: Identification → [Content ()]
show_Identification (a, b, c, d, e, f) =
```

```
  show_ELEMENT "identification" []
    (show_LIST show_Creator a ++ show_LIST show_Rights b ++
      show_MAYBE show_Encoding c ++ show_MAYBE show_Source d ++
      show_LIST show_Relation e ++ show_MAYBE show_Miscellaneous f)
  -- |
update_Identification :: ([Software], Encoding_Date) -> Identification -> Identification
update_Identification x (a, b, c, d, e, f) = (a, b, fmap (update_Encoding x) c, d, e, f)
```

The creator element is borrowed from Dublin Core. It is used for the creators of the score. The type attribute is used to distinguish different creative contributions. Thus, there can be multiple creators within an identification. Standard type values are composer, lyricist, and arranger. Other type values may be used for different types of creative roles. The type attribute should usually be used even if there is just a single creator element. The MusicXML format does not use the creator / contributor distinction from Dublin Core.

```
  -- ** Creator
  -- |
type Creator = (Maybe CDATA, PCDATA)
  -- |
read_Creator :: Eq i => STM Result [Content i] Creator
read_Creator = do
  y <- read_ELEMENT "creator"
  y1 <- read_1 (read_IMPLIED "type" read_CDATA) (attributes y)
  y2 <- read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Creator :: Creator -> [Content ()]
show_Creator (a, b) =
  show_ELEMENT "creator" (show_IMPLIED "type" show_CDATA a)
    (show_PCDATA b)
```

Rights is borrowed from Dublin Core. It contains copyright and other intellectual property notices. Words, music, and derivatives can have different types, so multiple rights tags with different type attributes are supported. Standard type values are music, words, and arrangement, but other types may be used. The type attribute is only needed when there are multiple rights elements.

```
  -- ** Rights
  -- |
type Rights = (Maybe CDATA, CDATA)
  -- |
read_Rights :: Eq i => STM Result [Content i] Rights
read_Rights = do
  y <- read_ELEMENT "rights"
  y1 <- read_1 (read_IMPLIED "type" read_CDATA) (attributes y)
  y2 <- read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Rights :: Rights -> [Content ()]
show_Rights (a, b) =
  show_ELEMENT "rights" (show_IMPLIED "type" show_CDATA a)
    (show_PCDATA b)
```

Encoding contains information about who did the digital encoding, when, with what software, and in what aspects. Standard type values for the encoder element are music, words, and arrangement, but other types may be used. The type attribute is only needed when there are multiple encoder elements.

The supports element indicates if the encoding supports a particular MusicXML element. This is recommended for elements like beam, stem, and accidental, where the absence of an element is ambiguous if you do not know if the encoding supports that element. For Version 2.0, the supports element is expanded to allow programs to indicate support for particular attributes or particular values. This lets applications communicate, for example, that all system and/or page breaks are contained in the MusicXML file.

```
  -- ** Encoding
  -- |
type Encoding = [Encoding_]
  -- |
read_Encoding :: Eq i => STM Result [Content i] Encoding
read_Encoding = do
  y <- read_ELEMENT "encoding"
  read_1 (read_LIST read_Encoding_) (childs y)
  -- |
show_Encoding :: Encoding -> [Content ()]
show_Encoding a = show_ELEMENT "encoding" [] (show_LIST show_Encoding_ a)
  -- |
update_Encoding :: ([Software], Encoding_Date) -> Encoding -> Encoding
update_Encoding (s, d) _ = (Encoding_1 d) : (fmap Encoding_3 s)
  -- |
data Encoding_ = Encoding_1 Encoding_Date
  | Encoding_2 Encoder
  | Encoding_3 Software
  | Encoding_4 Encoding_Description
  | Encoding_5 Supports
    deriving (Eq, Show)
  -- |
read_Encoding_ :: Eq i => STM Result [Content i] Encoding_
read_Encoding_ =
  (read_Encoding_Date >>= return . Encoding_1) `mplus`
  (read_Encoder >>= return . Encoding_2) `mplus`
  (read_Software >>= return . Encoding_3) `mplus`
  (read_Encoding_Description >>= return . Encoding_4) `mplus`
  (read_Supports >>= return . Encoding_5)
  -- |
show_Encoding_ :: Encoding_ -> [Content ()]
show_Encoding_ (Encoding_1 a) = show_Encoding_Date a
show_Encoding_ (Encoding_2 a) = show_Encoder a
show_Encoding_ (Encoding_3 a) = show_Software a
show_Encoding_ (Encoding_4 a) = show_Encoding_Description a
show_Encoding_ (Encoding_5 a) = show_Supports a
  -- |
type Encoding_Date = YYYY_MM_DD
  -- |
read_Encoding_Date :: Eq i => STM Result [Content i] Encoding_Date
read_Encoding_Date = do
  y <- read_ELEMENT "encoding-date"
  read_1 (read_YYYY_MM_DD) (childs y)
  -- |
show_Encoding_Date :: Encoding_Date -> [Content ()]
show_Encoding_Date a =
  show_ELEMENT "encoding-date" [] (show_YYYY_MM_DD a)
  -- |
type Encoder = (Maybe CDATA, PCDATA)
  -- |
read_Encoder :: Eq i => STM Result [Content i] Encoder
read_Encoder = do
  y <- read_ELEMENT "encoder"
  y1 <- read_1 (read_IMPLIED "type" read_CDATA) (attributes y)
  y2 <- read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Encoder :: Encoder -> [Content ()]
```

```
show_Encoder (a, b) =
  show_ELEMENT "encoder" (show_IMPLIED "type" show_CDATA a)
    (show_PCDATA b)
  -- |
type Software = PCDATA
  -- |
read_Software :: Eq i ⇒ STM Result [Content i] Software
read_Software = do
  y ← read_ELEMENT "software"
  read_1 read_PCDATA (childs y)
  -- |
show_Software :: Software → [Content ()]
show_Software a = show_ELEMENT "software" [] (show_PCDATA a)
  -- |
type Encoding_Description = PCDATA
  -- |
read_Encoding_Description :: STM Result [Content i] Encoding_Description
read_Encoding_Description = do
  y ← read_ELEMENT "encoding-description"
  read_1 read_PCDATA (childs y)
  -- |
show_Encoding_Description :: Encoding_Description → [Content ()]
show_Encoding_Description a =
  show_ELEMENT "encoding-description" [] (show_PCDATA a)
  -- |
type Supports = ((Yes_No, CDATA, Maybe CDATA, Maybe CDATA), ())
  -- |
read_Supports :: Eq i ⇒ STM Result [Content i] Supports
read_Supports = do
  y ← read_ELEMENT "supports"
  y1 ← read_4 (read_REQUIRED "type" read_Yes_No)
    (read_REQUIRED "element" read_CDATA)
    (read_IMPLIED "attribute" read_CDATA)
    (read_IMPLIED "value" read_CDATA) (attributes y)
  return (y1, ())
  -- |
show_Supports :: Supports → [Content ()]
show_Supports ((a, b, c, d), _) =
  show_ELEMENT "supports" (show_REQUIRED "type" show_Yes_No a ++
    show_REQUIRED "element" show_CDATA b ++
    show_IMPLIED "attribute" show_CDATA c ++
    show_IMPLIED "value" show_CDATA d) []
```

The source for the music that is encoded. This is similar to the Dublin Core source element.

```
  -- ** Source
  -- |
type Source = PCDATA
  -- |
read_Source :: STM Result [Content i] Source
read_Source = do
  y ← read_ELEMENT "source"
  read_1 read_PCDATA (childs y)
  -- |
show_Source :: Source → [Content ()]
show_Source a = show_ELEMENT "source" [] (show_PCDATA a)
```

A related resource for the music that is encoded. This is similar to the Dublin Core relation element. Standard type values are music, words, and arrangement, but other types may be used.

```
  -- ** Relation
  -- |
type Relation = (Maybe CDATA, CDATA)
  -- |
read_Relation :: STM Result [Content i] Relation
read_Relation = do
  y ← read_ELEMENT "relation"
  y1 ← read_1 (read_IMPLIED "type" read_CDATA) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Relation :: Relation → [Content ()]
show_Relation (a, b) =
  show_ELEMENT "relation" (show_IMPLIED "type" show_CDATA a)
    (show_PCDATA b)
```

If a program has other metadata not yet supported in the MusicXML format, it can go in the miscellaneous area.

```
  -- ** Miscellaneous
  -- |
type Miscellaneous = [Miscellaneous_Field]
  -- |
read_Miscellaneous :: Eq i ⇒ STM Result [Content i] Miscellaneous
read_Miscellaneous = do
  y ← read_ELEMENT "miscellaneous"
  read_1 (read_LIST read_Miscellaneous_Field) (childs y)
  -- |
show_Miscellaneous :: Miscellaneous → [Content ()]
show_Miscellaneous a =
  show_ELEMENT "miscellaneous" []
    (show_LIST show_Miscellaneous_Field a)
  -- |
type Miscellaneous_Field = (CDATA, PCDATA)
  -- |
read_Miscellaneous_Field :: STM Result [Content i] Miscellaneous_Field
read_Miscellaneous_Field = do
  y ← read_ELEMENT "miscellaneous-field"
  y1 ← read_1 (read_REQUIRED "name" read_CDATA) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Miscellaneous_Field :: Miscellaneous_Field → [Content ()]
show_Miscellaneous_Field (a, b) =
  show_ELEMENT "miscellaneous-field"
    (show_REQUIRED "name" show_CDATA a)
    (show_PCDATA b)
```

2.7 Layout

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Layout where
```

```
import Text.XML.MusicXML.Common hiding (Tenths, read_Tenths, show_Tenths)
import Text.XML.HaXml.Types (Content)
import Prelude (Maybe (..), Show, Eq, String, Monad (..), (++))
```

Version 1.1 of the MusicXML format added layout information for pages, systems, staffs, and measures. These layout elements joined the print and sound elements in providing formatting data as elements rather than attributes.

Everything is measured in tenths of staff space. Tenths are then scaled to millimeters within the scaling element, used in the defaults element at the start of a score. Individual staves can apply a scaling factor to adjust staff size. When a MusicXML element or attribute refers to tenths, it means the global tenths defined by the scaling element, not the local tenths as adjusted by the staff-size element.

Margins, page sizes, and distances are all measured in tenths to keep MusicXML data in a consistent coordinate system as much as possible. The translation to absolute units is done in the scaling element, which specifies how many millimeters are equal to how many tenths. For a staff height of 7 mm, millimeters would be set to 7 while tenths is set to 40. The ability to set a formula rather than a single scaling factor helps avoid roundoff errors.

```
  -- |
type Scaling = (Millimeters, Tenths)
  -- |
read_Scaling :: Eq i => STM Result [Content i] Scaling
read_Scaling = do
  y <- read_ELEMENT "scaling"
  read_2 read_Millimeters read_Tenths (childs y)
  -- |
show_Scaling :: Scaling -> [Content ()]
show_Scaling (a, b) =
  show_ELEMENT "scaling" [] (show_Millimeters a ++ show_Tenths b)
  -- |
type Millimeters = PCDATA
  -- |
read_Millimeters :: Eq i => STM Result [Content i] Millimeters
read_Millimeters = do
  y <- read_ELEMENT "millimeters"
  read_1 read_PCDATA (childs y)
  -- |
show_Millimeters :: Millimeters -> [Content ()]
show_Millimeters a = show_ELEMENT "millimeters" [] (show_PCDATA a)
  -- |
type Tenths = Layout_Tenths
  -- |
read_Tenths :: Eq i => STM Result [Content i] Tenths
read_Tenths = do
  y <- read_ELEMENT "tenths"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Tenths :: Tenths -> [Content ()]
show_Tenths a = show_ELEMENT "tenths" [] (show_Layout_Tenths a)
```

Margin elements are included within many of the larger layout elements.

```
  -- |
type Left_Margin = Layout_Tenths
  -- |
read_Left_Margin :: STM Result [Content i] Left_Margin
read_Left_Margin = do
  y <- read_ELEMENT "left-margin"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Left_Margin :: Left_Margin -> [Content ()]
```

```
show_Left_Margin a = show_ELEMENT "left-margin" [] (show_Layout_Tenths a)
  -- |
type Right_Margin = Layout_Tenths
  -- |
read_Right_Margin :: STM Result [Content i] Right_Margin
read_Right_Margin = do
  y ← read_ELEMENT "right-margin"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Right_Margin :: Right_Margin → [Content ()]
show_Right_Margin a = show_ELEMENT "right-margin" [] (show_Layout_Tenths a)
  -- |
type Top_Margin = Layout_Tenths
  -- |
read_Top_Margin :: STM Result [Content i] Top_Margin
read_Top_Margin = do
  y ← read_ELEMENT "top-margin"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Top_Margin :: Top_Margin → [Content ()]
show_Top_Margin a = show_ELEMENT "top-margin" [] (show_Layout_Tenths a)
  -- |
type Bottom_Margin = Layout_Tenths
  -- |
read_Bottom_Margin :: STM Result [Content i] Bottom_Margin
read_Bottom_Margin = do
  y ← read_ELEMENT "bottom-margin"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Bottom_Margin :: Bottom_Margin → [Content ()]
show_Bottom_Margin a = show_ELEMENT "bottom-margin" [] (show_Layout_Tenths a)
```

Page layout can be defined both in score-wide defaults and in the print element. Page margins are specified either for both even and odd pages, or via separate odd and even page number values. The type is not needed when used as part of a print element. If omitted when used in the defaults element, "both" is the default.

```
  -- |
type Page_Layout = (Maybe (Page_Height, Page_Width),
  Maybe (Page_Margins, Maybe Page_Margins))
  -- |
read_Page_Layout :: Eq i ⇒ STM Result [Content i] Page_Layout
read_Page_Layout = do
  y ← read_ELEMENT "page-layout"
  read_2 (read_MAYBE read_Page_Layout_aux1)
    (read_MAYBE read_Page_Layout_aux2) (childs y)
  -- |
show_Page_Layout :: Page_Layout → [Content ()]
show_Page_Layout (a, b) =
  show_ELEMENT "page-layout" [] (show_MAYBE show_Page_Layout_aux1 a ++
    show_MAYBE show_Page_Layout_aux2 b)
  -- |
read_Page_Layout_aux1 :: Eq i ⇒ STM Result [Content i] (Page_Height, Page_Width)
read_Page_Layout_aux1 = do
  y1 ← read_Page_Height
  y2 ← read_Page_Width
  return (y1, y2)
  -- |
show_Page_Layout_aux1 :: (Page_Height, Page_Width) → [Content ()]
```

```
show_Page_Layout_aux1 (a, b) = show_Page_Height a ++ show_Page_Width b
  -- |
read_Page_Layout_aux2 :: Eq i =>
  STM Result [Content i] (Page_Margins, Maybe Page_Margins)
read_Page_Layout_aux2 = do
  y1 <- read_Page_Margins
  y2 <- read_MAYBE read_Page_Margins
  return (y1, y2)
  -- |
show_Page_Layout_aux2 :: (Page_Margins, Maybe Page_Margins) -> [Content ()]
show_Page_Layout_aux2 (a, b) =
  show_Page_Margins a ++ show_MAYBE show_Page_Margins b
  -- |
type Page_Height = Layout_Tenths
  -- |
read_Page_Height :: Eq i => STM Result [Content i] Page_Height
read_Page_Height = do
  y <- read_ELEMENT "page-height"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Page_Height :: Page_Height -> [Content ()]
show_Page_Height a = show_ELEMENT "page-height" [] (show_Layout_Tenths a)
  -- |
type Page_Width = Layout_Tenths
  -- |
read_Page_Width :: Eq i => STM Result [Content i] Page_Width
read_Page_Width = do
  y <- read_ELEMENT "page-width"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Page_Width :: Page_Width -> [Content ()]
show_Page_Width a = show_ELEMENT "page-width" [] (show_Layout_Tenths a)
  -- |
type Page_Margins = (Maybe Page_Margins_,
  (Left_Margin, Right_Margin, Top_Margin, Bottom_Margin))
  -- |
read_Page_Margins :: Eq i => STM Result [Content i] Page_Margins
read_Page_Margins = do
  y <- read_ELEMENT "page-margins"
  y1 <- read_1 (read_IMPLIED "type" read_Page_Margins_) (attributes y)
  y2 <- read_4 read_Left_Margin read_Right_Margin
    read_Top_Margin read_Bottom_Margin (childs y)
  return (y1, y2)
  -- |
show_Page_Margins :: Page_Margins -> [Content ()]
show_Page_Margins (a, (b, c, d, e)) =
  show_ELEMENT "page-margins" (show_IMPLIED "type" show_Page_Margins_ a)
    (show_Left_Margin b ++ show_Right_Margin c ++
      show_Top_Margin d ++ show_Bottom_Margin e)
  -- |
data Page_Margins_ = Page_Margins_1 | Page_Margins_2 | Page_Margins_3
  deriving (Eq, Show)
  -- |
read_Page_Margins_ :: Prelude.String -> Result Page_Margins_
read_Page_Margins_ "odd" = return Page_Margins_1
read_Page_Margins_ "even" = return Page_Margins_2
read_Page_Margins_ "both" = return Page_Margins_3
read_Page_Margins_ x = fail x
```

```
  -- |
show_Page_Margins_ :: Page_Margins_ → Prelude.String
show_Page_Margins_ Page_Margins_1 = "odd"
show_Page_Margins_ Page_Margins_2 = "even"
show_Page_Margins_ Page_Margins_3 = "both"
```

System layout includes left and right margins and the vertical distance from the previous system. Margins are relative to the page margins. Positive values indent and negative values reduce the margin size. The system distance is measured from the bottom line of the previous system to the top line of the current system. It is ignored for the first system on a page. The top system distance is measured from the page's top margin to the top line of the first system. It is ignored for all but the first system on a page.

Sometimes the sum of measure widths in a system may not equal the system width specified by the layout elements due to roundoff or other errors. The behavior when reading MusicXML files in these cases is application-dependent. For instance, applications may find that the system layout data is more reliable than the sum of the measure widths, and adjust the measure widths accordingly.

```
  -- |
type System_Layout = (Maybe System_Margins,
  Maybe System_Distance, Maybe Top_System_Distance)
  -- |
read_System_Layout :: STM Result [Content i] System_Layout
read_System_Layout = do
  y ← read_ELEMENT "system-layout"
  read_3 (read_MAYBE read_System_Margins)
    (read_MAYBE read_System_Distance)
    (read_MAYBE read_Top_System_Distance)
    (childs y)
  -- |
show_System_Layout :: System_Layout → [Content ()]
show_System_Layout (a, b, c) =
  show_ELEMENT "system-layout" []
    (show_MAYBE show_System_Margins a ++
      show_MAYBE show_System_Distance b ++
      show_MAYBE show_Top_System_Distance c)
  -- |
type System_Margins = (Left_Margin, Right_Margin)
  -- |
read_System_Margins :: STM Result [Content i] System_Margins
read_System_Margins = do
  y ← read_ELEMENT "system-margins"
  read_2 read_Left_Margin read_Right_Margin (childs y)
  -- |
show_System_Margins :: System_Margins → [Content ()]
show_System_Margins (a, b) =
  show_ELEMENT "system-margins" []
    (show_Left_Margin a ++ show_Right_Margin b)
  -- |
type System_Distance = Layout_Tenths
  -- |
read_System_Distance :: STM Result [Content i] System_Distance
read_System_Distance = do
  y ← read_ELEMENT "system-distance"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_System_Distance :: System_Distance → [Content ()]
show_System_Distance a =
  show_ELEMENT "system-distance" [] (show_Layout_Tenths a)
  -- |
type Top_System_Distance = Layout_Tenths
```

```
  -- |
read_Top_System_Distance :: STM Result [Content i] Top_System_Distance
read_Top_System_Distance = do
  y ← read_ELEMENT "top-system-distance"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Top_System_Distance :: Top_System_Distance → [Content ()]
show_Top_System_Distance a =
  show_ELEMENT "top-system-distance" [] (show_Layout_Tenths a)
```

Staff layout includes the vertical distance from the bottom line of the previous staff in this system to the top line of the staff specified by the number attribute. The optional number attribute refers to staff numbers within the part, from top to bottom on the system. A value of 1 is assumed if not present. When used in the defaults element, the values apply to all parts. This value is ignored for the first staff in a system.

```
  -- |
type Staff_Layout = (Maybe CDATA, Maybe Staff_Distance)
  -- |
read_Staff_Layout :: STM Result [Content i] Staff_Layout
read_Staff_Layout = do
  y ← read_ELEMENT "staff-layout"
  y1 ← read_1 (read_IMPLIED "number" read_CDATA) (attributes y)
  y2 ← read_1 (read_MAYBE read_Staff_Distance) (childs y)
  return (y1, y2)
  -- |
show_Staff_Layout :: Staff_Layout → [Content ()]
show_Staff_Layout (a, b) =
  show_ELEMENT "staff-layout"
    (show_IMPLIED "number" show_CDATA a)
    (show_MAYBE show_Staff_Distance b)
  -- |
type Staff_Distance = Layout_Tenths
  -- |
read_Staff_Distance :: STM Result [Content i] Staff_Distance
read_Staff_Distance = do
  y ← read_ELEMENT "staff-distance"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Staff_Distance :: Staff_Distance → [Content ()]
show_Staff_Distance a =
  show_ELEMENT "staff-distance" [] (show_Layout_Tenths a)
```

Measure layout includes the horizontal distance from the previous measure. This value is only used for systems where there is horizontal whitespace in the middle of a system, as in systems with codas. To specify the measure width, use the width attribute of the measure element.

```
  -- |
type Measure_Layout = Maybe Measure_Distance
  -- |
read_Measure_Layout :: Eq i ⇒ STM Result [Content i] Measure_Layout
read_Measure_Layout = do
  y ← read_ELEMENT "measure-layout"
  read_1 (read_MAYBE read_Measure_Distance) (childs y)
  -- |
show_Measure_Layout :: Measure_Layout → [Content ()]
show_Measure_Layout a =
  show_ELEMENT "measure-layout" [] (show_MAYBE show_Measure_Distance a)
  -- |
type Measure_Distance = Layout_Tenths
```

```
  -- |
read_Measure_Distance :: Eq i ⇒ STM Result [Content i] Measure_Distance
read_Measure_Distance = do
  y ← read_ELEMENT "measure-distance"
  read_1 read_Layout_Tenths (childs y)
  -- |
show_Measure_Distance :: Measure_Distance → [Content ()]
show_Measure_Distance a =
  show_ELEMENT "measure-distance" [] (show_Layout_Tenths a)
```

The appearance element controls general graphical settings for the music's final form appearance on a printed page of display. Currently this includes support for line widths and definitions for note sizes, plus an extension element for other aspects of appearance.

The line-width element indicates the width of a line type in tenths. The type attribute defines what type of line is being defined. Values include beam, bracket, dashes, enclosure, ending, extend, heavy barline, leger, light barline, octave shift, pedal, slur middle, slur tip, staff, stem, tie middle, tie tip, tuplet bracket, and wedge. The text content is expressed in tenths.

The note-size element indicates the percentage of the regular note size to use for notes with a cue and large size as defined in the type element. The grace type is used for notes of cue size that that include a grace element. The cue type is used for all other notes with cue size, whether defined explicitly or implicitly via a cue element. The large type is used for notes of large size. The text content represent the numeric percentage. A value of 100 would be identical to the size of a regular note as defined by the music font.

The other-appearance element is used to define any graphical settings not yet in the current version of the MusicXML format. This allows extended representation, though without application interoperability.

```
  -- |
type Appearance = ([Line_Width], [Note_Size], [Other_Appearance])
  -- |
read_Appearance :: Eq i ⇒ STM Result [Content i] Appearance
read_Appearance = do
  y ← read_ELEMENT "appearance"
  read_3 (read_LIST read_Line_Width) (read_LIST read_Note_Size)
    (read_LIST read_Other_Appearance) (childs y)
  -- |
show_Appearance :: Appearance → [Content ()]
show_Appearance (a, b, c) =
  show_ELEMENT "appearance" [] (show_LIST show_Line_Width a ++
    show_LIST show_Note_Size b ++
    show_LIST show_Other_Appearance c)
  -- |
type Line_Width = (CDATA, Layout_Tenths)
  -- |
read_Line_Width :: STM Result [Content i] Line_Width
read_Line_Width = do
  y ← read_ELEMENT "line-width"
  y1 ← read_1 (read_REQUIRED "type" read_CDATA) (attributes y)
  y2 ← read_1 read_Layout_Tenths (childs y)
  return (y1, y2)
  -- |
show_Line_Width :: Line_Width → [Content ()]
show_Line_Width (a, b) =
  show_ELEMENT "line-width" (show_REQUIRED "type" show_CDATA a)
    (show_Layout_Tenths b)
  -- |
type Note_Size = (Note_Size_, PCDATA)
  -- |
read_Note_Size :: STM Result [Content i] Note_Size
read_Note_Size = do
```

```
    y ← read_ELEMENT "note-size"
    y1 ← read_1 (read_REQUIRED "type" read_Note_Size_) (attributes y)
    y2 ← read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Note_Size :: Note_Size → [Content ()]
show_Note_Size (a, b) =
    show_ELEMENT "note-size" (show_REQUIRED "type" show_Note_Size_ a)
      (show_PCDATA b)
    -- |
data Note_Size_ = Note_Size_1 | Note_Size_2 | Note_Size_3
    deriving (Eq, Show)
    -- |
read_Note_Size_ :: Prelude.String → Result Note_Size_
read_Note_Size_ "cue" = return Note_Size_1
read_Note_Size_ "grace" = return Note_Size_2
read_Note_Size_ "large" = return Note_Size_3
read_Note_Size_ x = fail x
    -- |
show_Note_Size_ :: Note_Size_ → Prelude.String
show_Note_Size_ Note_Size_1 = "cue"
show_Note_Size_ Note_Size_2 = "grace"
show_Note_Size_ Note_Size_3 = "large"
    -- |
type Other_Appearance = (CDATA, PCDATA)
    -- |
read_Other_Appearance :: STM Result [Content i] Other_Appearance
read_Other_Appearance = do
    y ← read_ELEMENT "other-appearance"
    y1 ← read_1 (read_REQUIRED "type" read_CDATA) (attributes y)
    y2 ← read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Other_Appearance :: Other_Appearance → [Content ()]
show_Other_Appearance (a, b) =
    show_ELEMENT "other-appearance" (show_REQUIRED "type" show_CDATA a)
      (show_PCDATA b)
```

2.8 Link

```
    -- |
    -- Maintainer : silva.samuel@alumni.uminho.pt
    -- Stability : experimental
    -- Portability: HaXML
    --
module Text.XML.MusicXML.Link where
import Text.XML.MusicXML.Common
import Text.XML.HaXml.Types (Content, Attribute)
import Prelude (Maybe, Show, Eq, Monad (..), String, (++))
```

The link-attributes entity includes all the simple XLink attributes supported in the MusicXML format.

```
    -- * XLink
    -- |
```

```
type Link_Attributes = (CDATA, CDATA, CDATA,
  Maybe CDATA, Maybe CDATA,
  Link_Attributes_A, Link_Attributes_B)
  -- |
read_Link_Attributes :: STM Result [Attribute] Link_Attributes
read_Link_Attributes = do
  y1 ← read_FIXED "xmlns:xlink" read_CDATA "http://www.w3.org/1999/xlink"
  y2 ← read_REQUIRED "xlink:href" read_CDATA
  y3 ← read_FIXED "xlink:type" read_CDATA "simple"
  y4 ← read_IMPLIED "xlink:role" read_CDATA
  y5 ← read_IMPLIED "xlink:title" read_CDATA
  y6 ← read_DEFAULT "xlink:show" read_Link_Attributes_A Link_Attributes_2
  y7 ← read_DEFAULT "xlink:actuate" read_Link_Attributes_B Link_Attributes_6
  return (y1, y2, y3, y4, y5, y6, y7)
  -- |
show_Link_Attributes :: Link_Attributes → [Attribute]
show_Link_Attributes (a, b, c, d, e, f, g) =
  show_FIXED "xmlns:xlink" show_CDATA a ++
  show_REQUIRED "xlink:href" show_CDATA b ++
  show_FIXED "xlink:type" show_CDATA c ++
  show_IMPLIED "xlink:role" show_CDATA d ++
  show_IMPLIED "xlink:title" show_CDATA e ++
  show_DEFAULT "xlink:show" show_Link_Attributes_A f ++
  show_DEFAULT "xlink:actuate" show_Link_Attributes_B g
  -- |
data Link_Attributes_A = Link_Attributes_1
  | Link_Attributes_2
  | Link_Attributes_3
  | Link_Attributes_4
  | Link_Attributes_5
    deriving (Eq, Show)
  -- |
read_Link_Attributes_A :: Prelude.String → Result Link_Attributes_A
read_Link_Attributes_A "new" = return Link_Attributes_1
read_Link_Attributes_A "replace" = return Link_Attributes_2
read_Link_Attributes_A "embed" = return Link_Attributes_3
read_Link_Attributes_A "other" = return Link_Attributes_4
read_Link_Attributes_A "none" = return Link_Attributes_5
read_Link_Attributes_A x = fail x
  -- |
show_Link_Attributes_A :: Link_Attributes_A → Prelude.String
show_Link_Attributes_A Link_Attributes_1 = "new"
show_Link_Attributes_A Link_Attributes_2 = "replace"
show_Link_Attributes_A Link_Attributes_3 = "embed"
show_Link_Attributes_A Link_Attributes_4 = "other"
show_Link_Attributes_A Link_Attributes_5 = "none"
  -- |
data Link_Attributes_B = Link_Attributes_6
  | Link_Attributes_7
  | Link_Attributes_8
  | Link_Attributes_9
    deriving (Eq, Show)
  -- |
read_Link_Attributes_B :: Prelude.String → Result Link_Attributes_B
read_Link_Attributes_B "onRequest" = return Link_Attributes_6
read_Link_Attributes_B "onLoad" = return Link_Attributes_7
read_Link_Attributes_B "other" = return Link_Attributes_8
read_Link_Attributes_B "none" = return Link_Attributes_9
```

```
read_Link_Attributes_B x = fail x
  -- |
show_Link_Attributes_B :: Link_Attributes_B → Prelude.String
show_Link_Attributes_B Link_Attributes_6 = "onRequest"
show_Link_Attributes_B Link_Attributes_7 = "onLoad"
show_Link_Attributes_B Link_Attributes_8 = "other"
show_Link_Attributes_B Link_Attributes_9 = "none"
```

The element and position attributes are new as of Version 2.0. They allow for bookmarks and links to be positioned at higher resolution than the level of music-data elements. When no element and position attributes are present, the bookmark or link element refers to the next sibling element in the MusicXML file. The element attribute specifies an element type for a descendant of the next sibling element that is not a link or bookmark. The position attribute specifies the position of this descendant element, where the first position is 1. The position attribute is ignored if the element attribute is not present. For instance, an element value of "beam" and a position value of "2" defines the link or bookmark to refer to the second beam descendant of the next sibling element that is not a link or bookmark. This is equivalent to an XPath test of [.//beam[2]] done in the context of the sibling element.

```
  -- * Link
  -- |
type Link = ((Link_Attributes,
  Maybe CDATA, Maybe CDATA, Maybe CDATA, Position), ())
  -- |
read_Link :: Eq i ⇒ STM Result [Content i] Link
read_Link = do
  y ← read_ELEMENT "link"
  y1 ← read_5 read_Link_Attributes (read_IMPLIED "name" read_CDATA)
    (read_IMPLIED "element" read_CDATA)
    (read_IMPLIED "position" read_CDATA)
    read_Position (attributes y)
  return (y1, ())
  -- |
show_Link :: Link → [Content ()]
show_Link ((a, b, c, d, e), _) =
  show_ELEMENT "link" (show_Link_Attributes a ++
    show_IMPLIED "name" show_CDATA b ++
    show_IMPLIED "element" show_CDATA c ++
    show_IMPLIED "position" show_CDATA d ++
    show_Position e) []
  -- * Bookmark
  -- |
type Bookmark = ((ID, Maybe CDATA, Maybe CDATA, Maybe CDATA), ())
  -- |
read_Bookmark :: Eq i ⇒ STM Result [Content i] Bookmark
read_Bookmark = do
  y ← read_ELEMENT "bookmark"
  y1 ← read_4 (read_REQUIRED "id" read_ID)
    (read_IMPLIED "name" read_CDATA)
    (read_IMPLIED "element" read_CDATA)
    (read_IMPLIED "position" read_CDATA)
    (attributes y)
  return (y1, ())
  -- |
show_Bookmark :: Bookmark → [Content ()]
show_Bookmark ((a, b, c, d), _) =
  show_ELEMENT "bookmark" (show_REQUIRED "id" show_ID a ++
    show_IMPLIED "name" show_CDATA b ++
    show_IMPLIED "element" show_CDATA c ++
    show_IMPLIED "position" show_CDATA d) []
```

2.9 MusicXML

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML (
  module Text.XML.MusicXML,
  module Text.XML.MusicXML.Common,
  module Text.XML.MusicXML.Attributes,
  module Text.XML.MusicXML.Identity,
  module Text.XML.MusicXML.Barline,
  module Text.XML.MusicXML.Link,
  module Text.XML.MusicXML.Direction,
  module Text.XML.MusicXML.Layout,
  module Text.XML.MusicXML.Note,
  module Text.XML.MusicXML.Score,
  module Text.XML.MusicXML.Partwise,
  module Text.XML.MusicXML.Timewise,
) where
import Prelude (IO, Int, String, FilePath,
  Monad (..), Show (..), Eq (..), Ord (..),
  Maybe (..), Bool (..),
  [·,·], maybe, otherwise, fromEnum, π2, mapM,
  readFile, writeFile,
  (·), (++), (+))                          -- base package
import qualified Data.Map as · ↦ ·          -- containers package
import Control.Monad (MonadPlus (..))       -- base package
import System.Time (CalendarTime (..),
  getClockTime, toCalendarTime)             -- old-time package
import System.Directory (doesFileExist)     -- directory package
import Text.PrettyPrint.HughesPJ            -- pretty package
import Text.XML.HaXml.Types                 -- HaXml package
import Text.XML.HaXml.Parse (xmlParse')     -- HaXml package
import Text.XML.HaXml.Pretty (document)     -- HaXml package
import Text.XML.HaXml.Posn (Posn, noPos)    -- HaXml package
import Text.XML.MusicXML.Common hiding
  (Tenths, read_Tenths, show_Tenths,
  Directive, read_Directive, show_Directive) -- MusicXML package
import Text.XML.MusicXML.Attributes         -- MusicXML package
import Text.XML.MusicXML.Barline            -- MusicXML package
import Text.XML.MusicXML.Link               -- MusicXML package
import Text.XML.MusicXML.Direction          -- MusicXML package
import Text.XML.MusicXML.Identity           -- MusicXML package
import Text.XML.MusicXML.Layout             -- MusicXML package
import Text.XML.MusicXML.Note               -- MusicXML package
import Text.XML.MusicXML.Score hiding
  (Opus, read_Opus, show_Opus)              -- MusicXML package
import Text.XML.MusicXML.Partwise hiding
  (doctype, Part, read_Part, show_Part,
  Measure, read_Measure, show_Measure)      -- MusicXML package
import Text.XML.MusicXML.Timewise hiding
  (doctype, Part, read_Part, show_Part,
```

```
  Measure, read_Measure, show_Measure)          -- MusicXML package
import Text.XML.MusicXML.Opus hiding
  (doctype)                                     -- MusicXML package
import Text.XML.MusicXML.Container hiding
  (doctype)                                     -- MusicXML package
import qualified Text.XML.MusicXML.Partwise  as Partwise     -- MusicXML package
import qualified Text.XML.MusicXML.Timewise  as Timewise     -- MusicXML package
import qualified Text.XML.MusicXML.Opus      as Opus         -- MusicXML package
import qualified Text.XML.MusicXML.Container as Container    -- MusicXML package


  -- * MusicXML
  -- |
data ScoreDoc = Partwise Score_Partwise
   | Timewise Score_Timewise
     deriving (Eq, Show)
data MusicXMLDoc = Score ScoreDoc
   | Opus Opus
   | Container Container
     deriving (Eq, Show)
  -- |
data MusicXMLRec = MusicXMLRec (Map.Map FilePath MusicXMLDoc)
     deriving (Eq, Show)


  -- |
read_DOCUMENT :: STM Result [Content Posn] a → Document Posn → Result a
read_DOCUMENT r (Document _ _ x _) = stateM r [CElem x noPos] ⪢= (return · π2)
  -- |
read_MusicXML_Partwise :: Document Posn → Result Score_Partwise
read_MusicXML_Partwise = read_DOCUMENT read_Score_Partwise
  -- |
read_MusicXML_Timewise :: Document Posn → Result Score_Timewise
read_MusicXML_Timewise = read_DOCUMENT read_Score_Timewise
  -- |
read_MusicXML_Opus :: Document Posn → Result Opus
read_MusicXML_Opus = read_DOCUMENT read_Opus
  -- |
read_MusicXML_Container :: Document Posn → Result Container
read_MusicXML_Container = read_DOCUMENT read_Container
  -- |
show_DOCUMENT :: DocTypeDecl → (t → [Content i]) → t → Result (Document i)
show_DOCUMENT doct f doc =
  case f doc of
    [(CElem processed _)] →
      return (Document (Prolog (Just xmldecl) []
        (Just doct) []) [] processed [])
    _ → fail "internal error"
  -- |
show_MusicXML_Partwise :: Score_Partwise → Result (Document ())
show_MusicXML_Partwise =
  show_DOCUMENT Partwise.doctype show_Score_Partwise
  -- |
show_MusicXML_Timewise :: Score_Timewise → Result (Document ())
show_MusicXML_Timewise =
  show_DOCUMENT Partwise.doctype show_Score_Timewise
  -- |
show_MusicXML_Opus :: Opus → Result (Document ())
show_MusicXML_Opus x =
  show_DOCUMENT Opus.doctype show_Opus x
```

```
  -- |
show_MusicXML_Container :: Container → Result (Document ())
show_MusicXML_Container x =
  show_DOCUMENT Container.doctype show_Container x
  -- |
update_MusicXML_Partwise :: ([Software], Encoding_Date) →
  Score_Partwise → Score_Partwise
update_MusicXML_Partwise = update_Score_Partwise
  -- |
update_MusicXML_Timewise :: ([Software], Encoding_Date) →
  Score_Timewise → Score_Timewise
update_MusicXML_Timewise = update_Score_Timewise


  -- |
read_MusicXMLDoc :: Document Posn → Result MusicXMLDoc
read_MusicXMLDoc doc =
  (read_DOCUMENT read_Score_Partwise doc ≫= return · Score · Partwise) `mplus`
  (read_DOCUMENT read_Score_Timewise doc ≫= return · Score · Timewise) `mplus`
  (read_DOCUMENT read_Opus doc ≫= return · Opus) `mplus`
  (read_DOCUMENT read_Container doc ≫= return · Container) `mplus`
  fail "<score-partwise> or <score-timewise> or <opus> or <container>"
  -- |
show_MusicXMLDoc :: MusicXMLDoc → Result (Document ())
show_MusicXMLDoc (Score (Partwise doc)) = show_MusicXML_Partwise doc
show_MusicXMLDoc (Score (Timewise doc)) = show_MusicXML_Timewise doc
show_MusicXMLDoc (Opus doc) = show_MusicXML_Opus doc
show_MusicXMLDoc (Container doc) = show_MusicXML_Container doc
  -- |
update_MusicXMLDoc :: ([Software], Encoding_Date) →
  MusicXMLDoc → MusicXMLDoc
update_MusicXMLDoc x (Score (Partwise doc)) =
  Score (Partwise (update_MusicXML_Partwise x doc))
update_MusicXMLDoc x (Score (Timewise doc)) =
  Score (Timewise (update_MusicXML_Timewise x doc))
update_MusicXMLDoc _ y = y
  -- |
read_MusicXMLRec :: FilePath → IO (Map.Map FilePath MusicXMLDoc)
read_MusicXMLRec f = do
  x ← read_FILE read_MusicXMLDoc f ≫= λa → return (f, a)
  case isOK (π2 x) of
    True → do
      xs ← mapM (λf' → read_FILE read_MusicXMLDoc f'
        ≫= λa → return (f', a))
        (Text.XML.MusicXML.getFiles (fromOK (π2 x)))
      return (Map.map fromOK (Map.filter isOK (Map.fromList (x : xs))))
    False → return (Map.empty)


  -- |
read_CONTENTS :: (Document Posn → Result a) →
  FilePath → Prelude.String → Result a
read_CONTENTS f filepath contents =
  [fail, f] (xmlParse' filepath contents)
  -- |
show_CONTENTS :: (a → Result (Document i)) → a → Prelude.String
show_CONTENTS f musicxml =
  maybe (fail "undefined error")
    (renderStyle (Style LeftMode 100 1.5) · document)
    ((toMaybe · f) musicxml)
```

```
  -- |
read_FILE :: (Document Posn → Result a) → FilePath → IO (Result a)
read_FILE f filepath = do
  exists ← doesFileExist filepath
  case exists of
    True → readFile filepath >>= return · (read_CONTENTS f) filepath
    False → (return · fail) ("no file: " ++ show filepath)
  -- |
show_FILE :: (a → Result (Document i)) → FilePath → a → IO ()
show_FILE f filepath musicxml =
  writeFile filepath (show_CONTENTS f musicxml)


  -- |
xmldecl :: XMLDecl
xmldecl = XMLDecl "1.0" Nothing Nothing
  -- |
getFiles :: MusicXMLDoc → [FilePath]
getFiles (Score _) = []
getFiles (Opus x) = Text.XML.MusicXML · Opus.getFiles x
getFiles (Container x) = Text.XML.MusicXML · Container.getFiles x
  -- |
toMaybe :: Result a → Maybe a
toMaybe (Ok x) = Just x
toMaybe (Error _) = Nothing
  -- | getTime uses old-time library. At future versions can be defined as:
  -- @getTime :: IO Prelude.String@
  -- @getTime = getCurrentTime ■= return . show . utctDay@
getTime :: IO Encoding_Date
getTime = getClockTime >>= toCalendarTime >>=
  return · (λ(CalendarTime yyyy mm dd _ _ _ _ _ _ _ _ _) →
    show4 yyyy ++ "-" ++ show2 (fromEnum mm + 1) ++ "-" ++ show2 dd)
  -- |
show2, show3, show4 :: Int → Prelude.String
show2 x | (x < 0) = show2 (−x)
        | otherwise = case show x of ; [a] → '0' : a : []; y → y
show3 x | (x < 0) = show3 (−x)
        | otherwise = case show2 x of ; [a, b] → '0' : a : b : []; y → y
show4 x | (x < 0) = show4 (−x)
        | otherwise = case show3 x of ; [a, b, c] → '0' : a : b : c : []; y → y
```

```
getTime :: IO Prelude.String
getTime = getCurrentTime >>= return . show . utctDay
```

2.10 Note

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Note where
import Text.XML.MusicXML.Common
import Prelude (Maybe (..), Monad (..), Show, Eq, (·), (++))
import Control.Monad (MonadPlus (..))
import qualified Data.Char (String)
import Text.XML.HaXml.Types (Content (..))
```

The common note elements between cue/grace notes and regular (full) notes: pitch, chord, and rest information, but not duration (cue and grace notes do not have duration encoded here). Unpitched elements are used for unpitched percussion, speaking voice, and other musical elements lacking determinate pitch.

```
  -- * Note entities
  -- |
type Full_Note = (Maybe Chord, Full_Note_)
  -- |
read_Full_Note :: STM Result [Content i] Full_Note
read_Full_Note = do
  y1 ← read_MAYBE read_Chord
  y2 ← read_Full_Note_
  return (y1, y2)
  -- |
show_Full_Note :: Full_Note → [Content ()]
show_Full_Note (a, b) =
  show_MAYBE show_Chord a ++
  show_Full_Note_ b
  -- |
data Full_Note_ = Full_Note_1 Pitch
  | Full_Note_2 Unpitched
  | Full_Note_3 Rest
  deriving (Eq, Show)
  -- |
read_Full_Note_ :: STM Result [Content i] Full_Note_
read_Full_Note_ =
  (read_Pitch >>= return · Full_Note_1) `mplus`
  (read_Unpitched >>= return · Full_Note_2) `mplus`
  (read_Rest >>= return · Full_Note_3)
  -- |
show_Full_Note_ :: Full_Note_ → [Content ()]
show_Full_Note_ (Full_Note_1 x) = show_Pitch x
show_Full_Note_ (Full_Note_2 x) = show_Unpitched x
show_Full_Note_ (Full_Note_3 x) = show_Rest x
```

Notes are the most common type of MusicXML data. The MusicXML format keeps the MuseData distinction between elements used for sound information and elements used for notation information (e.g., tie is used for sound, tied for notation). Thus grace notes do not have a duration element. Cue notes have a duration element, as do forward elements, but no tie elements. Having these two types of information available can make interchange considerably easier, as some programs handle one type of information much more readily than the other.

The position and printout entities for printing suggestions are defined in the common.mod file.

The dynamics and end-dynamics attributes correspond to MIDI 1.0's Note On and Note Off velocities, respectively. They are expressed in terms of percentages of the default forte value (90 for MIDI 1.0). The attack and release attributes are used to alter the staring and stopping time of the note from when it would otherwise occur based on the flow of durations - information that is specific to a performance. They are expressed in terms of divisions, either positive or negative. A note that starts a tie should not have a release attribute, and a note that stops a tie should not have an attack attribute. If a note is played only one time through a repeat, the time-only attribute shows which time to play the note. The pizzicato attribute is used when just this note is sounded pizzicato, vs. the pizzicato element which changes overall playback between pizzicato and arco.

```
  -- * Note
  -- |
type Note = ((Print_Style, Printout, Maybe CDATA, Maybe CDATA,
  Maybe CDATA, Maybe CDATA, Maybe CDATA, Maybe Yes_No),
  (Note_, Maybe Instrument, Editorial_Voice, Maybe Type, [Dot],
  Maybe Accidental, Maybe Time_Modification, Maybe Stem, Maybe Notehead,
  Maybe Staff, [Beam], [Notations], [Lyric]))
```

```
  -- |
read_Note :: Eq i ⇒ STM Result [Content i] Note
read_Note = do
  y ← read_ELEMENT "note"
  y1 ← read_8 read_Print_Style read_Printout
    (read_IMPLIED "dynamics" read_CDATA)
    (read_IMPLIED "end-dynamics" read_CDATA)
    (read_IMPLIED "attack" read_CDATA)
    (read_IMPLIED "release" read_CDATA)
    (read_IMPLIED "time-only" read_CDATA)
    (read_IMPLIED "pizzicato" read_Yes_No) (attributes y)
  y2 ← read_13 read_Note_ (read_MAYBE read_Instrument)
    read_Editorial_Voice (read_MAYBE read_Type)
    (read_LIST read_Dot) (read_MAYBE read_Accidental)
    (read_MAYBE read_Time_Modification)
    (read_MAYBE read_Stem) (read_MAYBE read_Notehead)
    (read_MAYBE read_Staff) (read_LIST read_Beam)
    (read_LIST read_Notations) (read_LIST read_Lyric)
    (childs y)
  return (y1, y2)
show_Note :: Note → [Content ()]
show_Note ((a, b, c, d, e, f, g, h), (i, j, k, l, m, n, o, p, q, r, s, t, u)) =
  show_ELEMENT "note" (show_Print_Style a ++ show_Printout b ++
      show_IMPLIED "dynamics" show_CDATA c ++
      show_IMPLIED "end-dynamics" show_CDATA d ++
      show_IMPLIED "attack" show_CDATA e ++
      show_IMPLIED "release" show_CDATA f ++
      show_IMPLIED "time-only" show_CDATA g ++
      show_IMPLIED "pizzicato" show_Yes_No h)
    (show_Note_ i ++ show_MAYBE show_Instrument j ++
      show_Editorial_Voice k ++
      show_MAYBE show_Type l ++
      show_LIST show_Dot m ++
      show_MAYBE show_Accidental n ++
      show_MAYBE show_Time_Modification o ++
      show_MAYBE show_Stem p ++
      show_MAYBE show_Notehead q ++
      show_MAYBE show_Staff r ++
      show_LIST show_Beam s ++
      show_LIST show_Notations t ++
      show_LIST show_Lyric u)
  -- ** Note_
  -- |
data Note_ = Note_1 (Grace, Full_Note, Maybe (Tie, Maybe Tie))
  | Note_2 (Cue, Full_Note, Duration)
  | Note_3 (Full_Note, Duration, Maybe (Tie, Maybe Tie))
  deriving (Eq, Show)
  -- |
read_Note_ :: STM Result [Content i] Note_
read_Note_ =
  (read_Note_aux1 >>= return · Note_1) `mplus`
  (read_Note_aux2 >>= return · Note_2) `mplus`
  (read_Note_aux3 >>= return · Note_3)
read_Note_aux1 ::
  STM Result [Content i] (Grace, Full_Note, Maybe (Tie, Maybe Tie))
read_Note_aux1 = do
  y1 ← read_Grace
  y2 ← read_Full_Note
```

```
    y3 ← read_MAYBE read_Note_aux4
    return (y1, y2, y3)
read_Note_aux2 :: STM Result [Content i] (Cue, Full_Note, Duration)
read_Note_aux2 = do
    y1 ← read_Cue
    y2 ← read_Full_Note
    y3 ← read_Duration
    return (y1, y2, y3)
read_Note_aux3 ::
    STM Result [Content i] (Full_Note, Duration, Maybe (Tie, Maybe Tie))
read_Note_aux3 = do
    y1 ← read_Full_Note
    y2 ← read_Duration
    y3 ← read_MAYBE read_Note_aux4
    return (y1, y2, y3)
read_Note_aux4 :: STM Result [Content i] (Tie, Maybe Tie)
read_Note_aux4 = do
    y1 ← read_Tie
    y2 ← read_MAYBE read_Tie
    return (y1, y2)
    -- |
show_Note_ :: Note_ → [Content ()]
show_Note_ (Note_1 (a, b, c)) =
    show_Grace a ++ show_Full_Note b ++ show_MAYBE show_Note_aux1 c
show_Note_ (Note_2 (a, b, c)) =
    show_Cue a ++ show_Full_Note b ++ show_Duration c
show_Note_ (Note_3 (a, b, c)) =
    show_Full_Note a ++ show_Duration b ++ show_MAYBE show_Note_aux1 c
    -- |
show_Note_aux1 :: (Tie, Maybe Tie) → [Content ()]
show_Note_aux1 (a, b) = show_Tie a ++ show_MAYBE show_Tie b
```

Pitch is represented as a combination of the step of the diatonic scale, the chromatic alteration, and the octave. The step element uses the English letters A through G. The alter element represents chromatic alteration in number of semitones (e.g., -1 for flat, 1 for sharp). Decimal values like 0.5 (quarter tone sharp) may be used for microtones. The octave element is represented by the numbers 0 to 9, where 4 indicates the octave started by middle C.

```
    -- |
type Pitch = (Step, Maybe Alter, Octave)
    -- |
read_Pitch :: STM Result [Content i] Pitch
read_Pitch = do
    y ← read_ELEMENT "pitch"
    read_3 read_Step (read_MAYBE read_Alter) read_Octave (childs y)
    -- |
show_Pitch :: Pitch → [Content ()]
show_Pitch (a, b, c) =
    show_ELEMENT "pitch" []
      (show_Step a ++ show_MAYBE show_Alter b ++ show_Octave c)
    -- |
type Step = PCDATA
    -- |
read_Step :: STM Result [Content i] Step
read_Step = do
    y ← read_ELEMENT "step"
    read_1 read_PCDATA (childs y)
    -- |
show_Step :: Step → [Content ()]
```

```
show_Step x = show_ELEMENT "step" [] (show_PCDATA x)
  -- |
type Alter = PCDATA
  -- |
read_Alter :: STM Result [Content i] Alter
read_Alter = do
  y ← read_ELEMENT "alter"
  read_1 read_PCDATA (childs y)
  -- |
show_Alter :: Alter → [Content ()]
show_Alter x = show_ELEMENT "alter" [] (show_PCDATA x)
  -- |
type Octave = PCDATA
  -- |
read_Octave :: STM Result [Content i] Octave
read_Octave = do
  y ← read_ELEMENT "octave"
  read_1 read_PCDATA (childs y)
  -- |
show_Octave :: Octave → [Content ()]
show_Octave x = show_ELEMENT "octave" [] (show_PCDATA x)
```

The cue and grace elements indicate the presence of cue and grace notes. The slash attribute for a grace note is yes for slashed eighth notes. The other grace note attributes come from MuseData sound suggestions. Steal-time-previous indicates the percentage of time to steal from the previous note for the grace note. Steal-time-following indicates the percentage of time to steal from the following note for the grace note. Make-time indicates to make time, not steal time; the units are in real-time divisions for the grace note.

```
  -- |
type Cue = ()
  -- |
read_Cue :: STM Result [Content i] Cue
read_Cue = read_ELEMENT "cue" ≫ return ()
  -- |
show_Cue :: Cue → [Content ()]
show_Cue _ = show_ELEMENT "cue" [] []
  -- |
type Grace = ((Maybe CDATA, Maybe CDATA, Maybe CDATA, Maybe Yes_No), ())
  -- |
read_Grace :: STM Result [Content i] Grace
read_Grace = do
  y ← read_ELEMENT "grace"
  y1 ← read_4 (read_IMPLIED "steal-time-previous" read_CDATA)
    (read_IMPLIED "steal-time-following" read_CDATA)
    (read_IMPLIED "make-time" read_CDATA)
    (read_IMPLIED "slash" read_Yes_No) (attributes y)
  return (y1, ())
  -- |
show_Grace :: Grace → [Content ()]
show_Grace ((a, b, c, d), _) =
  show_ELEMENT "grace"
    (show_IMPLIED "steal-time-previous" show_CDATA a ++
      show_IMPLIED "steal-time-following" show_CDATA b ++
      show_IMPLIED "make-time" show_CDATA c ++
      show_IMPLIED "slash" show_Yes_No d)
    []
```

The chord element indicates that this note is an additional chord tone with the preceding note. The duration of this note can be no longer than the preceding note. In MuseData, a missing duration indicates

the same length as the previous note, but the MusicXML format requires a duration for chord notes too.

```
  -- |
type Chord = ()
  -- |
read_Chord :: STM Result [Content i] Chord
read_Chord =
  read_ELEMENT "chord" ≫ return ()
  -- |
show_Chord :: Chord → [Content ()]
show_Chord _ = show_ELEMENT "chord" [] []
```

The unpitched element indicates musical elements that are notated on the staff but lack definite pitch, such as unpitched percussion and speaking voice. Like notes, it uses step and octave elements to indicate placement on the staff, following the current clef. If percussion clef is used, the display-step and display-octave elements are interpreted as if in treble clef, with a G in octave 4 on line 2. If not present, the note is placed on the middle line of the staff, generally used for one-line staffs.

```
type Unpitched = Maybe (Display_Step, Display_Octave)
  -- |
read_Unpitched :: STM Result [Content i] Unpitched
read_Unpitched = do
  y ← read_ELEMENT "unpitched"
  read_1 (read_MAYBE read_Unpitched_aux1) (childs y)
read_Unpitched_aux1 :: STM Result [Content i] (Display_Step, Display_Octave)
read_Unpitched_aux1 = do
  y1 ← read_Display_Step
  y2 ← read_Display_Octave
  return (y1, y2)
  -- |
show_Unpitched :: Unpitched → [Content ()]
show_Unpitched x =
  show_ELEMENT "unpitched" []
    (show_MAYBE (λ(a, b) → show_Display_Step a ++
      show_Display_Octave b) x)
  -- |
type Display_Step = PCDATA
  -- |
read_Display_Step :: STM Result [Content i] Display_Step
read_Display_Step = do
  y ← read_ELEMENT "display-step"
  read_1 read_PCDATA (childs y)
  -- |
show_Display_Step :: Display_Step → [Content ()]
show_Display_Step x = show_ELEMENT "display-step" [] (show_PCDATA x)
  -- |
type Display_Octave = PCDATA
  -- |
read_Display_Octave :: STM Result [Content i] Display_Octave
read_Display_Octave = do
  y ← read_ELEMENT "display-octave"
  read_1 read_PCDATA (childs y)
  -- |
show_Display_Octave :: Display_Octave → [Content ()]
show_Display_Octave x = show_ELEMENT "display-octave" [] (show_PCDATA x)
```

The rest element indicates notated rests or silences. Rest are usually empty, but placement on the staff can be specified using display-step and display-octave elements.

```
  -- |
type Rest = Maybe (Display_Step, Display_Octave)
```

```
  -- |
read_Rest :: STM Result [Content i] Rest
read_Rest = do
  y ← read_ELEMENT "rest"
  read_1 (read_MAYBE read_Rest_aux1) (childs y)
  -- |
read_Rest_aux1 :: STM Result [Content i] (Display_Step, Display_Octave)
read_Rest_aux1 = do
  y1 ← read_Display_Step
  y2 ← read_Display_Octave
  return (y1, y2)
  -- |
show_Rest :: Rest → [Content ()]
show_Rest x =
  show_ELEMENT "rest" []
    (show_MAYBE (λ(a, b) → show_Display_Step a ++
      show_Display_Octave b) x)
```

Duration is a positive number specified in division units. This is the intended duration vs. notated duration (for instance, swing eighths vs. even eighths, or differences in dotted notes in Baroque-era music). Differences in duration specific to an interpretation or performance should use the note element's attack and release attributes.

The tie element indicates that a tie begins or ends with this note. The tie element indicates sound; the tied element indicates notation.

```
type Duration = PCDATA
  -- |
read_Duration :: STM Result [Content i] Duration
read_Duration = do
  y ← read_ELEMENT "duration"
  read_1 read_PCDATA (childs y)
  -- |
show_Duration :: Duration → [Content ()]
show_Duration x = show_ELEMENT "duration" [] (show_PCDATA x)
  -- |
type Tie = (Start_Stop, ())
  -- |
read_Tie :: STM Result [Content i] Tie
read_Tie = do
  y ← read_ELEMENT "tie"
  y1 ← read_1 (read_REQUIRED "type" read_Start_Stop) (attributes y)
  return (y1, ())
  -- |
show_Tie :: Tie → [Content ()]
show_Tie (a, _) =
  show_ELEMENT "tie" (show_REQUIRED "type" show_Start_Stop a) []
```

If multiple score-instruments are specified on a score-part, there should be an instrument element for each note in the part. The id attribute is an IDREF back to the score-instrument ID.

```
  -- ** Instrument
  -- |
type Instrument = (ID, ())
  -- |
read_Instrument :: STM Result [Content i] Instrument
read_Instrument = do
  y ← read_ELEMENT "instrument"
  y1 ← read_1 (read_REQUIRED "id" read_ID) (attributes y)
  return (y1, ())
  -- |
```

```
show_Instrument :: Instrument → [Content ()]
show_Instrument (a, _) =
  show_ELEMENT "instrument" (show_REQUIRED "id" show_ID a) []
```

Type indicates the graphic note type, Valid values (from shortest to longest) are 256th, 128th, 64th, 32nd, 16th, eighth, quarter, half, whole, breve, and long. The size attribute indicates full, cue, or large size, with full the default for regular notes and cue the default for cue and grace notes.

```
  -- ** Type
  -- |
type Type = (Maybe Symbol_Size, PCDATA)
  -- |
read_Type :: STM Result [Content i] Type
read_Type = do
  y ← read_ELEMENT "type"
  y1 ← read_1 (read_IMPLIED "size" read_Symbol_Size) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Type :: Type → [Content ()]
show_Type (a, b) =
  show_ELEMENT "type"
    (show_IMPLIED "size" show_Symbol_Size a)
    (show_PCDATA b)
```

One dot element is used for each dot of prolongation. The placement element is used to specify whether the dot should appear above or below the staff line. It is ignored for notes that appear on a staff space.

```
  -- ** Dot
  -- |
type Dot = ((Print_Style, Placement), ())
  -- |
read_Dot :: STM Result [Content i] Dot
read_Dot = do
  y ← read_ELEMENT "dot"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Dot :: Dot → [Content ()]
show_Dot ((a, b), _) =
  show_ELEMENT "dot"
    (show_Print_Style a ++
      show_Placement b)
    []
```

Actual notated accidentals. Valid values include: sharp, natural, flat, double-sharp, sharp-sharp, flat-flat, natural-sharp, natural-flat, quarter-flat, quarter-sharp, three-quarters-flat, and three-quarters-sharp. Editorial and cautionary indications are indicated by attributes. Values for these attributes are "no" if not present. Specific graphic display such as parentheses, brackets, and size are controlled by the level-display entity defined in the common.mod file.

```
  -- ** Accidental
  -- |
type Accidental = ((Maybe Yes_No, Maybe Yes_No, Level_Display, Print_Style),
  PCDATA)
  -- |
read_Accidental :: STM Result [Content i] Accidental
read_Accidental = do
  y ← read_ELEMENT "accidental"
  y1 ← read_4 (read_IMPLIED "cautionary" read_Yes_No)
```

```
      (read_IMPLIED "editorial" read_Yes_No)
      read_Level_Display read_Print_Style (attributes y)
    y2 ← read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Accidental :: Accidental → [Content ()]
show_Accidental ((a, b, c, d), e) =
    show_ELEMENT "accidental"
      (show_IMPLIED "cautionary" show_Yes_No a ++
        show_IMPLIED "editorial" show_Yes_No b ++
        show_Level_Display c ++
        show_Print_Style d)
      (show_PCDATA e)
```

Time modification indicates tuplets and other durational changes. The child elements are defined in the common.mod file.

```
    -- ** Time_Modification
    -- |
type Time_Modification = (Actual_Notes, Normal_Notes,
    Maybe (Normal_Type, [Normal_Dot]))
    -- |
read_Time_Modification :: Eq i ⇒ STM Result [Content i] Time_Modification
read_Time_Modification = do
    y ← read_ELEMENT "time-modification"
    read_3 read_Actual_Notes read_Normal_Notes
      (read_MAYBE (read_Time_Modification_aux1)) (childs y)
    -- |
read_Time_Modification_aux1 :: Eq i ⇒
    STM Result [Content i] (Normal_Type, [Normal_Dot])
read_Time_Modification_aux1 = do
    y1 ← read_Normal_Type
    y2 ← read_LIST read_Normal_Dot
    return (y1, y2)
    -- |
show_Time_Modification :: Time_Modification → [Content ()]
show_Time_Modification (a, b, c) =
    show_ELEMENT "time-modification" []
      (show_Actual_Notes a ++ show_Normal_Notes b ++
        show_MAYBE (λ(c1, c2) → show_Normal_Type c1 ++
          show_LIST show_Normal_Dot c2) c)
```

Stems can be down, up, none, or double. For down and up stems, the position attributes can be used to specify stem length. The relative values specify the end of the stem relative to the program default. Default values specify an absolute end stem position. Negative values of relative-y that would flip a stem instead of shortening it are ignored.

```
    -- ** Stem
    -- |
type Stem = ((Position, Color), PCDATA)
    -- |
read_Stem :: STM Result [Content i] Stem
read_Stem = do
    y ← read_ELEMENT "stem"
    y1 ← read_2 read_Position read_Color (attributes y)
    y2 ← read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Stem :: Stem → [Content ()]
show_Stem ((a, b), c) =
```

```
  show_ELEMENT "stem"
    (show_Position a ++ show_Color b)
    (show_PCDATA c)
```

The notehead element indicates shapes other than the open and closed ovals associated with note durations. The element value can be slash, triangle, diamond, square, cross, x, circle-x, inverted triangle, arrow down, arrow up, slashed, back slashed, normal, cluster, or none. For shape note music, the element values do, re, mi, fa, so, la, and ti are used, corresponding to Aikin's 7-shape system.

The arrow shapes differ from triangle and inverted triangle by being centered on the stem. Slashed and back slashed notes include both the normal notehead and a slash. The triangle shape has the tip of the triangle pointing up; the inverted triangle shape has the tip of the triangle pointing down.

For the enclosed shapes, the default is to be hollow for half notes and longer, and filled otherwise. The filled attribute can be set to change this if needed.

If the parentheses attribute is set to yes, the notehead is parenthesized. It is no by default.

```
  -- ** Notehead
  -- |
type Notehead = ((Maybe Yes_No, Maybe Yes_No, Font, Color), PCDATA)
  -- |
read_Notehead :: STM Result [Content i] Notehead
read_Notehead = do
  y ← read_ELEMENT "notehead"
  y1 ← read_4 (read_IMPLIED "filled" read_Yes_No)
    (read_IMPLIED "parentheses" read_Yes_No)
    read_Font read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Notehead :: Notehead → [Content ()]
show_Notehead ((a, b, c, d), e) =
  show_ELEMENT "notehead"
    (show_IMPLIED "filled" show_Yes_No a ++
      show_IMPLIED "parentheses" show_Yes_No b ++
      show_Font c ++ show_Color d)
    (show_PCDATA e)
```

Beam types include begin, continue, end, forward hook, and backward hook. In MuseData, up to six concurrent beams are available to cover up to 256th notes. This seems sufficient so we use an enumerated type defined in the common.mod file. The repeater attribute, used for tremolos, needs to be specified with a "yes" value for each beam using it. Beams that have a begin value can also have a fan attribute to indicate accelerandos and ritardandos using fanned beams. The fan attribute may also be used with a continue value if the fanning direction changes on that note. The value is "none" if not specified.

Note that the beam number does not distinguish sets of beams that overlap, as it does for slur and other elements. Beaming groups are distinguished by being in different voices and/or the presence or absence of grace and cue elements.

```
  -- ** Beam
  -- |
type Beam = ((Beam_Level, Maybe Yes_No, Maybe Beam_, Color), PCDATA)
  -- |
read_Beam :: STM Result [Content i] Beam
read_Beam = do
  y ← read_ELEMENT "beam"
  y1 ← read_4 (read_DEFAULT "number" read_Beam_Level Beam_Level_1)
    (read_IMPLIED "repeater" read_Yes_No)
    (read_IMPLIED "fan" read_Beam_)
    read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
```

```
show_Beam :: Beam → [Content ()]
show_Beam ((a, b, c, d), e) =
  show_ELEMENT "beam"
    (show_IMPLIED "number" show_Beam_Level (Just a) ++
      show_IMPLIED "repeater" show_Yes_No b ++
      show_IMPLIED "fan" show_Beam_ c ++
      show_Color d)
    (show_PCDATA e)
  -- |
data Beam_ = Beam_Accel | Beam_Rit | Beam_None
  deriving (Eq, Show)
  -- |
read_Beam_ :: Data.Char.String → Result Beam_
read_Beam_ "accel" = return Beam_Accel
read_Beam_ "rit"   = return Beam_Rit
read_Beam_ "none" = return Beam_None
read_Beam_ _       =
  fail "I expect fan attribute"
  -- |
show_Beam_ :: Beam_ → Data.Char.String
show_Beam_ Beam_Accel = "accel"
show_Beam_ Beam_Rit   = "rit"
show_Beam_ Beam_None = "none"
```

Notations are musical notations, not XML notations. Multiple notations are allowed in order to represent multiple editorial levels. The set of notations will be refined and expanded over time, especially to handle more instrument-specific technical notations.

```
  -- ** Notations
  -- |
type Notations = [(Editorial, Notations_)]
  -- |
read_Notations :: Eq i ⇒ STM Result [Content i] Notations
read_Notations = do
  y ← read_ELEMENT "notations"
  read_1 (read_LIST read_Notations_aux1) (childs y)
  -- |
show_Notations :: Notations → [Content ()]
show_Notations a =
  show_ELEMENT "notations" [] (show_LIST show_Notations_aux1 a)
  -- |
read_Notations_aux1 :: Eq i ⇒ STM Result [Content i] (Editorial, Notations_)
read_Notations_aux1 = do
  y1 ← read_Editorial
  y2 ← read_Notations_
  return (y1, y2)
  -- |
show_Notations_aux1 :: (Editorial, Notations_) → [Content ()]
show_Notations_aux1 (a, b) = show_Editorial a ++ show_Notations_ b
  -- |
data Notations_ = Notations_1 Tied
  | Notations_2 Slur
  | Notations_3 Tuplet
  | Notations_4 Glissando
  | Notations_5 Slide
  | Notations_6 Ornaments
  | Notations_7 Technical
  | Notations_8 Articulations
  | Notations_9 Dynamics
```

```
    | Notations_10 Fermata
    | Notations_11 Arpeggiate
    | Notations_12 Non_Arpeggiate
    | Notations_13 Accidental_Mark
    | Notations_14 Other_Notation
  deriving (Eq, Show)
  -- |
read_Notations_ :: Eq i ⇒ STM Result [Content i] Notations_
read_Notations_ =
  (read_Tied ≫= return · Notations_1) `mplus`
  (read_Slur ≫= return · Notations_2) `mplus`
  (read_Tuplet ≫= return · Notations_3) `mplus`
  (read_Glissando ≫= return · Notations_4) `mplus`
  (read_Slide ≫= return · Notations_5) `mplus`
  (read_Ornaments ≫= return · Notations_6) `mplus`
  (read_Technical ≫= return · Notations_7) `mplus`
  (read_Articulations ≫= return · Notations_8) `mplus`
  (read_Dynamics ≫= return · Notations_9) `mplus`
  (read_Fermata ≫= return · Notations_10) `mplus`
  (read_Arpeggiate ≫= return · Notations_11) `mplus`
  (read_Non_Arpeggiate ≫= return · Notations_12) `mplus`
  (read_Accidental_Mark ≫= return · Notations_13) `mplus`
  (read_Other_Notation ≫= return · Notations_14)
  -- |
show_Notations_ :: Notations_ → [Content ()]
show_Notations_ (Notations_1 x) = show_Tied x
show_Notations_ (Notations_2 x) = show_Slur x
show_Notations_ (Notations_3 x) = show_Tuplet x
show_Notations_ (Notations_4 x) = show_Glissando x
show_Notations_ (Notations_5 x) = show_Slide x
show_Notations_ (Notations_6 x) = show_Ornaments x
show_Notations_ (Notations_7 x) = show_Technical x
show_Notations_ (Notations_8 x) = show_Articulations x
show_Notations_ (Notations_9 x) = show_Dynamics x
show_Notations_ (Notations_10 x) = show_Fermata x
show_Notations_ (Notations_11 x) = show_Arpeggiate x
show_Notations_ (Notations_12 x) = show_Non_Arpeggiate x
show_Notations_ (Notations_13 x) = show_Accidental_Mark x
show_Notations_ (Notations_14 x) = show_Other_Notation x
  -- *** Tied
  -- |
type Tied = ((Start_Stop, Maybe Number_Level, Line_Type, Position, Placement,
    Orientation, Bezier, Color), ())
  -- |
read_Tied :: STM Result [Content i] Tied
read_Tied = do
  y ← read_ELEMENT "tied"
  y1 ← read_8 (read_REQUIRED "type" read_Start_Stop)
    (read_IMPLIED "number" read_Number_Level)
    read_Line_Type read_Position read_Placement
    read_Orientation read_Bezier read_Color (attributes y)
  return (y1, ())
  -- |
show_Tied :: Tied → [Content ()]
show_Tied ((a, b, c, d, e, f, g, h), _) =
  show_ELEMENT "tied"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_IMPLIED "number" show_Number_Level b ++
```

```
        show_Line_Type c ++
        show_Position d ++
        show_Placement e ++
        show_Orientation f ++
        show_Bezier g ++
        show_Color h) []
```

Slur elements are empty. Most slurs are represented with two elements: one with a start type, and one with a stop type. Slurs can add more elements using a continue type. This is typically used to specify the formatting of cross- system slurs, or to specify the shape of very complex slurs.

```
-- *** Slur
-- |
type Slur = ((Start_Stop_Continue, Number_Level, Line_Type, Position, Placement,
    Orientation, Bezier, Color), ())
-- |
read_Slur :: STM Result [Content i] Slur
read_Slur = do
  y ← read_ELEMENT "slur"
  y1 ← read_8 (read_REQUIRED "type" read_Start_Stop_Continue)
    (read_DEFAULT "number" read_Number_Level Number_Level_1)
    read_Line_Type read_Position read_Placement
    read_Orientation read_Bezier read_Color (attributes y)
  return (y1, ())
-- |
show_Slur :: Slur → [Content ()]
show_Slur ((a, b, c, d, e, f, g, h), _) =
  show_ELEMENT "slur"
    (show_REQUIRED "type" show_Start_Stop_Continue a ++
      show_IMPLIED "number" show_Number_Level (Just b) ++
      show_Line_Type c ++
      show_Position d ++
      show_Placement e ++
      show_Orientation f ++
      show_Bezier g ++
      show_Color h) []
```

A tuplet element is present when a tuplet is to be displayed graphically, in addition to the sound data provided by the time-modification elements. The number attribute is used to distinguish nested tuplets. The bracket attribute is used to indicate the presence of a bracket. If unspecified, the results are implementation-dependent. The line-shape attribute is used to specify whether the bracket is straight or in the older curved or slurred style. It is straight by default.

Whereas a time-modification element shows how the cumulative, sounding effect of tuplets compare to the written note type, the tuplet element describes how this is displayed. The tuplet-actual and tuplet-normal elements provide optional full control over tuplet specifications. Each allows the number and note type (including dots) describing a single tuplet. If any of these elements are absent, their values are based on the time-modification element.

The show-number attribute is used to display either the number of actual notes, the number of both actual and normal notes, or neither. It is actual by default. The show-type attribute is used to display either the actual type, both the actual and normal types, or neither. It is none by default.

```
-- *** Tuplet
-- |
type Tuplet = ((Start_Stop, Maybe Number_Level, Maybe Yes_No, Maybe Tuplet_,
    Maybe Tuplet_, Line_Shape, Position, Placement),
  (Maybe Tuplet_Actual, Maybe Tuplet_Normal))
-- |
read_Tuplet :: Eq i ⇒ STM Result [Content i] Tuplet
read_Tuplet = do
  y ← read_ELEMENT "tuplet"
```

```
  y1 ← read_8 (read_REQUIRED "type" read_Start_Stop)
    (read_IMPLIED "number" read_Number_Level)
    (read_IMPLIED "bracket" read_Yes_No)
    (read_IMPLIED "show-number" read_Tuplet_)
    (read_IMPLIED "show-type" read_Tuplet_)
    read_Line_Shape read_Position read_Placement
    (attributes y)
  y2 ← read_2 (read_MAYBE read_Tuplet_Actual)
    (read_MAYBE read_Tuplet_Normal)
    (childs y)
  return (y1, y2)
  -- |
show_Tuplet :: Tuplet → [Content ()]
show_Tuplet ((a, b, c, d, e, f, g, h), (i, j)) =
  show_ELEMENT "tuplet"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_IMPLIED "number" show_Number_Level b ++
      show_IMPLIED "bracket" show_Yes_No c ++
      show_IMPLIED "show-number" show_Tuplet_ d ++
      show_IMPLIED "show-type" show_Tuplet_ e ++
      show_Line_Shape f ++ show_Position g ++
      show_Placement h)
    (show_MAYBE show_Tuplet_Actual i ++
      show_MAYBE show_Tuplet_Normal j)
  -- |
data Tuplet_ = Tuplet_1 | Tuplet_2 | Tuplet_3
  deriving (Eq, Show)
  -- |
read_Tuplet_ :: Data.Char.String → Result Tuplet_
read_Tuplet_ "actual" = return Tuplet_1
read_Tuplet_ "both" = return Tuplet_2
read_Tuplet_ "none" = return Tuplet_3
read_Tuplet_ _ = fail "wrong value at tuplet"
  -- |
show_Tuplet_ :: Tuplet_ → Data.Char.String
show_Tuplet_ Tuplet_1 = "actual"
show_Tuplet_ Tuplet_2 = "both"
show_Tuplet_ Tuplet_3 = "none"
  -- |
type Tuplet_Actual = (Maybe Tuplet_Number, Maybe Tuplet_Type, [Tuplet_Dot])
  -- |
read_Tuplet_Actual :: Eq i ⇒ STM Result [Content i] Tuplet_Actual
read_Tuplet_Actual = do
  y ← read_ELEMENT "tuplet-actual"
  read_3 (read_MAYBE read_Tuplet_Number)
    (read_MAYBE read_Tuplet_Type)
    (read_LIST read_Tuplet_Dot) (childs y)
  -- |
show_Tuplet_Actual :: Tuplet_Actual → [Content ()]
show_Tuplet_Actual (a, b, c) =
  show_ELEMENT "tuplet-actual" []
    (show_MAYBE show_Tuplet_Number a ++
      show_MAYBE show_Tuplet_Type b ++
      show_LIST show_Tuplet_Dot c)
  -- |
type Tuplet_Normal = (Maybe Tuplet_Number, Maybe Tuplet_Type, [Tuplet_Dot])
  -- |
read_Tuplet_Normal :: Eq i ⇒ STM Result [Content i] Tuplet_Normal
```

```
read_Tuplet_Normal = do
  y ← read_ELEMENT "tuplet-normal"
  read_3 (read_MAYBE read_Tuplet_Number)
    (read_MAYBE read_Tuplet_Type)
    (read_LIST read_Tuplet_Dot) (childs y)
  -- |
show_Tuplet_Normal :: Tuplet_Normal → [Content ()]
show_Tuplet_Normal (a, b, c) =
  show_ELEMENT "tuplet-normal" []
    (show_MAYBE show_Tuplet_Number a ++
      show_MAYBE show_Tuplet_Type b ++
      show_LIST show_Tuplet_Dot c)
  -- |
type Tuplet_Number = ((Font, Color), PCDATA)
  -- |
read_Tuplet_Number :: STM Result [Content i] Tuplet_Number
read_Tuplet_Number = do
  y ← read_ELEMENT "tuplet-number"
  y1 ← read_2 read_Font read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Tuplet_Number :: Tuplet_Number → [Content ()]
show_Tuplet_Number ((a, b), c) =
  show_ELEMENT "tuplet-number"
    (show_Font a ++ show_Color b)
    (show_PCDATA c)
  -- |
type Tuplet_Type = ((Font, Color), PCDATA)
  -- |
read_Tuplet_Type :: STM Result [Content i] Tuplet_Type
read_Tuplet_Type = do
  y ← read_ELEMENT "tuplet-type"
  y1 ← read_2 read_Font read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Tuplet_Type :: Tuplet_Type → [Content ()]
show_Tuplet_Type ((a, b), c) =
  show_ELEMENT "tuplet-type"
    (show_Font a ++ show_Color b)
    (show_PCDATA c)
  -- |
type Tuplet_Dot = ((Font, Color), ())
  -- |
read_Tuplet_Dot :: STM Result [Content i] Tuplet_Dot
read_Tuplet_Dot = do
  y ← read_ELEMENT "tuplet-dot"
  y1 ← read_2 read_Font read_Color (attributes y)
  return (y1, ())
  -- |
show_Tuplet_Dot :: Tuplet_Dot → [Content ()]
show_Tuplet_Dot ((a, b), _) =
  show_ELEMENT "tuplet-dot"
    (show_Font a ++ show_Color b) []
```

Glissando and slide elements both indicate rapidly moving from one pitch to the other so that individual notes are not discerned. The distinction is similar to that between NIFF's glissando and portamento elements. A glissando sounds the half notes in between the slide and defaults to a wavy line. A slide is

continuous between two notes and defaults to a solid line. The optional text for a glissando or slide is printed alongside the line.

```
  -- *** Glissando
  -- |
type Glissando = ((Start_Stop, Number_Level, Line_Type, Print_Style), Text)
  -- |
read_Glissando :: STM Result [Content i] Glissando
read_Glissando = do
  y ← read_ELEMENT "glissando"
  y1 ← read_4 (read_REQUIRED "type" read_Start_Stop)
    (read_DEFAULT "number" read_Number_Level Number_Level_1)
    read_Line_Type read_Print_Style (attributes y)
  y2 ← read_1 read_Text (childs y)
  return (y1, y2)
  -- |
show_Glissando :: Glissando → [Content ()]
show_Glissando ((a, b, c, d), e) =
  show_ELEMENT "glissando"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_IMPLIED "number" show_Number_Level (Just b) ++
      show_Line_Type c ++ show_Print_Style d)
    (show_Text e)
  -- *** Slide
  -- |
type Slide = ((Start_Stop, Number_Level, Line_Type, Print_Style, Bend_Sound), Text)
  -- |
read_Slide :: STM Result [Content i] Slide
read_Slide = do
  y ← read_ELEMENT "slide"
  y1 ← read_5 (read_REQUIRED "type" read_Start_Stop)
    (read_DEFAULT "number" read_Number_Level Number_Level_1)
    read_Line_Type read_Print_Style read_Bend_Sound
    (attributes y)
  y2 ← read_1 read_Text (childs y)
  return (y1, y2)
  -- |
show_Slide :: Slide → [Content ()]
show_Slide ((a, b, c, d, e), f) =
  show_ELEMENT "slide"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_IMPLIED "number" show_Number_Level (Just b) ++
      show_Line_Type c ++ show_Print_Style d ++
      show_Bend_Sound e)
    (show_Text f)
```

The other-notation element is used to define any notations not yet in the MusicXML format. This allows extended representation, though without application interoperability. It handles notations where more specific extension elements such as other-dynamics and other-technical are not appropriate.

```
  -- |
type Other_Notation = ((Start_Stop_Single, Number_Level, Print_Object,
    Print_Style, Placement), PCDATA)
  -- |
read_Other_Notation :: STM Result [Content i] Other_Notation
read_Other_Notation = do
  y ← read_ELEMENT "other-notation"
  y1 ← read_5 (read_REQUIRED "type" read_Start_Stop_Single)
    (read_DEFAULT "number" read_Number_Level Number_Level_1)
    read_Print_Object read_Print_Style read_Placement
```

```
    (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Other_Notation :: Other_Notation → [Content ()]
show_Other_Notation ((a, b, c, d, e), f) =
  show_ELEMENT "other-notation"
    (show_REQUIRED "type" show_Start_Stop_Single a ++
      show_IMPLIED "number" show_Number_Level (Just b) ++
      show_Print_Object c ++ show_Print_Style d ++
      show_Placement e)
    (show_PCDATA f)
```

Ornaments can be any of several types, followed optionally by accidentals. The accidental-mark element's content is represented the same as an accidental element, but with a different name to reflect the different musical meaning.

```
  -- *** Ornaments
  -- |
type Ornaments = [(Ornaments_, [Accidental_Mark])]
  -- |
read_Ornaments :: Eq i ⇒ STM Result [Content i] Ornaments
read_Ornaments = do
  y ← read_ELEMENT "ornaments"
  read_1 (read_LIST read_Ornaments_aux1) (childs y)
read_Ornaments_aux1 :: Eq i ⇒ STM Result [Content i] (Ornaments_, [Accidental_Mark])
read_Ornaments_aux1 = do
  y1 ← read_Ornaments_
  y2 ← read_LIST read_Accidental_Mark
  return (y1, y2)
  -- |
show_Ornaments :: Ornaments → [Content ()]
show_Ornaments l =
  show_ELEMENT "ornaments" [] (show_LIST show_Ornaments_aux1 l)
show_Ornaments_aux1 :: (Ornaments_, [Accidental_Mark]) → [Content ()]
show_Ornaments_aux1 (a, b) =
  show_Ornaments_ a ++ show_LIST show_Accidental_Mark b
  -- |
data Ornaments_ = Ornaments_1 Trill_Mark
  | Ornaments_2 Turn
  | Ornaments_3 Delayed_Turn
  | Ornaments_4 Inverted_Turn
  | Ornaments_5 Shake
  | Ornaments_6 Wavy_Line
  | Ornaments_7 Mordent
  | Ornaments_8 Inverted_Mordent
  | Ornaments_9 Schleifer
  | Ornaments_10 Tremolo
  | Ornaments_11 Other_Ornament
    deriving (Eq, Show)
  -- |
read_Ornaments_ :: STM Result [Content i] Ornaments_
read_Ornaments_ =
  (read_Trill_Mark >>= return · Ornaments_1) `mplus`
  (read_Turn >>= return · Ornaments_2) `mplus`
  (read_Delayed_Turn >>= return · Ornaments_3) `mplus`
  (read_Inverted_Turn >>= return · Ornaments_4) `mplus`
  (read_Shake >>= return · Ornaments_5) `mplus`
  (read_Wavy_Line >>= return · Ornaments_6) `mplus`
```

```
    (read_Mordent >>= return . Ornaments_7) `mplus`
    (read_Inverted_Mordent >>= return . Ornaments_8) `mplus`
    (read_Schleifer >>= return . Ornaments_9) `mplus`
    (read_Tremolo >>= return . Ornaments_10) `mplus`
    (read_Other_Ornament >>= return . Ornaments_11)
  -- |
show_Ornaments_ :: Ornaments_ -> [Content ()]
show_Ornaments_ (Ornaments_1 x) = show_Trill_Mark x
show_Ornaments_ (Ornaments_2 x) = show_Turn x
show_Ornaments_ (Ornaments_3 x) = show_Delayed_Turn x
show_Ornaments_ (Ornaments_4 x) = show_Inverted_Turn x
show_Ornaments_ (Ornaments_5 x) = show_Shake x
show_Ornaments_ (Ornaments_6 x) = show_Wavy_Line x
show_Ornaments_ (Ornaments_7 x) = show_Mordent x
show_Ornaments_ (Ornaments_8 x) = show_Inverted_Mordent x
show_Ornaments_ (Ornaments_9 x) = show_Schleifer x
show_Ornaments_ (Ornaments_10 x) = show_Tremolo x
show_Ornaments_ (Ornaments_11 x) = show_Other_Ornament x
  -- |
type Trill_Mark = ((Print_Style, Placement, Trill_Sound), ())
  -- |
read_Trill_Mark :: STM Result [Content i] Trill_Mark
read_Trill_Mark = do
  y <- read_ELEMENT "trill-mark"
  y1 <- read_3 read_Print_Style read_Placement read_Trill_Sound
    (attributes y)
  return (y1, ())
  -- |
show_Trill_Mark :: Trill_Mark -> [Content ()]
show_Trill_Mark ((a, b, c), _) =
  show_ELEMENT "trill-mark"
    (show_Print_Style a ++
      show_Placement b ++
      show_Trill_Sound c) []
```

The turn and delayed-turn elements are the normal turn shape which goes up then down. The delayed-turn element indicates a turn that is delayed until the end of the current note. The inverted-turn element has the shape which goes down and then up.

```
  -- |
type Turn = ((Print_Style, Placement, Trill_Sound), ())
  -- |
read_Turn :: STM Result [Content i] Turn
read_Turn = do
  y <- read_ELEMENT "turn"
  y1 <- read_3 read_Print_Style read_Placement read_Trill_Sound
    (attributes y)
  return (y1, ())
  -- |
show_Turn :: Turn -> [Content ()]
show_Turn ((a, b, c), _) =
  show_ELEMENT "turn"
    (show_Print_Style a ++
      show_Placement b ++
      show_Trill_Sound c) []
  -- |
type Delayed_Turn = ((Print_Style, Placement, Trill_Sound), ())
  -- |
read_Delayed_Turn :: STM Result [Content i] Delayed_Turn
```

```
read_Delayed_Turn = do
  y ← read_ELEMENT "delayed-turn"
  y1 ← read_3 read_Print_Style read_Placement read_Trill_Sound
    (attributes y)
  return (y1, ())
  -- |
show_Delayed_Turn :: Delayed_Turn → [Content ()]
show_Delayed_Turn ((a, b, c), _) =
  show_ELEMENT "delayed-turn"
    (show_Print_Style a ++
      show_Placement b ++
      show_Trill_Sound c) []
  -- |
type Inverted_Turn = ((Print_Style, Placement, Trill_Sound), ())
  -- |
read_Inverted_Turn :: STM Result [Content i] Inverted_Turn
read_Inverted_Turn = do
  y ← read_ELEMENT "inverted-turn"
  y1 ← read_3 read_Print_Style read_Placement read_Trill_Sound
    (attributes y)
  return (y1, ())
  -- |
show_Inverted_Turn :: Inverted_Turn → [Content ()]
show_Inverted_Turn ((a, b, c), _) =
  show_ELEMENT "inverted-turn"
    (show_Print_Style a ++
      show_Placement b ++
      show_Trill_Sound c) []
  -- |
type Shake = ((Print_Style, Placement, Trill_Sound), ())
  -- |
read_Shake :: STM Result [Content i] Shake
read_Shake = do
  y ← read_ELEMENT "shake"
  y1 ← read_3 read_Print_Style read_Placement read_Trill_Sound
    (attributes y)
  return (y1, ())
  -- |
show_Shake :: Shake → [Content ()]
show_Shake ((a, b, c), _) =
  show_ELEMENT "shake"
    (show_Print_Style a ++
      show_Placement b ++
      show_Trill_Sound c) []
```

The wavy-line element is defined in the Common.lhs file, as it applies to more than just note elements.

The long attribute for the mordent and inverted-mordent elements is "no" by default. The mordent element represents the sign with the vertical line; the inverted-mordent element represents the sign without the vertical line.

```
  -- |
type Mordent = ((Maybe Yes_No, Print_Style, Placement, Trill_Sound), ())
  -- |
read_Mordent :: STM Result [Content i] Mordent
read_Mordent = do
  y ← read_ELEMENT "mordent"
  y1 ← read_4 (read_IMPLIED "long" read_Yes_No)
    read_Print_Style read_Placement read_Trill_Sound
    (attributes y)
```

```
    return (y1, ())
  -- |
show_Mordent :: Mordent → [Content ()]
show_Mordent ((a, b, c, d), _) =
  show_ELEMENT "mordent"
    (show_IMPLIED [] show_Yes_No a ++
      show_Print_Style b ++
      show_Placement c ++
      show_Trill_Sound d) []
  -- |
type Inverted_Mordent = ((Maybe Yes_No, Print_Style, Placement, Trill_Sound), ())
  -- |
read_Inverted_Mordent :: STM Result [Content i] Inverted_Mordent
read_Inverted_Mordent = do
  y ← read_ELEMENT "inverted-mordent"
  y1 ← read_4 (read_IMPLIED "long" read_Yes_No)
    read_Print_Style read_Placement read_Trill_Sound
    (attributes y)
  return (y1, ())
  -- |
show_Inverted_Mordent :: Inverted_Mordent → [Content ()]
show_Inverted_Mordent ((a, b, c, d), _) =
  show_ELEMENT "inverted-mordent"
    (show_IMPLIED [] show_Yes_No a ++
      show_Print_Style b ++
      show_Placement c ++
      show_Trill_Sound d) []
```

The name for this ornament is based on the German, to avoid confusion with the more common slide element defined earlier.

```
  -- |
type Schleifer = ((Print_Style, Placement), ())
  -- |
read_Schleifer :: STM Result [Content i] Schleifer
read_Schleifer = do
  y ← read_ELEMENT "schleifer"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Schleifer :: Schleifer → [Content ()]
show_Schleifer ((a, b), _) =
  show_ELEMENT "schleifer" (show_Print_Style a ++ show_Placement b) []
```

While using repeater beams is the preferred method for indicating tremolos, often playback and display are not well-enough integrated in an application to make that feasible. The tremolo ornament can be used to indicate either single-note or double-note tremolos. Single-note tremolos use the single type, while double-note tremolos use the start and stop types. The default is "single" for compatibility with Version 1.1. The text of the element indicates the number of tremolo marks and is an integer from 0 to 6. Note that the number of attached beams is not included in this value, but is represented separately using the beam element.

```
  -- |
type Tremolo = ((Start_Stop_Single, Print_Style, Placement), PCDATA)
  -- |
read_Tremolo :: STM Result [Content i] Tremolo
read_Tremolo = do
  y ← read_ELEMENT "tremolo"
  y1 ← read_3 (read_DEFAULT "type" read_Start_Stop_Single Start_Stop_Single_3)
    read_Print_Style read_Placement (attributes y)
```

```
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Tremolo :: Tremolo → [Content ()]
show_Tremolo ((a, b, c), d) =
  show_ELEMENT "tremolo"
    (show_IMPLIED "type" show_Start_Stop_Single (Just a) ++
      show_Print_Style b ++ show_Placement c)
    (show_PCDATA d)
```

The other-ornament element is used to define any ornaments not yet in the MusicXML format. This allows extended representation, though without application interoperability.

```
  -- |
type Other_Ornament = ((Print_Style, Placement), PCDATA)
  -- |
read_Other_Ornament :: STM Result [Content i] Other_Ornament
read_Other_Ornament = do
  y ← read_ELEMENT "other-ornament"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Other_Ornament :: Other_Ornament → [Content ()]
show_Other_Ornament ((a, b), c) =
  show_ELEMENT "other-ornament"
    (show_Print_Style a ++ show_Placement b)
    (show_PCDATA c)
```

An accidental-mark can be used as a separate notation or as part of an ornament. When used in an ornament, position and placement are relative to the ornament, not relative to the note.

```
  -- |
type Accidental_Mark = ((Print_Style, Placement), CDATA)
  -- |
read_Accidental_Mark :: STM Result [Content i] Accidental_Mark
read_Accidental_Mark = do
  y ← read_ELEMENT "accidental-mark"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Accidental_Mark :: Accidental_Mark → [Content ()]
show_Accidental_Mark ((a, b), c) =
  show_ELEMENT "accidental-mark"
    (show_Print_Style a ++ show_Placement b)
    (show_PCDATA c)
```

Technical indications give performance information for individual instruments.

```
  -- *** Technical
  -- |
type Technical = [Technical_]
  -- |
read_Technical :: Eq i ⇒ STM Result [Content i] Technical
read_Technical = do
  y ← read_ELEMENT "technical"
  read_1 (read_LIST read_Technical_) (childs y)
  -- |
show_Technical :: Technical → [Content ()]
```

```
show_Technical x =
  show_ELEMENT "technical" [] (show_LIST show_Technical_ x)
  -- |
data Technical_ = Technical_1 Up_Bow
  | Technical_2 Down_Bow
  | Technical_3 Harmonic
  | Technical_4 Open_String
  | Technical_5 Thumb_Position
  | Technical_6 Fingering
  | Technical_7 Pluck
  | Technical_8 Double_Tongue
  | Technical_9 Triple_Tongue
  | Technical_10 Stopped
  | Technical_11 Snap_Pizzicato
  | Technical_12 Fret
  | Technical_13 String
  | Technical_14 Hammer_On
  | Technical_15 Pull_Off
  | Technical_16 Bend
  | Technical_17 Tap
  | Technical_18 Heel
  | Technical_19 Toe
  | Technical_20 Fingernails
  | Technical_21 Other_Technical
    deriving (Eq, Show)
  -- |
read_Technical_ :: STM Result [Content i] Technical_
read_Technical_ =
  (read_Up_Bow >>= return . Technical_1) `mplus`
  (read_Down_Bow >>= return . Technical_2) `mplus`
  (read_Harmonic >>= return . Technical_3) `mplus`
  (read_Open_String >>= return . Technical_4) `mplus`
  (read_Thumb_Position >>= return . Technical_5) `mplus`
  (read_Fingering >>= return . Technical_6) `mplus`
  (read_Pluck >>= return . Technical_7) `mplus`
  (read_Double_Tongue >>= return . Technical_8) `mplus`
  (read_Triple_Tongue >>= return . Technical_9) `mplus`
  (read_Stopped >>= return . Technical_10) `mplus`
  (read_Snap_Pizzicato >>= return . Technical_11) `mplus`
  (read_Fret >>= return . Technical_12) `mplus`
  (read_String >>= return . Technical_13) `mplus`
  (read_Hammer_On >>= return . Technical_14) `mplus`
  (read_Pull_Off >>= return . Technical_15) `mplus`
  (read_Bend >>= return . Technical_16) `mplus`
  (read_Tap >>= return . Technical_17) `mplus`
  (read_Heel >>= return . Technical_18) `mplus`
  (read_Toe >>= return . Technical_19) `mplus`
  (read_Fingernails >>= return . Technical_20) `mplus`
  (read_Other_Technical >>= return . Technical_21)
  -- |
show_Technical_ :: Technical_ -> [Content ()]
show_Technical_ (Technical_1 x) = show_Up_Bow x
show_Technical_ (Technical_2 x) = show_Down_Bow x
show_Technical_ (Technical_3 x) = show_Harmonic x
show_Technical_ (Technical_4 x) = show_Open_String x
show_Technical_ (Technical_5 x) = show_Thumb_Position x
show_Technical_ (Technical_6 x) = show_Fingering x
show_Technical_ (Technical_7 x) = show_Pluck x
```

```
show_Technical_ (Technical_8 x) = show_Double_Tongue x
show_Technical_ (Technical_9 x) = show_Triple_Tongue x
show_Technical_ (Technical_10 x) = show_Stopped x
show_Technical_ (Technical_11 x) = show_Snap_Pizzicato x
show_Technical_ (Technical_12 x) = show_Fret x
show_Technical_ (Technical_13 x) = show_String x
show_Technical_ (Technical_14 x) = show_Hammer_On x
show_Technical_ (Technical_15 x) = show_Pull_Off x
show_Technical_ (Technical_16 x) = show_Bend x
show_Technical_ (Technical_17 x) = show_Tap x
show_Technical_ (Technical_18 x) = show_Heel x
show_Technical_ (Technical_19 x) = show_Toe x
show_Technical_ (Technical_20 x) = show_Fingernails x
show_Technical_ (Technical_21 x) = show_Other_Technical x
```

The up-bow and down-bow elements represent the symbol that is used both for bowing indications on bowed instruments, and up-stroke / down-stoke indications on plucked instruments.

```
  -- |
type Up_Bow = ((Print_Style, Placement), ())
  -- |
read_Up_Bow :: STM Result [Content i] Up_Bow
read_Up_Bow = do
  y ← read_ELEMENT "up-bow"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Up_Bow :: Up_Bow → [Content ()]
show_Up_Bow ((a, b), _) =
  show_ELEMENT "up-bow"
    (show_Print_Style a ++ show_Placement b) []
  -- |
type Down_Bow = ((Print_Style, Placement), ())
  -- |
read_Down_Bow :: STM Result [Content i] Down_Bow
read_Down_Bow = do
  y ← read_ELEMENT "down-bow"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Down_Bow :: Down_Bow → [Content ()]
show_Down_Bow ((a, b), _) =
  show_ELEMENT "down-bow"
    (show_Print_Style a ++ show_Placement b) []
```

The harmonic element indicates natural and artificial harmonics. Natural harmonics usually notate the base pitch rather than the sounding pitch. Allowing the type of pitch to be specified, combined with controls for appearance/playback differences, allows both the notation and the sound to be represented. Artificial harmonics can add a notated touching-pitch; the pitch or fret at which the string is touched lightly to produce the harmonic. Artificial pinch harmonics will usually not notate a touching pitch. The attributes for the harmonic element refer to the use of the circular harmonic symbol, typically but not always used with natural harmonics.

```
type Harmonic = ((Print_Object, Print_Style, Placement),
  (Maybe Harmonic_A, Maybe Harmonic_B))
  -- |
read_Harmonic :: STM Result [Content i] Harmonic
read_Harmonic = do
  y ← read_ELEMENT "harmonic"
  y1 ← read_3 read_Print_Object read_Print_Style
```

```
      read_Placement (attributes y)
    y2 ← read_2 (read_MAYBE read_Harmonic_A)
      (read_MAYBE read_Harmonic_B) (childs y)
    return (y1, y2)
    -- |
show_Harmonic :: Harmonic → [Content ()]
show_Harmonic ((a, b, c), (d, e)) =
    show_ELEMENT "harmonic"
      (show_Print_Object a ++
        show_Print_Style b ++
        show_Placement c)
      (show_MAYBE show_Harmonic_A d ++
        show_MAYBE show_Harmonic_B e)
data Harmonic_A = Harmonic_1 Natural
    | Harmonic_2 Artificial
      deriving (Eq, Show)
    -- |
read_Harmonic_A :: STM Result [Content i] Harmonic_A
read_Harmonic_A =
    (read_Natural >>= return · Harmonic_1) `mplus`
    (read_Artificial >>= return · Harmonic_2)
    -- |
show_Harmonic_A :: Harmonic_A → [Content ()]
show_Harmonic_A (Harmonic_1 x) = show_Natural x
show_Harmonic_A (Harmonic_2 x) = show_Artificial x
    -- |
data Harmonic_B = Harmonic_3 Base_Pitch
    | Harmonic_4 Touching_Pitch
    | Harmonic_5 Sounding_Pitch
      deriving (Eq, Show)
    -- |
read_Harmonic_B :: STM Result [Content i] Harmonic_B
read_Harmonic_B =
    (read_Base_Pitch >>= return · Harmonic_3) `mplus`
    (read_Touching_Pitch >>= return · Harmonic_4) `mplus`
    (read_Sounding_Pitch >>= return · Harmonic_5)
    -- |
show_Harmonic_B :: Harmonic_B → [Content ()]
show_Harmonic_B (Harmonic_3 x) = show_Base_Pitch x
show_Harmonic_B (Harmonic_4 x) = show_Touching_Pitch x
show_Harmonic_B (Harmonic_5 x) = show_Sounding_Pitch x
    -- |
type Natural = ()
    -- |
read_Natural :: STM Result [Content i] Natural
read_Natural = do
    read_ELEMENT "natural" >> return ()
    -- |
show_Natural :: Natural → [Content ()]
show_Natural _ = show_ELEMENT "natural" [] []
    -- |
type Artificial = ()
    -- |
read_Artificial :: STM Result [Content i] Artificial
read_Artificial = do
    read_ELEMENT "artificial" >> return ()
    -- |
show_Artificial :: Artificial → [Content ()]
```

```
show_Artificial _ = show_ELEMENT "artificial" [] []
  -- |
type Base_Pitch = ()
  -- |
read_Base_Pitch :: STM Result [Content i] Base_Pitch
read_Base_Pitch = do
  read_ELEMENT "base-picth" >> return ()
  -- |
show_Base_Pitch :: Base_Pitch -> [Content ()]
show_Base_Pitch _ = show_ELEMENT "base-pitch" [] []
  -- |
type Touching_Pitch = ()
  -- |
read_Touching_Pitch :: STM Result [Content i] Touching_Pitch
read_Touching_Pitch = do
  read_ELEMENT "touching-pitch" >> return ()
  -- |
show_Touching_Pitch :: Touching_Pitch -> [Content ()]
show_Touching_Pitch _ = show_ELEMENT "touching-picth" [] []
  -- |
type Sounding_Pitch = ()
  -- |
read_Sounding_Pitch :: STM Result [Content i] Sounding_Pitch
read_Sounding_Pitch = do
  read_ELEMENT "sounding-picth" >> return ()
  -- |
show_Sounding_Pitch :: Sounding_Pitch -> [Content ()]
show_Sounding_Pitch _ = show_ELEMENT "sounding-pitch" [] []
  -- |
type Open_String = ((Print_Style, Placement), ())
  -- |
read_Open_String :: STM Result [Content i] Open_String
read_Open_String = do
  y <- read_ELEMENT "open-string"
  y1 <- read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Open_String :: Open_String -> [Content ()]
show_Open_String ((a, b), _) =
  show_ELEMENT "open-string"
    (show_Print_Style a ++ show_Placement b) []
  -- |
type Thumb_Position = ((Print_Style, Placement), ())
  -- |
read_Thumb_Position :: STM Result [Content i] Thumb_Position
read_Thumb_Position = do
  y <- read_ELEMENT "thumb-position"
  y1 <- read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Thumb_Position :: Thumb_Position -> [Content ()]
show_Thumb_Position ((a, b), _) =
  show_ELEMENT "thumb-position"
    (show_Print_Style a ++ show_Placement b) []
```

The pluck element is used to specify the plucking fingering on a fretted instrument, where the fingering element refers to the fretting fingering. Typical values are p, i, m, a for pulgar/thumb, indicio/index, medio/middle, and anular/ring fingers.

```
  -- |
type Pluck = ((Print_Style, Placement), PCDATA)
  -- |
read_Pluck :: STM Result [Content i] Pluck
read_Pluck = do
  y ← read_ELEMENT "pluck"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Pluck :: Pluck → [Content ()]
show_Pluck ((a, b), c) =
  show_ELEMENT "pluck"
    (show_Print_Style a ++ show_Placement b)
    (show_PCDATA c)
  -- |
type Double_Tongue = ((Print_Style, Placement), ())
  -- |
read_Double_Tongue :: STM Result [Content i] Double_Tongue
read_Double_Tongue = do
  y ← read_ELEMENT "double-tongue"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Double_Tongue :: Double_Tongue → [Content ()]
show_Double_Tongue ((a, b), _) =
  show_ELEMENT "double-tongue"
    (show_Print_Style a ++ show_Placement b) []
  -- |
type Triple_Tongue = ((Print_Style, Placement), ())
  -- |
read_Triple_Tongue :: STM Result [Content i] Triple_Tongue
read_Triple_Tongue = do
  y ← read_ELEMENT "triple-tongue"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Triple_Tongue :: Triple_Tongue → [Content ()]
show_Triple_Tongue ((a, b), _) =
  show_ELEMENT "triple-tongue"
    (show_Print_Style a ++ show_Placement b) []
  -- |
type Stopped = ((Print_Style, Placement), ())
  -- |
read_Stopped :: STM Result [Content i] Stopped
read_Stopped = do
  y ← read_ELEMENT "stopped"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Stopped :: Stopped → [Content ()]
show_Stopped ((a, b), _) =
  show_ELEMENT "stopped"
    (show_Print_Style a ++ show_Placement b) []
  -- |
type Snap_Pizzicato = ((Print_Style, Placement), ())
  -- |
read_Snap_Pizzicato :: STM Result [Content i] Snap_Pizzicato
```

```
read_Snap_Pizzicato = do
  y ← read_ELEMENT "snap-pizzicato"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Snap_Pizzicato :: Snap_Pizzicato → [Content ()]
show_Snap_Pizzicato ((a, b), _) =
  show_ELEMENT "snap-pizzicato"
    (show_Print_Style a ++ show_Placement b) []
```

The hammer-on and pull-off elements are used in guitar and fretted instrument notation. Since a single slur can be marked over many notes, the hammer-on and pull-off elements are separate so the individual pair of notes can be specified. The element content can be used to specify how the hammer-on or pull-off should be notated. An empty element leaves this choice up to the application.

```
  -- |
type Hammer_On = ((Start_Stop, Number_Level, Print_Style, Placement), PCDATA)
  -- |
read_Hammer_On :: STM Result [Content i] Hammer_On
read_Hammer_On = do
  y ← read_ELEMENT "hammer-on"
  y1 ← read_4 (read_REQUIRED "type" read_Start_Stop)
    (read_DEFAULT "number" read_Number_Level Number_Level_1)
      read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Hammer_On :: Hammer_On → [Content ()]
show_Hammer_On ((a, b, c, d), e) =
  show_ELEMENT "hammer-on"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_IMPLIED "number" show_Number_Level (Just b) ++
      show_Print_Style c ++ show_Placement d)
    (show_PCDATA e)
  -- |
type Pull_Off = ((Start_Stop, Number_Level, Print_Style, Placement), PCDATA)
  -- |
read_Pull_Off :: STM Result [Content i] Pull_Off
read_Pull_Off = do
  y ← read_ELEMENT "pull-off"
  y1 ← read_4 (read_REQUIRED "type" read_Start_Stop)
    (read_DEFAULT "number" read_Number_Level Number_Level_1)
    read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Pull_Off :: Pull_Off → [Content ()]
show_Pull_Off ((a, b, c, d), e) =
  show_ELEMENT "pull-off"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_IMPLIED "number" show_Number_Level (Just b) ++
      show_Print_Style c ++ show_Placement d)
    (show_PCDATA e)
```

The bend element is used in guitar and tablature. The bend-alter element indicates the number of steps in the bend, similar to the alter element. As with the alter element, numbers like 0.5 can be used to indicate microtones. Negative numbers indicate pre-bends or releases; the pre-bend and release elements are used to distinguish what is intended. A with-bar element indicates that the bend is to be done at the bridge with a whammy or vibrato bar. The content of the element indicates how this should be notated.

```
  -- |
type Bend = ((Print_Style, Bend_Sound),
  (Bend_Alter, Maybe Bend_, Maybe With_Bar))
  -- |
read_Bend :: STM Result [Content i] Bend
read_Bend = do
  y ← read_ELEMENT "bend"
  y1 ← read_2 read_Print_Style read_Bend_Sound (attributes y)
  y2 ← read_3 read_Bend_Alter (read_MAYBE read_Bend_)
    (read_MAYBE read_With_Bar) (childs y)
  return (y1, y2)
  -- |
show_Bend :: Bend → [Content ()]
show_Bend ((a, b), (c, d, e)) =
  show_ELEMENT "bend"
    (show_Print_Style a ++ show_Bend_Sound b)
    (show_Bend_Alter c ++
      show_MAYBE show_Bend_ d ++
      show_MAYBE show_With_Bar e)
  -- |
data Bend_ = Bend_1 Pre_Bend | Bend_2 Release
  deriving (Eq, Show)
  -- |
read_Bend_ :: STM Result [Content i] Bend_
read_Bend_ =
  (read_Pre_Bend >>= return · Bend_1) `mplus`
  (read_Release >>= return · Bend_2)
  -- |
show_Bend_ :: Bend_ → [Content ()]
show_Bend_ (Bend_1 x) = show_Pre_Bend x
show_Bend_ (Bend_2 x) = show_Release x
  -- |
type Bend_Alter = PCDATA
  -- |
read_Bend_Alter :: STM Result [Content i] Bend_Alter
read_Bend_Alter = do
  y ← read_ELEMENT "bend-alter"
  read_1 read_PCDATA (childs y)
  -- |
show_Bend_Alter :: Bend_Alter → [Content ()]
show_Bend_Alter a = show_ELEMENT "bend-alter" [] (show_PCDATA a)
  -- |
type Pre_Bend = ()
  -- |
read_Pre_Bend :: STM Result [Content i] Pre_Bend
read_Pre_Bend = read_ELEMENT "pre-bend" >> return ()
  -- |
show_Pre_Bend :: Pre_Bend → [Content ()]
show_Pre_Bend _ = show_ELEMENT "pre-bend" [] []
  -- |
type Release = ()
  -- |
read_Release :: STM Result [Content i] Release
read_Release = read_ELEMENT "release" >> return ()
  -- |
show_Release :: Release → [Content ()]
show_Release _ = show_ELEMENT "release" [] []
  -- |
```

```
type With_Bar = ((Print_Style, Placement), CDATA)
  -- |
read_With_Bar :: STM Result [Content i] With_Bar
read_With_Bar = do
  y ← read_ELEMENT "with-bar"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_With_Bar :: With_Bar → [Content ()]
show_With_Bar ((a, b), c) =
  show_ELEMENT "with-bar"
    (show_Print_Style a ++ show_Placement b)
    (show_PCDATA c)
```

The tap element indicates a tap on the fretboard. The element content allows specification of the notation; + and T are common choices. If empty, the display is application-specific.

```
  -- |
type Tap = ((Print_Style, Placement), CDATA)
  -- |
read_Tap :: STM Result [Content i] Tap
read_Tap = do
  y ← read_ELEMENT "tap"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Tap :: Tap → [Content ()]
show_Tap ((a, b), c) =
  show_ELEMENT "tap"
    (show_Print_Style a ++ show_Placement b)
    (show_PCDATA c)
```

The heel and toe element are used with organ pedals. The substitution value is "no" if the attribute is not present.

```
  -- |
type Heel = ((Maybe Yes_No, Print_Style, Placement), ())
  -- |
read_Heel :: STM Result [Content i] Heel
read_Heel = do
  y ← read_ELEMENT "heel"
  y1 ← read_3 (read_IMPLIED "substitution" read_Yes_No)
    read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Heel :: Heel → [Content ()]
show_Heel ((a, b, c), _) =
  show_ELEMENT "heel"
    (show_IMPLIED "substitution" show_Yes_No a ++
      show_Print_Style b ++
      show_Placement c) []
  -- |
type Toe = ((Maybe Yes_No, Print_Style, Placement), ())
  -- |
read_Toe :: STM Result [Content i] Toe
read_Toe = do
  y ← read_ELEMENT "toe"
  y1 ← read_3 (read_IMPLIED "substitution" read_Yes_No)
```

```
      read_Print_Style read_Placement (attributes y)
    return (y1, ())
  -- |
show_Toe :: Toe → [Content ()]
show_Toe ((a, b, c), _) =
  show_ELEMENT "toe"
    (show_IMPLIED "substitution" show_Yes_No a ++
      show_Print_Style b ++
      show_Placement c) []
```

The fingernails element is used in harp notation.

```
  -- |
type Fingernails = ((Print_Style, Placement), ())
  -- |
read_Fingernails :: STM Result [Content i] Fingernails
read_Fingernails = do
  y ← read_ELEMENT "fingernails"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Fingernails :: Fingernails → [Content ()]
show_Fingernails ((a, b), _) =
  show_ELEMENT "fingernails"
    (show_Print_Style a ++ show_Placement b) []
```

The other-technical element is used to define any technical indications not yet in the MusicXML format. This allows extended representation, though without application interoperability.

```
  -- |
type Other_Technical = ((Print_Style, Placement), CDATA)
  -- |
read_Other_Technical :: STM Result [Content i] Other_Technical
read_Other_Technical = do
  y ← read_ELEMENT "other-technical"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Other_Technical :: Other_Technical → [Content ()]
show_Other_Technical ((a, b), c) =
  show_ELEMENT "other-technical"
    (show_Print_Style a ++ show_Placement b)
    (show_PCDATA c)
```

Articulations and accents are grouped together here.

```
  -- *** Articulations
  -- |
type Articulations = [Articulations_]
  -- |
read_Articulations :: Eq i ⇒ STM Result [Content i] Articulations
read_Articulations = do
  y ← read_ELEMENT "articulations"
  read_1 (read_LIST read_Articulations_) (childs y)
  -- |
show_Articulations :: Articulations → [Content ()]
show_Articulations a =
  show_ELEMENT "articulations" [] (show_LIST show_Articulations_ a)
data Articulations_ = Articulations_1 Accent
```

```
    | Articulations_2 Strong_Accent
    | Articulations_3 Staccato
    | Articulations_4 Tenuto
    | Articulations_5 Detached_Legato
    | Articulations_6 Staccatissimo
    | Articulations_7 Spiccato
    | Articulations_8 Scoop
    | Articulations_9 Plop
    | Articulations_10 Doit
    | Articulations_11 Falloff
    | Articulations_12 Breath_Mark
    | Articulations_13 Caesura
    | Articulations_14 Stress
    | Articulations_15 Unstress
    | Articulations_16 Other_Articulation
      deriving (Eq, Show)
  -- |
read_Articulations_ :: STM Result [Content i] Articulations_
read_Articulations_ =
  (read_Accent >>= return . Articulations_1) `mplus`
  (read_Strong_Accent >>= return . Articulations_2) `mplus`
  (read_Staccato >>= return . Articulations_3) `mplus`
  (read_Tenuto >>= return . Articulations_4) `mplus`
  (read_Detached_Legato >>= return . Articulations_5) `mplus`
  (read_Staccatissimo >>= return . Articulations_6) `mplus`
  (read_Spiccato >>= return . Articulations_7) `mplus`
  (read_Scoop >>= return . Articulations_8) `mplus`
  (read_Plop >>= return . Articulations_9) `mplus`
  (read_Doit >>= return . Articulations_10) `mplus`
  (read_Falloff >>= return . Articulations_11) `mplus`
  (read_Breath_Mark >>= return . Articulations_12) `mplus`
  (read_Caesura >>= return . Articulations_13) `mplus`
  (read_Stress >>= return . Articulations_14) `mplus`
  (read_Unstress >>= return . Articulations_15) `mplus`
  (read_Other_Articulation >>= return . Articulations_16)
  -- |
show_Articulations_ :: Articulations_ -> [Content ()]
show_Articulations_ (Articulations_1 x) = show_Accent x
show_Articulations_ (Articulations_2 x) = show_Strong_Accent x
show_Articulations_ (Articulations_3 x) = show_Staccato x
show_Articulations_ (Articulations_4 x) = show_Tenuto x
show_Articulations_ (Articulations_5 x) = show_Detached_Legato x
show_Articulations_ (Articulations_6 x) = show_Staccatissimo x
show_Articulations_ (Articulations_7 x) = show_Spiccato x
show_Articulations_ (Articulations_8 x) = show_Scoop x
show_Articulations_ (Articulations_9 x) = show_Plop x
show_Articulations_ (Articulations_10 x) = show_Doit x
show_Articulations_ (Articulations_11 x) = show_Falloff x
show_Articulations_ (Articulations_12 x) = show_Breath_Mark x
show_Articulations_ (Articulations_13 x) = show_Caesura x
show_Articulations_ (Articulations_14 x) = show_Stress x
show_Articulations_ (Articulations_15 x) = show_Unstress x
show_Articulations_ (Articulations_16 x) = show_Other_Articulation x
  -- |
type Accent = ((Print_Style, Placement), ())
  -- |
read_Accent :: STM Result [Content i] Accent
read_Accent = do
```

```
    y ← read_ELEMENT "accent"
    y1 ← read_2 read_Print_Style read_Placement (attributes y)
    return (y1, ())
    -- |
  show_Accent :: Accent → [Content ()]
  show_Accent ((a, b), _) =
    show_ELEMENT "accent"
      (show_Print_Style a ++ show_Placement b) []
    -- |
  type Strong_Accent = ((Print_Style, Placement, Up_Down), ())
    -- |
  read_Strong_Accent :: STM Result [Content i] Strong_Accent
  read_Strong_Accent = do
    y ← read_ELEMENT "strong-accent"
    y1 ← read_3 read_Print_Style read_Placement
      (read_DEFAULT "type" read_Up_Down Up_Down_1) (attributes y)
    return (y1, ())
    -- |
  show_Strong_Accent :: Strong_Accent → [Content ()]
  show_Strong_Accent ((a, b, c), _) =
    show_ELEMENT "strong-accent"
      (show_Print_Style a ++ show_Placement b ++
        show_REQUIRED "type" show_Up_Down c) []
```

The staccato element is used for a dot articulation, as opposed to a stroke or a wedge.

```
    -- |
  type Staccato = ((Print_Style, Placement), ())
    -- |
  read_Staccato :: STM Result [Content i] Staccato
  read_Staccato = do
    y ← read_ELEMENT "staccato"
    y1 ← read_2 read_Print_Style read_Placement (attributes y)
    return (y1, ())
    -- |
  show_Staccato :: Staccato → [Content ()]
  show_Staccato ((a, b), _) =
    show_ELEMENT "staccato"
      (show_Print_Style a ++ show_Placement b) []
    -- |
  type Tenuto = ((Print_Style, Placement), ())
    -- |
  read_Tenuto :: STM Result [Content i] Tenuto
  read_Tenuto = do
    y ← read_ELEMENT "tenuto"
    y1 ← read_2 read_Print_Style read_Placement (attributes y)
    return (y1, ())
    -- |
  show_Tenuto :: Tenuto → [Content ()]
  show_Tenuto ((a, b), _) =
    show_ELEMENT "tenuto"
      (show_Print_Style a ++ show_Placement b) []
    -- |
  type Detached_Legato = ((Print_Style, Placement), ())
    -- |
  read_Detached_Legato :: STM Result [Content i] Detached_Legato
  read_Detached_Legato = do
    y ← read_ELEMENT "detached-legato"
    y1 ← read_2 read_Print_Style read_Placement (attributes y)
```

```
  return (y1, ())
  -- |
show_Detached_Legato :: Detached_Legato → [Content ()]
show_Detached_Legato ((a, b), _) =
  show_ELEMENT "detached-legato"
    (show_Print_Style a ++ show_Placement b) []
```

The staccatissimo element is used for a wedge articulation, as opposed to a dot or a stroke.

```
  -- |
type Staccatissimo = ((Print_Style, Placement), ())
  -- |
read_Staccatissimo :: STM Result [Content i] Staccato
read_Staccatissimo = do
  y ← read_ELEMENT "staccatissimo"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Staccatissimo :: Staccatissimo → [Content ()]
show_Staccatissimo ((a, b), _) =
  show_ELEMENT "staccatissimo"
    (show_Print_Style a ++ show_Placement b) []
```

The spiccato element is used for a stroke articulation, as opposed to a dot or a wedge.

```
  -- |
type Spiccato = ((Print_Style, Placement), ())
  -- |
read_Spiccato :: STM Result [Content i] Spiccato
read_Spiccato = do
  y ← read_ELEMENT "spiccato"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Spiccato :: Spiccato → [Content ()]
show_Spiccato ((a, b), _) =
  show_ELEMENT "spiccato"
    (show_Print_Style a ++ show_Placement b) []
```

The scoop, plop, doit, and falloff elements are indeterminate slides attached to a single note. Scoops and plops come before the main note, coming from below and above the pitch, respectively. Doits and falloffs come after the main note, going above and below the pitch, respectively.

```
  -- |
type Scoop = ((Line_Shape, Line_Type, Print_Style, Placement), ())
  -- |
read_Scoop :: STM Result [Content i] Scoop
read_Scoop = do
  y ← read_ELEMENT "scoop"
  y1 ← read_4 read_Line_Shape read_Line_Type
    read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Scoop :: Scoop → [Content ()]
show_Scoop ((a, b, c, d), _) =
  show_ELEMENT "scoop"
    (show_Line_Shape a ++ show_Line_Type b ++
      show_Print_Style c ++ show_Placement d) []
  -- |
type Plop = ((Line_Shape, Line_Type, Print_Style, Placement), ())
```

```
  -- |
read_Plop :: STM Result [Content i] Plop
read_Plop = do
  y ← read_ELEMENT "plop"
  y1 ← read_4 read_Line_Shape read_Line_Type
    read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Plop :: Plop → [Content ()]
show_Plop ((a, b, c, d), _) =
  show_ELEMENT "plop"
    (show_Line_Shape a ++ show_Line_Type b ++
      show_Print_Style c ++ show_Placement d) []
  -- |
type Doit = ((Line_Shape, Line_Type, Print_Style, Placement), ())
  -- |
read_Doit :: STM Result [Content i] Doit
read_Doit = do
  y ← read_ELEMENT "doit"
  y1 ← read_4 read_Line_Shape read_Line_Type
    read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Doit :: Doit → [Content ()]
show_Doit ((a, b, c, d), _) =
  show_ELEMENT "doit"
    (show_Line_Shape a ++ show_Line_Type b ++
      show_Print_Style c ++ show_Placement d) []
  -- |
type Falloff = ((Line_Shape, Line_Type, Print_Style, Placement), ())
  -- |
read_Falloff :: STM Result [Content i] Falloff
read_Falloff = do
  y ← read_ELEMENT "falloff"
  y1 ← read_4 read_Line_Shape read_Line_Type
    read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Falloff :: Falloff → [Content ()]
show_Falloff ((a, b, c, d), _) =
  show_ELEMENT "falloff"
    (show_Line_Shape a ++ show_Line_Type b ++
      show_Print_Style c ++ show_Placement d) []
  -- |
type Breath_Mark = ((Print_Style, Placement), ())
  -- |
read_Breath_Mark :: STM Result [Content i] Breath_Mark
read_Breath_Mark = do
  y ← read_ELEMENT "breath-mark"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Breath_Mark :: Breath_Mark → [Content ()]
show_Breath_Mark ((a, b), _) =
  show_ELEMENT "breath-mark"
    (show_Print_Style a ++ show_Placement b) []
  -- |
type Caesura = ((Print_Style, Placement), ())
```

```
  -- |
read_Caesura :: STM Result [Content i] Caesura
read_Caesura = do
  y ← read_ELEMENT "caesura"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Caesura :: Caesura → [Content ()]
show_Caesura ((a, b), _) =
  show_ELEMENT "caesura"
    (show_Print_Style a ++ show_Placement b) []
  -- |
type Stress = ((Print_Style, Placement), ())
  -- |
read_Stress :: STM Result [Content i] Stress
read_Stress = do
  y ← read_ELEMENT "stress"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Stress :: Stress → [Content ()]
show_Stress ((a, b), _) =
  show_ELEMENT "stress"
    (show_Print_Style a ++ show_Placement b) []
  -- |
type Unstress = ((Print_Style, Placement), ())
  -- |
read_Unstress :: STM Result [Content i] Unstress
read_Unstress = do
  y ← read_ELEMENT "unstress"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Unstress :: Staccato → [Content ()]
show_Unstress ((a, b), _) =
  show_ELEMENT "unstress"
    (show_Print_Style a ++ show_Placement b) []
```

The other-articulation element is used to define any articulations not yet in the MusicXML format. This allows extended representation, though without application interoperability.

```
  -- |
type Other_Articulation = ((Print_Style, Placement), ())
  -- |
read_Other_Articulation :: STM Result [Content i] Other_Articulation
read_Other_Articulation = do
  y ← read_ELEMENT "other-articulation"
  y1 ← read_2 read_Print_Style read_Placement (attributes y)
  return (y1, ())
  -- |
show_Other_Articulation :: Other_Articulation → [Content ()]
show_Other_Articulation ((a, b), _) =
  show_ELEMENT "other-articulation"
    (show_Print_Style a ++ show_Placement b) []
```

The dynamics and fermata elements are defined in the common.mod file as they apply to more than just note elements.

The arpeggiate element indicates that this note is part of an arpeggiated chord. The number attribute can be used to distinguish between two simultaneous chords arpeggiated separately (different numbers)

or together (same number). The up-down attribute is used if there is an arrow on the arpeggio sign. By default, arpeggios go from the lowest to highest note.

```
  -- *** Arpeggiate
  -- |
type Arpeggiate = ((Maybe Number_Level, Maybe Up_Down, Position, Placement,
  Color), ())
  -- |
read_Arpeggiate :: STM Result [Content i] Arpeggiate
read_Arpeggiate = do
  y ← read_ELEMENT "arpeggiate"
  y1 ← read_5 (read_IMPLIED "number" read_Number_Level)
    (read_IMPLIED "direction" read_Up_Down)
    read_Position read_Placement read_Color (attributes y)
  return (y1, ())
  -- |
show_Arpeggiate :: Arpeggiate → [Content ()]
show_Arpeggiate ((a, b, c, d, e), _) =
  show_ELEMENT "arpeggiate" (show_IMPLIED "number" show_Number_Level a ++
    show_IMPLIED "direction" show_Up_Down b ++
    show_Position c ++ show_Placement d ++
    show_Color e) []
```

The non-arpeggiate element indicates that this note is at the top or bottom of a bracket indicating to not arpeggiate these notes. Since this does not involve playback, it is only used on the top or bottom notes, not on each note as for the arpeggiate element.

```
  -- *** Non_Arpeggiate
  -- |
type Non_Arpeggiate =
  ((Top_Bottom, Maybe Number_Level, Position, Placement, Color), ())
  -- |
read_Non_Arpeggiate :: STM Result [Content i] Non_Arpeggiate
read_Non_Arpeggiate = do
  y ← read_ELEMENT "non-arpeggiate"
  y1 ← read_5 (read_REQUIRED "type" read_Top_Bottom)
    (read_IMPLIED "number" read_Number_Level)
    read_Position read_Placement read_Color
    (attributes y)
  return (y1, ())
  -- |
show_Non_Arpeggiate :: Non_Arpeggiate → [Content ()]
show_Non_Arpeggiate ((a, b, c, d, e), _) =
  show_ELEMENT "non-arpeggiate"
    (show_REQUIRED "type" show_Top_Bottom a ++
      show_IMPLIED "number" show_Number_Level b ++
      show_Position c ++ show_Placement d ++
      show_Color e) []
```

Text underlays for lyrics, based on Humdrum with support for other formats. The lyric number indicates multiple lines, though a name can be used as well (as in Finale's verse/chorus/section specification). Word extensions are represented using the extend element. Hyphenation is indicated by the syllabic element, which can be single, begin, end, or middle. These represent single-syllable words, word-beginning syllables, word-ending syllables, and mid-word syllables. Multiple syllables on a single note are separated by elision elements. A hyphen in the text element should only be used for an actual hyphenated word. Two text elements that are not separated by an elision element are part of the same syllable, but may have different text formatting.

Humming and laughing representations are taken from Humdrum. The end-line and end-paragraph elements come from RP-017 for Standard MIDI File Lyric meta-events; they help facilitate lyric display

for Karaoke and similar applications. Language names for text elements come from ISO 639, with optional country subcodes from ISO 3166. Justification is center by default; placement is below by default.

```
  -- ** Lyric
  -- |
type Lyric = ((Maybe CDATA, Maybe CDATA,
    Justify, Position, Placement, Color),
  (Lyric_, Maybe End_Line, Maybe End_Paragraph, Editorial))
read_Lyric :: Eq i ⇒ STM Result [Content i] Lyric
read_Lyric = do
  y ← read_ELEMENT "lyric"
  y1 ← read_6 (read_IMPLIED "number" read_CDATA)
    (read_IMPLIED "name" read_CDATA)
    read_Justify read_Position read_Placement
    read_Color (attributes y)
  y2 ← read_4 read_Lyric_ (read_MAYBE read_End_Line)
    (read_MAYBE read_End_Paragraph) read_Editorial
    (childs y)
  return (y1, y2)
show_Lyric :: Lyric → [Content ()]
show_Lyric ((a, b, c, d, e, f), (g, h, i, j)) =
  show_ELEMENT "lyric" (show_IMPLIED "number" show_CDATA a ++
      show_IMPLIED "name" show_CDATA b ++
      show_Justify c ++ show_Position d ++
      show_Placement e ++ show_Color f)
    (show_Lyric_ g ++ show_MAYBE show_End_Line h ++
      show_MAYBE show_End_Paragraph i ++
      show_Editorial j)
  -- |
data Lyric_ = Lyric_1 ((Maybe Syllabic, Text),
  [(Maybe Elision, Maybe Syllabic, Text)], Maybe Extend)
  | Lyric_2 Extend
  | Lyric_3 Laughing
  | Lyric_4 Humming
  deriving (Eq, Show)
read_Lyric_ :: Eq i ⇒ STM Result [Content i] Lyric_
read_Lyric_ =
  (read_Lyric_aux1 >>= (return · Lyric_1)) `mplus`
  (read_Extend >>= (return · Lyric_2)) `mplus`
  (read_Laughing >>= (return · Lyric_3)) `mplus`
  (read_Humming >>= (return · Lyric_4)) `mplus`
  fail "No lyric_ parsed"
read_Lyric_aux1 :: Eq i ⇒ STM Result [Content i] ((Maybe Syllabic, Text),
  [(Maybe Elision, Maybe Syllabic, Text)], Maybe Extend)
read_Lyric_aux1 = do
  y1 ← read_MAYBE read_Syllabic
  y2 ← read_Text
  y3 ← read_LIST read_Lyric_aux2
  y4 ← read_MAYBE read_Extend
  return ((y1, y2), y3, y4)
read_Lyric_aux2 :: STM Result [Content i] (Maybe Elision, Maybe Syllabic, Text)
read_Lyric_aux2 = do
  y1 ← read_MAYBE read_Elision
  y2 ← read_MAYBE read_Syllabic
  y3 ← read_Text
  return (y1, y2, y3)
show_Lyric_ :: Lyric_ → [Content ()]
show_Lyric_ (Lyric_1 ((a, b), c, d)) =
```

```
    show_MAYBE show_Syllabic a ++ show_Text b ++
    show_LIST show_Lyric_aux1 c ++ show_MAYBE show_Extend d
show_Lyric_ (Lyric_2 x) = show_Extend x
show_Lyric_ (Lyric_3 x) = show_Laughing x
show_Lyric_ (Lyric_4 x) = show_Humming x
show_Lyric_aux1 :: (Maybe Elision, Maybe Syllabic, Text) -> [Content ()]
show_Lyric_aux1 (a, b, c) = show_MAYBE show_Elision a ++
    show_MAYBE show_Syllabic b ++ show_Text c
    -- |
type Text = ((Font, Color, Text_Decoration, Text_Rotation, Letter_Spacing,
      Maybe CDATA, Text_Direction), CDATA)
    -- |
read_Text :: STM Result [Content i] Text
read_Text = do
    y <- read_ELEMENT "text"
    y1 <- read_7 read_Font read_Color read_Text_Decoration
      read_Text_Rotation read_Letter_Spacing
      (read_IMPLIED "xml:lang" read_CDATA)
      read_Text_Direction (attributes y)
    y2 <- read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Text :: Text -> [Content ()]
show_Text ((a, b, c, d, e, f, g), h) =
    show_ELEMENT "text"
      (show_Font a ++ show_Color b ++
        show_Text_Decoration c ++
        show_Text_Rotation d ++
        show_Letter_Spacing e ++
        show_IMPLIED "xml:lang" show_CDATA f ++
        show_Text_Direction g)
      (show_PCDATA h)
    -- |
type Syllabic = PCDATA
    -- |
read_Syllabic :: STM Result [Content i] Syllabic
read_Syllabic = do
    y <- read_ELEMENT "syllabic"
    read_1 read_PCDATA (childs y)
    -- |
show_Syllabic :: Syllabic -> [Content ()]
show_Syllabic a = show_ELEMENT "syllabic" [] (show_PCDATA a)
```

In Version 2.0, the elision element text is used to specify the symbol used to display the elision. Common values are a no-break space (Unicode 00A0), an underscore (Unicode 005F), or an undertie (Unicode 203F).

```
type Elision = ((Font, Color), CDATA)
    -- |
read_Elision :: STM Result [Content i] Elision
read_Elision = do
    y <- read_ELEMENT "elision"
    y1 <- read_2 read_Font read_Color (attributes y)
    y2 <- read_1 read_PCDATA (childs y)
    return (y1, y2)
    -- |
show_Elision :: Elision -> [Content ()]
show_Elision ((a, b), c) =
    show_ELEMENT "elision"
```

```
      (show_Font a ++ show_Color b)
      (show_PCDATA c)
type Extend = ((Font, Color), ())
  -- |
read_Extend :: STM Result [Content i] Extend
read_Extend = do
  y ← read_ELEMENT "extend"
  y1 ← read_2 read_Font read_Color (attributes y)
  return (y1, ())
  -- |
show_Extend :: Extend → [Content ()]
show_Extend ((a, b), _) =
  show_ELEMENT "extend" (show_Font a ++ show_Color b) []
  -- |
type Laughing = ()
  -- |
read_Laughing :: STM Result [Content i] Laughing
read_Laughing = read_ELEMENT "laughing" ≫ return ()
  -- |
show_Laughing :: Laughing → [Content ()]
show_Laughing _ = show_ELEMENT "laughing" [] []
  -- |
type Humming = ()
  -- |
read_Humming :: STM Result [Content i] Humming
read_Humming = read_ELEMENT "humming" ≫ return ()
  -- |
show_Humming :: Humming → [Content ()]
show_Humming _ = show_ELEMENT "humming" [] []
  -- |
type End_Line = ()
  -- |
read_End_Line :: STM Result [Content i] End_Line
read_End_Line = read_ELEMENT "end-line" ≫ return ()
  -- |
show_End_Line :: End_Line → [Content ()]
show_End_Line _ = show_ELEMENT "end-line" [] []
  -- |
type End_Paragraph = ()
  -- |
read_End_Paragraph :: STM Result [Content i] End_Paragraph
read_End_Paragraph = read_ELEMENT "end-paragraph" ≫ return ()
  -- |
show_End_Paragraph :: End_Paragraph → [Content ()]
show_End_Paragraph _ = show_ELEMENT "end-paragraph" [] []
  -- |
```

Figured bass elements take their position from the first regular note that follows. Figures are ordered from top to bottom. A figure-number is a number. Values for prefix and suffix include the accidental values sharp, flat, natural, double-sharp, flat-flat, and sharp-sharp. Suffixes include both symbols that come after the figure number and those that overstrike the figure number. The suffix value slash is used for slashed numbers indicating chromatic alteration. The orientation and display of the slash usually depends on the figure number. The prefix and suffix elements may contain additional values for symbols specific to particular figured bass styles. The value of parentheses is "no" if not present.

```
  -- |
type Figured_Bass = ((Print_Style, Printout, Maybe Yes_No),
  ([Figure], Maybe Duration, Editorial))
  -- |
```

```
read_Figured_Bass :: Eq i ⇒ STM Result [Content i] Figured_Bass
read_Figured_Bass = do
  y ← read_ELEMENT "figured-bass"
  y1 ← read_3 read_Print_Style read_Printout
    (read_IMPLIED "parentheses" read_Yes_No) (attributes y)
  y2 ← read_3 (read_LIST read_Figure) (read_MAYBE read_Duration)
    read_Editorial (childs y)
  return (y1, y2)
  -- |
show_Figured_Bass :: Figured_Bass → [Content ()]
show_Figured_Bass ((a, b, c), (d, e, f)) =
  show_ELEMENT "figured-bass"
    (show_Print_Style a ++ show_Printout b ++
      show_IMPLIED "parentheses" show_Yes_No c)
    (show_LIST show_Figure d ++ show_MAYBE show_Duration e ++
      show_Editorial f)
  -- |
type Figure = (Maybe Prefix, Maybe Figure_Number, Maybe Suffix, Maybe Extend)
  -- |
read_Figure :: STM Result [Content i] Figure
read_Figure = do
  y ← read_ELEMENT "figure"
  read_4 (read_MAYBE read_Prefix) (read_MAYBE read_Figure_Number)
    (read_MAYBE read_Suffix) (read_MAYBE read_Extend) (childs y)
  -- |
show_Figure :: Figure → [Content ()]
show_Figure (a, b, c, d) =
  show_ELEMENT "figure" []
  (show_MAYBE show_Prefix a ++
    show_MAYBE show_Figure_Number b ++
    show_MAYBE show_Suffix c ++
    show_MAYBE show_Extend d)
  -- |
type Prefix = (Print_Style, CDATA)
  -- |
read_Prefix :: STM Result [Content i] Prefix
read_Prefix = do
  y ← read_ELEMENT "prefix"
  y1 ← read_1 read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Prefix :: Prefix → [Content ()]
show_Prefix (a, b) =
  show_ELEMENT "prefix"
    (show_Print_Style a)
    (show_PCDATA b)
  -- |
type Figure_Number = (Print_Style, PCDATA)
  -- |
read_Figure_Number :: STM Result [Content i] Figure_Number
read_Figure_Number = do
  y ← read_ELEMENT "figure-number"
  y1 ← read_1 read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Figure_Number :: Figure_Number → [Content ()]
```

```
show_Figure_Number (a, b) =
  show_ELEMENT "figure-number"
    (show_Print_Style a)
    (show_PCDATA b)
  -- |
type Suffix = (Print_Style, PCDATA)
  -- |
read_Suffix :: STM Result [Content i] Suffix
read_Suffix = do
  y ← read_ELEMENT "suffix"
  y1 ← read_1 read_Print_Style (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Suffix :: Suffix → [Content ()]
show_Suffix (a, b) =
  show_ELEMENT "suffix"
    (show_Print_Style a)
    (show_PCDATA b)
```

The backup and forward elements are required to coordinate multiple voices in one part, including music on multiple staves. The forward element is generally used within voices and staves, while the backup element is generally used to move between voices and staves. Thus the backup element does not include voice or staff elements. Duration values should always be positive, and should not cross measure boundaries.

```
  -- |
type Backup = (Duration, Editorial)
  -- |
read_Backup :: STM Result [Content i] Backup
read_Backup = do
  y ← read_ELEMENT "backup"
  read_2 read_Duration read_Editorial (childs y)
  -- |
show_Backup :: Backup → [Content ()]
show_Backup (a, b) =
  show_ELEMENT "backup" []
    (show_Duration a ++
      show_Editorial b)
  -- |
type Forward = (Duration, Editorial_Voice, Maybe Staff)
  -- |
read_Forward :: STM Result [Content i] Forward
read_Forward = do
  y ← read_ELEMENT "forward"
  read_3 read_Duration read_Editorial_Voice
    (read_MAYBE read_Staff) (childs y)
  -- |
show_Forward :: Forward → [Content ()]
show_Forward (a, b, c) =
  show_ELEMENT "forward" []
    (show_Duration a ++
      show_Editorial_Voice b ++
      show_MAYBE show_Staff c)
```

2.11 Opus

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Opus where
import Text.XML.MusicXML.Common
import Text.XML.MusicXML.Link
import Text.XML.HaXml.Types (Content,
  DocTypeDecl (..), ExternalID (..), PubidLiteral (..), SystemLiteral (..))
import Control.Monad (MonadPlus (..))
import Prelude (FilePath, Maybe (..), Show, Eq, Monad (..), (++), (.),
  map, concat)
```

An opus collects MusicXML scores together into a larger entity. The individual scores could be movements in a symphony, scenes or acts in an opera, or songs in an album. The opus definition allows arbitrary nesting either via an opus (included in the document) or an opus-link (linked like scores). Future versions of the MusicXML format may expand this DTD to include reference data and other metadata related to musical scores.

Suggested use:

```
<!DOCTYPE opus PUBLIC
        "-//Recordare//DTD MusicXML 2.0 Opus//EN"
        "http://www.musicxml.org/dtds/opus.dtd">
```

```
  -- |
doctype :: DocTypeDecl
doctype = DTD "opus"
  (Just (PUBLIC (PubidLiteral "-//Recordare//DTD MusicXML 2.0 Opus//EN")
    (SystemLiteral "http://www.musicxml.org/dtds/opus.dtd")))
    []
  -- |
getFiles :: Opus -> [FilePath]
getFiles (_, (_, l)) = concat (map (\x -> getFiles_aux1 x) l)
    where getFiles_aux1 (Opus_1 o) = getFiles o
      getFiles_aux1 (Opus_2 (x, _)) = [getFiles_aux2 x]
      getFiles_aux1 (Opus_3 ((x, _), _)) = [getFiles_aux2 x]
      getFiles_aux2 (_, x, _, _, _, _) = x
```

Opus is the document element. The document-attributes entity includes the version attribute and is defined in the common.mod file.

```
  -- * Opus
  -- |
type Opus = (Document_Attributes, (Maybe Title, [Opus_]))
  -- |
read_Opus :: Eq i => STM Result [Content i] Opus
read_Opus = do
  y <- read_ELEMENT "opus"
  y1 <- read_1 read_Document_Attributes (attributes y)
  y2 <- read_2 (read_MAYBE read_Title) (read_LIST read_Opus_) (childs y)
  return (y1, y2)
  -- |
show_Opus :: Opus -> [Content ()]
show_Opus (a, (b, c)) =
  show_ELEMENT "opus"
```

```
    (show_Document_Attributes a)
    (show_MAYBE show_Title b ++ show_LIST show_Opus_ c)
  -- |
data Opus_ = Opus_1 Opus | Opus_2 Opus_Link | Opus_3 Score
  deriving (Eq, Show)
  -- |
read_Opus_ :: Eq i ⇒ STM Result [Content i] Opus_
read_Opus_ =
  (read_Opus >>= return · Opus_1) `mplus`
  (read_Opus_Link >>= return · Opus_2) `mplus`
  (read_Score >>= return · Opus_3)
  -- |
show_Opus_ :: Opus_ → [Content ()]
show_Opus_ (Opus_1 x) = show_Opus x
show_Opus_ (Opus_2 x) = show_Opus_Link x
show_Opus_ (Opus_3 x) = show_Score x
```

The score elements provide the links to the individual movements. The new-page attribute, added in Version 2.0, is used to indicate if the first page of the score is different than the last page of the previous score. If new-page is "yes", then a different page is used; if "no", then the same page is used. The default value is implementation-dependent.

```
  -- |
type Score = ((Link_Attributes, Maybe Yes_No), ())
  -- |
read_Score :: STM Result [Content i] Score
read_Score = do
  y ← read_ELEMENT "score"
  y1 ← read_2 read_Link_Attributes
    (read_IMPLIED "new-page" read_Yes_No) (attributes y)
  return (y1, ())
  -- |
show_Score :: Score → [Content ()]
show_Score ((a, b), _) =
  show_ELEMENT "score"
    (show_Link_Attributes a ++ show_IMPLIED "new-page" show_Yes_No b) []
```

An opus-link provides a link to another opus document, allowing for multiple levels of opus collections via linking as well as nesting.

```
  -- |
type Opus_Link = (Link_Attributes, ())
  -- |
read_Opus_Link :: STM Result [Content i] Opus_Link
read_Opus_Link = do
  y ← read_ELEMENT "opus-link"
  y1 ← read_1 read_Link_Attributes (attributes y)
  return (y1, ())
  -- |
show_Opus_Link :: Opus_Link → [Content ()]
show_Opus_Link (a, _) =
  show_ELEMENT "opus-link" (show_Link_Attributes a) []
```

Future versions may include metadata elements. In this version, we just include the title of the opus.

```
  -- |
type Title = PCDATA
  -- |
read_Title :: STM Result [Content i] Title
read_Title = do
```

```
  y ← read_ELEMENT "title"
  read_1 read_PCDATA (childs y)
  -- |
show_Title :: Title → [Content ()]
show_Title a =
  show_ELEMENT "title" [] (show_PCDATA a)
```

2.12 Partwise

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Partwise where
import Text.XML.MusicXML.Common
import Text.XML.MusicXML.Identity
import Text.XML.MusicXML.Score
import Text.XML.HaXml.Types (Content,
  DocTypeDecl (..), ExternalID (..), PubidLiteral (..), SystemLiteral (..))
import Prelude (Maybe (..), Monad (..), Eq, (++))
```

The MusicXML format is designed to represent musical scores, specifically common western musical notation from the 17th century onwards. It is designed as an interchange format for notation, analysis, retrieval, and performance applications. Therefore it is intended to be sufficient, not optimal, for these applications.

The MusicXML format is based on the MuseData and Humdrum formats. Humdrum explicitly represents the two-dimensional nature of musical scores by a 2-D layout notation. Since the XML format is hierarchical, we cannot do this explicitly. Instead, there are two top-level formats:

partwise.dtd Represents scores by part/instrument timewise.dtd Represents scores by time/measure

Thus partwise.dtd contains measures within each part, while timewise.dtd contains parts within each measure. XSLT stylesheets are provided to convert between the two formats.

The partwise and timewise score DTDs represent a single movement of music. Multiple movements or other musical collections are presented using opus.dtd. An opus document contains XLinks to individual scores.

Suggested use:

```
<!DOCTYPE score-partwise PUBLIC
        "-//Recordare//DTD MusicXML 2.0 Partwise//EN"
        "http://www.musicxml.org/dtds/partwise.dtd">
```

This DTD is made up of a series of component DTD modules, all of which are included here.

```
  -- |
doctype :: DocTypeDecl
doctype = DTD "score-partwise"
  (Just (PUBLIC (PubidLiteral "-//Recordare//DTD MusicXML 2.0 Partwise//EN")
    (SystemLiteral "http://www.musicxml.org/dtds/partwise.dtd")))
    []
```

The score is the root element for the DTD. It includes the score-header entity, followed either by a series of parts with measures inside (score-partwise) or a series of measures with parts inside (score-timewise). Having distinct top-level elements for partwise and timewise scores makes it easy to ensure that an XSLT stylesheet does not try to transform a document already in the desired format. The document-attributes entity includes the version attribute and is defined in the common.mod file.

In either format, the part element has an id attribute that is an IDREF back to a score-part in the part-list. Measures have a required number attribute (going from partwise to timewise, measures are grouped via the number).

The implicit attribute is set to "yes" for measures where the measure number should never appear, such as pickup measures and the last half of mid-measure repeats. The value is "no" if not specified.

The non-controlling attribute is intended for use in multimetric music like the Don Giovanni minuet. If set to "yes", the left barline in this measure does not coincide with the left barline of measures in other parts. The value is "no" if not specified.

In partwise files, the number attribute should be the same for measures in different parts that share the same left barline. While the number attribute is often numeric, it does not have to be. Non-numeric values are typically used together with the implicit or non-controlling attributes being set to "yes". For a pickup measure, the number attribute is typically set to "0" and the implicit attribute is typically set to "yes". Further details about measure numbering can be defined using the measure-numbering element defined in the direction.mod file

Measure width is specified in tenths. These are the global tenths specified in the scaling element, not local tenths as modified by the staff-size element.

```
  -- * Score_Partwise
  -- |
type Score_Partwise = (Document_Attributes, (Score_Header, [Part]))
  -- |
read_Score_Partwise :: Eq i ⇒ STM Result [Content i] Score_Partwise
read_Score_Partwise = do
  y ← read_ELEMENT "score-partwise"
  y1 ← read_1 read_Document_Attributes (attributes y)
  y2 ← read_2 read_Score_Header (read_LIST1 read_Part) (childs y)
  return (y1, y2)
  -- |
show_Score_Partwise :: Score_Partwise → [Content ()]
show_Score_Partwise (a, (b, c)) =
  show_ELEMENT "score-partwise" (show_Document_Attributes a)
    (show_Score_Header b ++
      show_LIST1 show_Part c)
  -- |
update_Score_Partwise :: ([Software], Encoding_Date) →
  Score_Partwise → Score_Partwise
update_Score_Partwise x (a, (b, c)) = (a, (update_Score_Header x b, c))
  -- |
type Part = (ID, [Measure])
  -- |
read_Part :: Eq i ⇒ STM Result [Content i] Part
read_Part = do
  y ← read_ELEMENT "part"
  y1 ← read_1 (read_REQUIRED "id" read_ID) (attributes y)
  y2 ← read_1 (read_LIST1 read_Measure) (childs y)
  return (y1, y2)
show_Part :: Part → [Content ()]
show_Part (a, b) = show_ELEMENT "part" (show_REQUIRED "id" show_ID a)
  (show_LIST1 show_Measure b)
  -- |
type Measure = ((CDATA, Maybe Yes_No, Maybe Yes_No, Maybe Tenths), Music_Data)
  -- |
read_Measure :: Eq i ⇒ STM Result [Content i] Measure
read_Measure = do
  y ← read_ELEMENT "measure"
  y1 ← read_4 (read_REQUIRED "number" read_CDATA)
    (read_IMPLIED "implicit" read_Yes_No)
    (read_IMPLIED "non-controlling" read_Yes_No)
    (read_IMPLIED "width" read_Tenths)
```

```
    (attributes y)
  y2 ← read_1 read_Music_Data (childs y)
  return (y1, y2)
  -- |
show_Measure :: Measure → [Content ()]
show_Measure ((a, b, c, d), e) =
  show_ELEMENT "measure" (show_REQUIRED "number" show_CDATA a ++
    show_IMPLIED "implicit" show_Yes_No b ++
    show_IMPLIED "non-controlling" show_Yes_No c ++
    show_IMPLIED "width" show_Tenths d)
    (show_Music_Data e)
```

2.13 Score

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Score where
import Text.XML.MusicXML.Common
import Text.XML.MusicXML.Attributes
import Text.XML.MusicXML.Link
import Text.XML.MusicXML.Barline
import Text.XML.MusicXML.Note
import Text.XML.MusicXML.Layout hiding (Tenths)
import Text.XML.MusicXML.Identity
import Text.XML.MusicXML.Direction
import Text.XML.HaXml.Types (Content)
import Control.Monad (MonadPlus (..))
import Prelude (Maybe (..), Monad (..), Functor (..), Show, Eq, (++), (·))
```

Works and movements are optionally identified by number and title. The work element also may indicate a link to the opus document that composes multiple movements into a collection.

```
  -- * Work
  -- |
type Work = (Maybe Work_Number, Maybe Work_Title, Maybe Opus)
  -- |
read_Work :: STM Result [Content i] Work
read_Work = do
  y ← read_ELEMENT "work"
  read_3 (read_MAYBE read_Work_Number)
    (read_MAYBE read_Work_Title)
    (read_MAYBE read_Opus) (childs y)
  -- |
show_Work :: Work → [Content ()]
show_Work (a, b, c) =
  show_ELEMENT "work" []
    (show_MAYBE show_Work_Number a ++
      show_MAYBE show_Work_Title b ++
      show_MAYBE show_Opus c)
  -- |
type Work_Number = PCDATA
  -- |
```

```
read_Work_Number :: STM Result [Content i] Work_Number
read_Work_Number = do
  y ← read_ELEMENT "work-number"
  read_1 read_PCDATA (childs y)
  -- |
show_Work_Number :: Work_Number → [Content ()]
show_Work_Number a = show_ELEMENT "work-number" [] (show_PCDATA a)
  -- |
type Work_Title = PCDATA
  -- |
read_Work_Title :: STM Result [Content i] Work_Title
read_Work_Title = do
  y ← read_ELEMENT "work-title"
  read_1 read_PCDATA (childs y)
  -- |
show_Work_Title :: Work_Title → [Content ()]
show_Work_Title a = show_ELEMENT "work-title" [] (show_PCDATA a)
  -- |
type Opus = (Link_Attributes, ())
  -- |
read_Opus :: STM Result [Content i] Opus
read_Opus = do
  y ← read_ELEMENT "opus"
  y1 ← read_1 read_Link_Attributes (attributes y)
  return (y1, ())
  -- |
show_Opus :: Opus → [Content ()]
show_Opus (a, _) = show_ELEMENT "opus" (show_Link_Attributes a) []
  -- |
type Movement_Number = PCDATA
  -- |
read_Movement_Number :: STM Result [Content i] Movement_Number
read_Movement_Number = do
  y ← read_ELEMENT "movement-number"
  read_1 read_PCDATA (childs y)
  -- |
show_Movement_Number :: Movement_Number → [Content ()]
show_Movement_Number a = show_ELEMENT "movement-number" [] (show_PCDATA a)
  -- |
type Movement_Title = PCDATA
  -- |
read_Movement_Title :: STM Result [Content i] Movement_Title
read_Movement_Title = do
  y ← read_ELEMENT "movement-title"
  read_1 read_PCDATA (childs y)
  -- |
show_Movement_Title :: Movement_Title → [Content ()]
show_Movement_Title a = show_ELEMENT "movement-title" [] (show_PCDATA a)
```

Collect score-wide defaults. This includes scaling and layout, defined in layout.mod, and default values for the music font, word font, lyric font, and lyric language. The number and name attributes in lyric-font and lyric-language elements are typically used when lyrics are provided in multiple languages. If the number and name attributes are omitted, the lyric-font and lyric-language values apply to all numbers and names.

```
  -- * Defaults
  -- |
type Defaults = (Maybe Scaling, Maybe Page_Layout,
  Maybe System_Layout, [Staff_Layout], Maybe Appearance,
```

```
    Maybe Music_Font, Maybe Word_Font, [Lyric_Font], [Lyric_Language])
  -- |
read_Defaults :: Eq i ⇒ STM Result [Content i] Defaults
read_Defaults = do
  y ← read_ELEMENT "defaults"
  read_9 (read_MAYBE read_Scaling) (read_MAYBE read_Page_Layout)
    (read_MAYBE read_System_Layout)
    (read_LIST read_Staff_Layout) (read_MAYBE read_Appearance)
    (read_MAYBE read_Music_Font) (read_MAYBE read_Word_Font)
    (read_LIST read_Lyric_Font) (read_LIST read_Lyric_Language)
    (childs y)
  -- |
show_Defaults :: Defaults → [Content ()]
show_Defaults (a, b, c, d, e, f, g, h, i) =
  show_ELEMENT "defaults" []
    (show_MAYBE show_Scaling a ++ show_MAYBE show_Page_Layout b ++
      show_MAYBE show_System_Layout c ++
      show_LIST show_Staff_Layout d ++ show_MAYBE show_Appearance e ++
      show_MAYBE show_Music_Font f ++ show_MAYBE show_Word_Font g ++
      show_LIST show_Lyric_Font h ++ show_LIST show_Lyric_Language i)
  -- |
type Music_Font = (Font, ())
  -- |
read_Music_Font :: Eq i ⇒ STM Result [Content i] Music_Font
read_Music_Font = do
  y ← read_ELEMENT "music-font"
  y1 ← read_1 read_Font (attributes y)
  return (y1, ())
  -- |
show_Music_Font :: Music_Font → [Content ()]
show_Music_Font (a, _) =
  show_ELEMENT "music-font" (show_Font a) []
  -- |
type Word_Font = (Font, ())
  -- |
read_Word_Font :: Eq i ⇒ STM Result [Content i] Word_Font
read_Word_Font = do
  y ← read_ELEMENT "word-font"
  y1 ← read_1 read_Font (attributes y)
  return (y1, ())
  -- |
show_Word_Font :: Word_Font → [Content ()]
show_Word_Font (a, _) =
  show_ELEMENT "word-font" (show_Font a) []
  -- |
type Lyric_Font = ((Maybe CDATA, Maybe CDATA, Font), ())
  -- |
read_Lyric_Font :: Eq i ⇒ STM Result [Content i] Lyric_Font
read_Lyric_Font = do
  y ← read_ELEMENT "lyric-font"
  y1 ← read_3 (read_IMPLIED "number" read_CDATA)
    (read_IMPLIED "name" read_CDATA)
    read_Font (attributes y)
  return (y1, ())
  -- |
show_Lyric_Font :: Lyric_Font → [Content ()]
show_Lyric_Font ((a, b, c), _) =
  show_ELEMENT "lyric-font"
```

```
      (show_IMPLIED "number" show_CDATA a ++
        show_IMPLIED "name" show_CDATA b ++ show_Font c) []
  -- |
type Lyric_Language = ((Maybe CDATA, Maybe CDATA, CDATA), ())
  -- |
read_Lyric_Language :: Eq i => STM Result [Content i] Lyric_Language
read_Lyric_Language = do
  y <- read_ELEMENT "lyric-language"
  y1 <- read_3 (read_IMPLIED "number" read_CDATA)
    (read_IMPLIED "name" read_CDATA)
    (read_REQUIRED "xml:lang" read_CDATA) (attributes y)
  return (y1, ())
  -- |
show_Lyric_Language :: Lyric_Language -> [Content ()]
show_Lyric_Language ((a, b, c), _) =
  show_ELEMENT "lyric-language"
    (show_IMPLIED "number" show_CDATA a ++
      show_IMPLIED "name" show_CDATA b ++
      show_REQUIRED "xml:lang" show_CDATA c) []
```

Credit elements refer to the title, composer, arranger, lyricist, copyright, dedication, and other text that usually appears on the first page of a score. The credit-words and credit-image elements are similar to the words and image elements for directions. However, since the credit is not part of a measure, the default-x and default-y attributes adjust the origin relative to the bottom left-hand corner of the first page. The enclosure for credit-words is none by default.

By default, a series of credit-words elements within a single credit element follow one another in sequence visually. Non-positional formatting attributes are carried over from the previous element by default.

The page attribute for the credit element, new in Version 2.0, specifies the page number where the credit should appear. This is an integer value that starts with 1 for the first page. Its value is 1 by default. Since credits occur before the music, these page numbers do not refer to the page numbering specified by the print element's page-number attribute.

In the initial release of Version 2.0, the credit element had a non-deterministic definition. The current credit element definition has the same meaning, but avoids the validity errors arising from a non-deterministic definition.

```
  -- * Credit
  -- |
type Credit = (Maybe CDATA, ([Link], [Bookmark], Credit_))
  -- |
read_Credit :: Eq i => STM Result [Content i] Credit
read_Credit = do
  y <- read_ELEMENT "credit"
  y1 <- read_1 (read_IMPLIED "page" read_CDATA) (attributes y)
  y2 <- read_3 (read_LIST read_Link) (read_LIST read_Bookmark)
    read_Credit_ (childs y)
  return (y1, y2)
  -- |
show_Credit :: Credit -> [Content ()]
show_Credit (a, (b, c, d)) =
  show_ELEMENT "credit"
    (show_IMPLIED "page" show_CDATA a)
    (show_LIST show_Link b ++ show_LIST show_Bookmark c ++ show_Credit_ d)
  -- |
data Credit_ = Credit_1 Credit_Image
  | Credit_2 (Credit_Words, [([Link], [Bookmark], Credit_Words)])
    deriving (Eq, Show)
  -- |
read_Credit_ :: Eq i => STM Result [Content i] Credit_
```

```
read_Credit_ =
  (read_Credit_Image >>= return . Credit_1) `mplus`
  (read_Credit_aux1 >>= return . Credit_2)
  -- |
show_Credit_ :: Credit_ -> [Content ()]
show_Credit_ (Credit_1 a) = show_Credit_Image a
show_Credit_ (Credit_2 (a, b)) =
  show_Credit_Words a ++ show_LIST show_Credit_aux1 b
  -- |
read_Credit_aux1 :: Eq i => STM Result [Content i]
  (Credit_Words, [([Link], [Bookmark], Credit_Words)])
read_Credit_aux1 = do
  y1 <- read_Credit_Words
  y2 <- read_LIST read_Credit_aux2
  return (y1, y2)
  -- |
read_Credit_aux2 :: Eq i => STM Result [Content i] ([Link], [Bookmark], Credit_Words)
read_Credit_aux2 = do
  y1 <- read_LIST read_Link
  y2 <- read_LIST read_Bookmark
  y3 <- read_Credit_Words
  return (y1, y2, y3)
  -- |
show_Credit_aux1 :: ([Link], [Bookmark], Credit_Words) -> [Content ()]
show_Credit_aux1 (a, b, c) =
  show_LIST show_Link a ++ show_LIST show_Bookmark b ++
  show_Credit_Words c
  -- |
type Credit_Words = (Text_Formatting, PCDATA)
  -- |
read_Credit_Words :: STM Result [Content i] Credit_Words
read_Credit_Words = do
  y <- read_ELEMENT "credit-words"
  y1 <- read_1 read_Text_Formatting (attributes y)
  y2 <- read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Credit_Words :: Credit_Words -> [Content ()]
show_Credit_Words (a, b) =
  show_ELEMENT "credit-words" (show_Text_Formatting a) (show_PCDATA b)
  -- |
type Credit_Image = ((CDATA, CDATA,
    Position, Halign, Valign_Image), ())
  -- |
read_Credit_Image :: STM Result [Content i] Credit_Image
read_Credit_Image = do
  y <- read_ELEMENT "credit-image"
  y1 <- read_5 (read_REQUIRED "source" read_CDATA)
    (read_REQUIRED "type" read_CDATA) read_Position
    read_Halign read_Valign_Image (attributes y)
  return (y1, ())
  -- |
show_Credit_Image :: Credit_Image -> [Content ()]
show_Credit_Image ((a, b, c, d, e), _) =
  show_ELEMENT "credit-image"
    (show_REQUIRED "source" show_CDATA a ++
      show_REQUIRED "type" show_CDATA b ++
      show_Position c ++ show_Halign d ++
```

```
      show_Valign_Image e) []
  -- |
```

The part-list identifies the different musical parts in this movement. Each part has an ID that is used later within the musical data. Since parts may be encoded separately and combined later, identification elements are present at both the score and score-part levels. There must be at least one score-part, combined as desired with part-group elements that indicate braces and brackets. Parts are ordered from top to bottom in a score based on the order in which they appear in the part-list.

Each MusicXML part corresponds to a track in a Standard MIDI Format 1 file. The score-instrument elements are used when there are multiple instruments per track. The midi-device element is used to make a MIDI device or port assignment for the given track. Initial midi-instrument assignments may be made here as well.

The part-name and part-abbreviation elements are defined in the common.mod file, as they can be used within both the part-list and print elements.

```
  -- * Part_List
  -- |
type Part_List = ([Part_Group], Score_Part, [Part_List_])
  -- |
read_Part_List :: Eq i ⇒ STM Result [Content i] Part_List
read_Part_List = do
  y ← read_ELEMENT "part-list"
  read_3 (read_LIST read_Part_Group) read_Score_Part
    (read_LIST read_Part_List_) (childs y)
  -- |
show_Part_List :: Part_List → [Content ()]
show_Part_List (a, b, c) =
  show_ELEMENT "part-list" []
    (show_LIST show_Part_Group a ++ show_Score_Part b ++
      show_LIST show_Part_List_ c)
  -- |
data Part_List_ = Part_List_1 Part_Group
  | Part_List_2 Score_Part
    deriving (Eq, Show)
  -- |
read_Part_List_ :: Eq i ⇒ STM Result [Content i] Part_List_
read_Part_List_ =
  (read_Part_Group >>= return · Part_List_1) `mplus`
  (read_Score_Part >>= return · Part_List_2)
  -- |
show_Part_List_ :: Part_List_ → [Content ()]
show_Part_List_ (Part_List_1 a) = show_Part_Group a
show_Part_List_ (Part_List_2 a) = show_Score_Part a
  -- |
type Score_Part = (ID, (Maybe Identification,
  Part_Name, Maybe Part_Name_Display,
  Maybe Part_Abbreviation, Maybe Part_Abbreviation_Display,
  [Group], [Score_Instrument], Maybe Midi_Device, [Midi_Instrument]))
  -- |
read_Score_Part :: Eq i ⇒ STM Result [Content i] Score_Part
read_Score_Part = do
  y ← read_ELEMENT "score-part"
  y1 ← read_1 (read_REQUIRED "id" read_ID) (attributes y)
  y2 ← read_9 (read_MAYBE read_Identification) read_Part_Name
    (read_MAYBE read_Part_Name_Display)
    (read_MAYBE read_Part_Abbreviation)
    (read_MAYBE read_Part_Abbreviation_Display)
    (read_LIST read_Group)
    (read_LIST read_Score_Instrument)
```

```
    (read_MAYBE read_Midi_Device)
    (read_LIST read_Midi_Instrument) (childs y)
  return (y1, y2)
  -- |
show_Score_Part :: Score_Part → [Content ()]
show_Score_Part (a, (b, c, d, e, f, g, h, i, j)) =
  show_ELEMENT "score-part" (show_REQUIRED "id" show_ID a)
    (show_MAYBE show_Identification b ++
      show_Part_Name c ++
      show_MAYBE show_Part_Name_Display d ++
      show_MAYBE show_Part_Abbreviation e ++
      show_MAYBE show_Part_Abbreviation_Display f ++
      show_LIST show_Group g ++
      show_LIST show_Score_Instrument h ++
      show_MAYBE show_Midi_Device i ++
      show_LIST show_Midi_Instrument j)
```

The part-name indicates the full name of the musical part. The part-abbreviation indicates the abbreviated version of the name of the musical part. The part-name will often precede the first system, while the part-abbreviation will precede the other systems. The formatting attributes for these elements are deprecated in Version 2.0 in favor of the new part-name-display and part-abbreviation-display elements. These are defined in the common.mod file as they are used in both the part-list and print elements. They provide more complete formatting control for how part names and abbreviations appear in a score.

```
  -- |
type Part_Name = ((Print_Style, Print_Object, Justify), PCDATA)
  -- |
read_Part_Name :: STM Result [Content i] Part_Name
read_Part_Name = do
  y ← read_ELEMENT "part-name"
  y1 ← read_3 read_Print_Style read_Print_Object read_Justify
    (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Part_Name :: Part_Name → [Content ()]
show_Part_Name ((a, b, c), d) =
  show_ELEMENT "part-name"
    (show_Print_Style a ++ show_Print_Object b ++ show_Justify c)
    (show_PCDATA d)
  -- |
type Part_Abbreviation = ((Print_Style, Print_Object, Justify), PCDATA)
  -- |
read_Part_Abbreviation :: STM Result [Content i] Part_Abbreviation
read_Part_Abbreviation = do
  y ← read_ELEMENT "part-abbreviation"
  y1 ← read_3 read_Print_Style read_Print_Object read_Justify
    (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Part_Abbreviation :: Part_Abbreviation → [Content ()]
show_Part_Abbreviation ((a, b, c), d) =
  show_ELEMENT "part-abbreviation"
    (show_Print_Style a ++ show_Print_Object b ++ show_Justify c)
    (show_PCDATA d)
```

The part-group element indicates groupings of parts in the score, usually indicated by braces and brackets. Braces that are used for multi-staff parts should be defined in the attributes element for that

part. The part-group start element appears before the first score-part in the group. The part-group stop element appears after the last score-part in the group.

The number attribute is used to distinguish overlapping and nested part-groups, not the sequence of groups. As with parts, groups can have a name and abbreviation. Formatting attributes for group-name and group-abbreviation are deprecated in Version 2.0 in favor of the new group-name-display and group-abbreviation-display elements. Formatting specified in the group-name-display and group-abbreviation-display elements overrides formatting specified in the group-name and group-abbreviation elements, respectively.

The group-symbol element indicates how the symbol for a group is indicated in the score. Values include none, brace, line, and bracket; the default is none. The group-barline element indicates if the group should have common barlines. Values can be yes, no, or Mensurstrich. The group-time element indicates that the displayed time signatures should stretch across all parts and staves in the group. Values for the child elements are ignored at the stop of a group.

A part-group element is not needed for a single multi-staff part. By default, multi-staff parts include a brace symbol and (if appropriate given the bar-style) common barlines. The symbol formatting for a multi-staff part can be more fully specified using the part-symbol element, defined in the attributes.mod file.

```
  -- |
type Part_Group = ((Start_Stop, CDATA),
  (Maybe Group_Name, Maybe Group_Name_Display,
  Maybe Group_Abbreviation, Maybe Group_Abbreviation_Display,
  Maybe Group_Symbol, Maybe Group_Barline, Maybe Group_Time, Editorial))
  -- |
read_Part_Group :: Eq i ⇒ STM Result [Content i] Part_Group
read_Part_Group = do
  y ← read_ELEMENT "part-group"
  y1 ← read_2 (read_REQUIRED "type" read_Start_Stop)
    (read_DEFAULT "number" read_CDATA "1") (attributes y)
  y2 ← read_8 (read_MAYBE read_Group_Name)
    (read_MAYBE read_Group_Name_Display)
    (read_MAYBE read_Group_Abbreviation)
    (read_MAYBE read_Group_Abbreviation_Display)
    (read_MAYBE read_Group_Symbol) (read_MAYBE read_Group_Barline)
    (read_MAYBE read_Group_Time) read_Editorial (childs y)
  return (y1, y2)
  -- |
show_Part_Group :: Part_Group → [Content ()]
show_Part_Group ((a, b), (c, d, e, f, g, h, i, j)) =
  show_ELEMENT "part-group"
    (show_REQUIRED "type" show_Start_Stop a ++
      show_DEFAULT "number" show_CDATA b)
    (show_MAYBE show_Group_Name c ++
      show_MAYBE show_Group_Name_Display d ++
      show_MAYBE show_Group_Abbreviation e ++
      show_MAYBE show_Group_Abbreviation_Display f ++
      show_MAYBE show_Group_Symbol g ++
      show_MAYBE show_Group_Barline h ++
      show_MAYBE show_Group_Time i ++ show_Editorial j)
  -- |
type Group_Name = ((Print_Style, Justify), PCDATA)
  -- |
read_Group_Name :: STM Result [Content i] Group_Name
read_Group_Name = do
  y ← read_ELEMENT "group-name"
  y1 ← read_2 read_Print_Style read_Justify (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
```

```
show_Group_Name :: Group_Name → [Content ()]
show_Group_Name ((a, b), c) =
  show_ELEMENT "group-name"
    (show_Print_Style a ++ show_Justify b)
    (show_PCDATA c)
  -- |
type Group_Name_Display = (Print_Object, [Group_Name_Display_])
  -- |
read_Group_Name_Display :: Eq i ⇒ STM Result [Content i] Group_Name_Display
read_Group_Name_Display = do
  y ← read_ELEMENT "group-name-display"
  y1 ← read_1 read_Print_Object (attributes y)
  y2 ← read_1 (read_LIST read_Group_Name_Display_) (childs y)
  return (y1, y2)
  -- |
show_Group_Name_Display :: Group_Name_Display → [Content ()]
show_Group_Name_Display (a, b) =
  show_ELEMENT "group-name-display"
    (show_Print_Object a) (show_LIST show_Group_Name_Display_ b)
  -- |
data Group_Name_Display_ = Group_Name_Display_1 Display_Text
  | Group_Name_Display_2 Accidental_Text
    deriving (Eq, Show)
  -- |
read_Group_Name_Display_ :: STM Result [Content i] Group_Name_Display_
read_Group_Name_Display_ =
  (read_Display_Text ≫= return · Group_Name_Display_1) `mplus`
  (read_Accidental_Text ≫= return · Group_Name_Display_2)
  -- |
show_Group_Name_Display_ :: Group_Name_Display_ → [Content ()]
show_Group_Name_Display_ (Group_Name_Display_1 a) = show_Display_Text a
show_Group_Name_Display_ (Group_Name_Display_2 a) = show_Accidental_Text a
  -- |
type Group_Abbreviation = ((Print_Style, Justify), PCDATA)
  -- |
read_Group_Abbreviation :: STM Result [Content i] Group_Abbreviation
read_Group_Abbreviation = do
  y ← read_ELEMENT "group-abbreviation"
  y1 ← read_2 read_Print_Style read_Justify (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Group_Abbreviation :: Group_Abbreviation → [Content ()]
show_Group_Abbreviation ((a, b), c) =
  show_ELEMENT "group-abbreviation"
    (show_Print_Style a ++ show_Justify b)
    (show_PCDATA c)
  -- |
type Group_Abbreviation_Display = (Print_Object, [Group_Abbreviation_Display_])
  -- |
read_Group_Abbreviation_Display :: Eq i ⇒
  STM Result [Content i] Group_Abbreviation_Display
read_Group_Abbreviation_Display = do
  y ← read_ELEMENT "group-abbreviation-display"
  y1 ← read_1 read_Print_Object (attributes y)
  y2 ← read_1 (read_LIST read_Group_Abbreviation_Display_) (childs y)
  return (y1, y2)
  -- |
```

```
show_Group_Abbreviation_Display ::
  Group_Abbreviation_Display → [Content ()]
show_Group_Abbreviation_Display (a, b) =
  show_ELEMENT "group-abbreviation-display"
    (show_Print_Object a) (show_LIST show_Group_Abbreviation_Display_ b)
  -- |
data Group_Abbreviation_Display_ =
    Group_Abbreviation_Display_1 Display_Text
  | Group_Abbreviation_Display_2 Accidental_Text
    deriving (Eq, Show)
  -- |
read_Group_Abbreviation_Display_ ::
  STM Result [Content i] Group_Abbreviation_Display_
read_Group_Abbreviation_Display_ =
  (read_Display_Text >>= return · Group_Abbreviation_Display_1) `mplus`
  (read_Accidental_Text >>= return · Group_Abbreviation_Display_2)
  -- |
show_Group_Abbreviation_Display_ ::
  Group_Abbreviation_Display_ → [Content ()]
show_Group_Abbreviation_Display_ (Group_Abbreviation_Display_1 a) =
  show_Display_Text a
show_Group_Abbreviation_Display_ (Group_Abbreviation_Display_2 a) =
  show_Accidental_Text a
  -- |
type Group_Symbol = ((Position, Color), PCDATA)
  -- |
read_Group_Symbol :: STM Result [Content i] Group_Symbol
read_Group_Symbol = do
  y ← read_ELEMENT "group-symbol"
  y1 ← read_2 read_Position read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Group_Symbol :: Group_Symbol → [Content ()]
show_Group_Symbol ((a, b), c) =
  show_ELEMENT "group-symbol"
    (show_Position a ++ show_Color b) (show_PCDATA c)
  -- |
type Group_Barline = (Color, PCDATA)
  -- |
read_Group_Barline :: STM Result [Content i] Group_Barline
read_Group_Barline = do
  y ← read_ELEMENT "group-barline"
  y1 ← read_1 read_Color (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Group_Barline :: Group_Barline → [Content ()]
show_Group_Barline (a, b) =
  show_ELEMENT "group-barline" (show_Color a) (show_PCDATA b)
  -- |
type Group_Time = ()
  -- |
read_Group_Time :: STM Result [Content i] Group_Time
read_Group_Time = read_ELEMENT "group-time" >> return ()
  -- |
show_Group_Time :: Group_Time → [Content ()]
show_Group_Time _ = show_ELEMENT "group-time" [] []
```

The score-instrument element allows for multiple instruments per score-part. As with the score-part element, each score-instrument has a required ID attribute, a name, and an optional abbreviation. The instrument-name and instrument-abbreviation are typically used within a software application, rather than appearing on the printed page of a score.

A score-instrument element is also required if the score specifies MIDI 1.0 channels, banks, or programs. An initial midi-instrument assignment can also be made here. MusicXML software should be able to automatically assign reasonable channels and instruments without these elements in simple cases, such as where part names match General MIDI instrument names.

The solo and ensemble elements are new as of Version 2.0. The solo element is present if performance is intended by a solo instrument. The ensemble element is present if performance is intended by an ensemble such as an orchestral section. The text of the ensemble element contains the size of the section, or is empty if the ensemble size is not specified.

The midi-instrument element is defined in the common.mod file, as it can be used within both the score-part and sound elements.

```
  -- |
type Score_Instrument = (ID, (Instrument_Name, Maybe Instrument_Abbreviation,
  Maybe Score_Instrument_))
  -- |
read_Score_Instrument :: STM Result [Content i] Score_Instrument
read_Score_Instrument = do
  y ← read_ELEMENT "score-instrument"
  y1 ← read_1 (read_REQUIRED "id" read_ID) (attributes y)
  y2 ← read_3 read_Instrument_Name
    (read_MAYBE read_Instrument_Abbreviation)
    (read_MAYBE read_Score_Instrument_) (childs y)
  return (y1, y2)
  -- |
show_Score_Instrument :: Score_Instrument → [Content ()]
show_Score_Instrument (a, (b, c, d)) =
  show_ELEMENT "score-instrument" (show_REQUIRED "id" show_ID a)
    (show_Instrument_Name b ++
      show_MAYBE show_Instrument_Abbreviation c ++
      show_MAYBE show_Score_Instrument_ d)
  -- |
data Score_Instrument_ = Score_Instrument_1 Solo
  | Score_Instrument_2 Ensemble
    deriving (Eq, Show)
  -- |
read_Score_Instrument_ :: STM Result [Content i] Score_Instrument_
read_Score_Instrument_ =
  (read_Solo >>= return · Score_Instrument_1) `mplus`
  (read_Ensemble >>= return · Score_Instrument_2)
  -- |
show_Score_Instrument_ :: Score_Instrument_ → [Content ()]
show_Score_Instrument_ (Score_Instrument_1 a) = show_Solo a
show_Score_Instrument_ (Score_Instrument_2 a) = show_Ensemble a
  -- |
type Instrument_Name = PCDATA
  -- |
read_Instrument_Name :: STM Result [Content i] Instrument_Name
read_Instrument_Name = do
  y ← read_ELEMENT "instrument-name"
  read_1 read_PCDATA (childs y)
  -- |
show_Instrument_Name :: Instrument_Name → [Content ()]
show_Instrument_Name a = show_ELEMENT "instrument-name" [] (show_PCDATA a)
  -- |
type Instrument_Abbreviation = PCDATA
```

```
  -- |
read_Instrument_Abbreviation :: STM Result [Content i] Instrument_Abbreviation
read_Instrument_Abbreviation = do
  y ← read_ELEMENT "instrument-abbreviation"
  read_1 read_PCDATA (childs y)
  -- |
show_Instrument_Abbreviation :: Instrument_Abbreviation → [Content ()]
show_Instrument_Abbreviation a =
  show_ELEMENT "instrument-abbreviation" [] (show_PCDATA a)
  -- |
type Solo = ()
  -- |
read_Solo :: STM Result [Content i] Solo
read_Solo = read_ELEMENT "solo" ≫ return ()
  -- |
show_Solo :: Solo → [Content ()]
show_Solo _ = show_ELEMENT "solo" [] []
  -- |
type Ensemble = PCDATA
  -- |
read_Ensemble :: STM Result [Content i] Ensemble
read_Ensemble = do
  y ← read_ELEMENT "ensemble"
  read_1 read_PCDATA (childs y)
  -- |
show_Ensemble :: Ensemble → [Content ()]
show_Ensemble a = show_ELEMENT "ensemble" [] (show_PCDATA a)
```

The midi-device content corresponds to the DeviceName meta event in Standard MIDI Files. The optional port attribute is a number from 1 to 16 that can be used with the unofficial MIDI port (or cable) meta event.

```
  -- |
type Midi_Device = (Maybe CDATA, PCDATA)
  -- |
read_Midi_Device :: STM Result [Content i] Midi_Device
read_Midi_Device = do
  y ← read_ELEMENT "midi-device"
  y1 ← read_1 (read_IMPLIED "port" read_CDATA) (attributes y)
  y2 ← read_1 read_PCDATA (childs y)
  return (y1, y2)
  -- |
show_Midi_Device :: Midi_Device → [Content ()]
show_Midi_Device (a, b) =
  show_ELEMENT "midi-device" (show_IMPLIED "port" show_CDATA a)
    (show_PCDATA b)
```

The group element allows the use of different versions of the part for different purposes. Typical values include score, parts, sound, and data. Ordering information that is directly encoded in MuseData can be derived from the ordering within a MusicXML score or opus.

```
  -- |
type Group = PCDATA
  -- |
read_Group :: STM Result [Content i] Group
read_Group = do
  y ← read_ELEMENT "group"
  read_1 read_PCDATA (childs y)
  -- |
```

```
show_Group :: Group → [Content ()]
show_Group a = show_ELEMENT "group" [] (show_PCDATA a)
```

Here is the basic musical data that is either associated with a part or a measure, depending on whether partwise or timewise hierarchy is used.

```
  -- * Music_Data
  -- |
type Music_Data = [Music_Data_]
  -- |
read_Music_Data :: Eq i ⇒ STM Result [Content i] Music_Data
read_Music_Data = read_LIST read_Music_Data_
  -- |
show_Music_Data :: Music_Data → [Content ()]
show_Music_Data x = show_LIST show_Music_Data_ x
  -- |
data Music_Data_ = Music_Data_1 Note
   | Music_Data_2 Backup
   | Music_Data_3 Forward
   | Music_Data_4 Direction
   | Music_Data_5 Attributes
   | Music_Data_6 Harmony
   | Music_Data_7 Figured_Bass
   | Music_Data_8 Print
   | Music_Data_9 Sound
   | Music_Data_10 Barline
   | Music_Data_11 Grouping
   | Music_Data_12 Link
   | Music_Data_13 Bookmark
  deriving (Eq, Show)
  -- |
read_Music_Data_ :: Eq i ⇒ STM Result [Content i] Music_Data_
read_Music_Data_ =
  (read_Note ≫= return · Music_Data_1) `mplus`
  (read_Backup ≫= return · Music_Data_2) `mplus`
  (read_Forward ≫= return · Music_Data_3) `mplus`
  (read_Direction ≫= return · Music_Data_4) `mplus`
  (read_Attributes ≫= return · Music_Data_5) `mplus`
  (read_Harmony ≫= return · Music_Data_6) `mplus`
  (read_Figured_Bass ≫= return · Music_Data_7) `mplus`
  (read_Print ≫= return · Music_Data_8) `mplus`
  (read_Sound ≫= return · Music_Data_9) `mplus`
  (read_Barline ≫= return · Music_Data_10) `mplus`
  (read_Grouping ≫= return · Music_Data_11) `mplus`
  (read_Link ≫= return · Music_Data_12) `mplus`
  (read_Bookmark ≫= return · Music_Data_13)
  -- |
show_Music_Data_ :: Music_Data_ → [Content ()]
show_Music_Data_ (Music_Data_1 x) = show_Note x
show_Music_Data_ (Music_Data_2 x) = show_Backup x
show_Music_Data_ (Music_Data_3 x) = show_Forward x
show_Music_Data_ (Music_Data_4 x) = show_Direction x
show_Music_Data_ (Music_Data_5 x) = show_Attributes x
show_Music_Data_ (Music_Data_6 x) = show_Harmony x
show_Music_Data_ (Music_Data_7 x) = show_Figured_Bass x
show_Music_Data_ (Music_Data_8 x) = show_Print x
show_Music_Data_ (Music_Data_9 x) = show_Sound x
show_Music_Data_ (Music_Data_10 x) = show_Barline x
show_Music_Data_ (Music_Data_11 x) = show_Grouping x
```

```
show_Music_Data_ (Music_Data_12 x) = show_Link x
show_Music_Data_ (Music_Data_13 x) = show_Bookmark x
```

The score-header entity contains basic score metadata about the work and movement, score-wide defaults for layout and fonts, credits that appear on the first page, and the part list.

```
  -- * Score_Header
  -- |
type Score_Header = (Maybe Work, Maybe Movement_Number,
  Maybe Movement_Title, Maybe Identification,
  Maybe Defaults, [Credit], Part_List)
  -- |
read_Score_Header :: Eq i ⇒ STM Result [Content i] Score_Header
read_Score_Header = do
  y1 ← read_MAYBE read_Work
  y2 ← read_MAYBE read_Movement_Number
  y3 ← read_MAYBE read_Movement_Title
  y4 ← read_MAYBE read_Identification
  y5 ← read_MAYBE read_Defaults
  y6 ← read_LIST read_Credit
  y7 ← read_Part_List
  return (y1, y2, y3, y4, y5, y6, y7)
  -- |
show_Score_Header :: Score_Header → [Content ()]
show_Score_Header (a, b, c, d, e, f, g) =
  show_MAYBE show_Work a ++
  show_MAYBE show_Movement_Number b ++
  show_MAYBE show_Movement_Title c ++
  show_MAYBE show_Identification d ++
  show_MAYBE show_Defaults e ++
  show_LIST show_Credit f ++
  show_Part_List g
  -- |
update_Score_Header :: ([Software], Encoding_Date) → Score_Header → Score_Header
update_Score_Header x (a, b, c, d, e, f, g) =
  (a, b, c, fmap (update_Identification x) d, e, f, g)
```

2.14 Timewise

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Timewise where
import Text.XML.MusicXML.Common
import Text.XML.MusicXML.Identity
import Text.XML.MusicXML.Score
import Text.XML.HaXml.Types (Content,
  DocTypeDecl (..), ExternalID (..), PubidLiteral (..), SystemLiteral (..))
import Prelude (Maybe (..), Monad (..), Eq, (++))
```

The MusicXML format is designed to represent musical scores, specifically common western musical notation from the 17th century onwards. It is designed as an interchange format for notation, analysis, retrieval, and performance applications. Therefore it is intended to be sufficient, not optimal, for these applications.

The MusicXML format is based on the MuseData and Humdrum formats. Humdrum explicitly represents the two-dimensional nature of musical scores by a 2-D layout notation. Since the XML format is hierarchical, we cannot do this explicitly. Instead, there are two top-level formats:

partwise.dtd Represents scores by part/instrument timewise.dtd Represents scores by time/measure

Thus partwise.dtd contains measures within each part, while timewise.dtd contains parts within each measure. XSLT stylesheets are provided to convert between the two formats.

The partwise and timewise score DTDs represent a single movement of music. Multiple movements or other musical collections are presented using opus.dtd. An opus document contains XLinks to individual scores.

Suggested use:

```
<!DOCTYPE score-timewise PUBLIC
        "-//Recordare//DTD MusicXML 2.0 Timewise//EN"
        "http://www.musicxml.org/dtds/timewise.dtd">
```

This DTD is made up of a series of component DTD modules, all of which are included here.

```
  -- |
doctype :: DocTypeDecl
doctype = DTD "score-timewise"
  (Just (PUBLIC (PubidLiteral "-//Recordare//DTD MusicXML 2.0 Timewise//EN")
    (SystemLiteral "http://www.musicxml.org/dtds/timewise.dtd")))
    []
```

The score is the root element for the DTD. It includes the score-header entity, followed either by a series of parts with measures inside (score-partwise) or a series of measures with parts inside (score-timewise). Having distinct top-level elements for partwise and timewise scores makes it easy to ensure that an XSLT stylesheet does not try to transform a document already in the desired format. The document-attributes entity includes the version attribute and is defined in the common.mod file.

In either format, the part element has an id attribute that is an IDREF back to a score-part in the part-list. Measures have a required number attribute (going from partwise to timewise, measures are grouped via the number).

The implicit attribute is set to "yes" for measures where the measure number should never appear, such as pickup measures and the last half of mid-measure repeats. The value is "no" if not specified.

The non-controlling attribute is intended for use in multimetric music like the Don Giovanni minuet. If set to "yes", the left barline in this measure does not coincide with the left barline of measures in other parts. The value is "no" if not specified.

In partwise files, the number attribute should be the same for measures in different parts that share the same left barline. While the number attribute is often numeric, it does not have to be. Non-numeric values are typically used together with the implicit or non-controlling attributes being set to "yes". For a pickup measure, the number attribute is typically set to "0" and the implicit attribute is typically set to "yes". Further details about measure numbering can be defined using the measure-numbering element defined in the direction.mod file

Measure width is specified in tenths. These are the global tenths specified in the scaling element, not local tenths as modified by the staff-size element.

```
  -- * Score_Timewise
  -- |
type Score_Timewise = (Document_Attributes, (Score_Header, [Measure]))
  -- |
read_Score_Timewise :: Eq i ⇒ STM Result [Content i] Score_Timewise
read_Score_Timewise = do
  y ← read_ELEMENT "score-timewise"
  y1 ← read_1 read_Document_Attributes (attributes y)
  y2 ← read_2 read_Score_Header (read_LIST1 read_Measure) (childs y)
  return (y1, y2)
  -- |
show_Score_Timewise :: Score_Timewise → [Content ()]
show_Score_Timewise (a, (b, c)) =
  show_ELEMENT "score-timewise" (show_Document_Attributes a)
```

```
      (show_Score_Header b ++
        show_LIST1 show_Measure c)
  -- |
update_Score_Timewise :: ([Software], Encoding_Date) ->
  Score_Timewise -> Score_Timewise
update_Score_Timewise x (a,(b,c)) = (a,(update_Score_Header x b,c))
  -- |
type Measure = ((CDATA, Maybe Yes_No, Maybe Yes_No, Maybe Tenths),[Part])
  -- |
read_Measure :: Eq i => STM Result [Content i] Measure
read_Measure = do
  y <- read_ELEMENT "measure"
  y1 <- read_4 (read_REQUIRED "number" read_CDATA)
    (read_IMPLIED "implicit" read_Yes_No)
    (read_IMPLIED "non-controlling" read_Yes_No)
    (read_IMPLIED "width" read_Tenths)
    (attributes y)
  y2 <- read_1 (read_LIST1 read_Part) (childs y)
  return (y1,y2)
  -- |
show_Measure :: Measure -> [Content ()]
show_Measure ((a,b,c,d),e) =
  show_ELEMENT "measure" (show_REQUIRED "number" show_CDATA a ++
    show_IMPLIED "implicit" show_Yes_No b ++
    show_IMPLIED "non-controlling" show_Yes_No c ++
    show_IMPLIED "width" show_Tenths d)
    (show_LIST1 show_Part e)
  -- |
type Part = (ID, Music_Data)
  -- |
read_Part :: Eq i => STM Result [Content i] Part
read_Part = do
  y <- read_ELEMENT "part"
  y1 <- read_1 (read_REQUIRED "id" read_ID) (attributes y)
  y2 <- read_1 read_Music_Data (childs y)
  return (y1,y2)
  -- |
show_Part :: Part -> [Content ()]
show_Part (a,b) =
  show_ELEMENT "part" (show_REQUIRED "id" show_ID a)
    (show_Music_Data b)
```

2.15 Util

```
  -- |
  -- Maintainer : silva.samuel@alumni.uminho.pt
  -- Stability : experimental
  -- Portability: HaXML
  --
module Text.XML.MusicXML.Util where
import Text.XML.HaXml.Types
  -- (Attribute,AttValue(..),
  -- Element(..), Content(..))
import Control.Exception (throw, Exception (..))
import Control.Monad (MonadPlus (..))
```

```
import Data.Char (isSpace)
import Prelude (String, Maybe (..), (..) + ·, Bool (..),
  Monad (..), Show (..), Int, Functor (..), Eq (..),
  (·), (++), (∧),
  id, map, concat, [·, ·], maybe, and,
  _, lookup, unwords)

  -- * Result
  -- |
data Result a = Ok a | Error String
  deriving (Eq, Show)
  -- |
instance Monad Result where
  (Ok a) >>= b = b a
  (Error msg) >>= _ = Error msg
  return x = Ok x
  fail msg = Error msg
  -- |
instance Functor Result where
  fmap f (Ok a) = Ok (f a)
  fmap _ (Error msg) = Error msg
  -- |
instance MonadPlus Result where
  mzero = Error "unknow error"
  (Ok a) `mplus` _ = (Ok a)
  (Error _) `mplus` b = b
  -- |
isOK :: Result a → Bool
isOK (Ok _) = True
isOK _ = False
  -- |
isError :: Result a → Bool
isError (Error _) = True
isError _ = False
  -- |
fromOK :: Result a → a
fromOK (Ok a) = a
fromOK (Error msg) = throw (ErrorCall msg)
  -- |
fromError :: Result a → String
fromError (Ok _) = []
fromError (Error msg) = msg
  -- * ST
  -- |
newtype ST s a = ST{state :: s → (s, a)}
instance Monad (ST s) where
  return x = ST (λs → (s, x))
  p >>= f = ST (λs1 → let (s2, r) = state p s1 in state (f r) s2)
instance Functor (ST s) where
  fmap f st = ST (λs → (λ(x, y) → (x, f y)) (state st s))
  -- |
liftST :: (s → a) → ST s a
liftST f = ST (λs → (s, f s))
  -- * STM
  -- |
newtype STM m s a = STM{stateM :: s → m (s, a)}
  -- |
instance (Monad m) ⇒ Monad (STM m s) where
```

```
  return x = STM (λs → return (s, x))
  p >>= f = STM (λs → do {
    ; (s', l) ← stateM p s
    ; stateM (f l) s' })
  fail msg = STM (λ_ → fail msg)
  -- |
instance MonadPlus m ⇒ MonadPlus (STM m s) where
  mzero = STM (λ_ → mzero)
  a `mplus` b = STM (λs → (stateM a s) `mplus` (stateM b s))
  -- |
instance Monad m ⇒ Functor (STM m s) where
  fmap f stm = STM (λs → stateM stm s >>= (λ(s1, a) → return (s1, f a)))
  -- |
liftSTM :: Monad m ⇒ ST s (m a) → STM m s a
liftSTM p = STM (λs → do {
  ; let (s', l) = (state p s)
  ; lx ← l
  ; return (s', lx) })
  -- |
returnSTM :: Monad m ⇒ m a → STM m s a
returnSTM x = STM (λs → x >>= (λy → return (s, y)))


  -- * Basic
  -- |
type CDATA = Prelude.String
  -- |
read_CDATA :: Prelude.String → Result CDATA
read_CDATA = return
  -- |
show_CDATA :: CDATA → Prelude.String
show_CDATA = id
  -- |
type ID = Prelude.String
  -- |
read_ID :: Prelude.String → Result ID
read_ID = return
  -- |
show_ID :: ID → Prelude.String
show_ID = id


  -- * Attributes
  -- |
read_IMPLIED' :: String → (String → Result a) → [Attribute] → Maybe a
read_IMPLIED' key func s = maybe Nothing
  (result · func · unwords ·
    map ([id, "__"]) · (λ(AttValue l) → l))
  (lookup key s)
  where    -- |
    result :: Result a → Maybe a
    result (Ok x) = Just x
    result (Error _) = Nothing
  -- |
read_IMPLIED :: Monad m ⇒
  String → (String → Result a) → STM m [Attribute] (Maybe a)
read_IMPLIED key func =
  STM (λs → return (s, read_IMPLIED' key func s))
  -- |
show_IMPLIED :: String → (a → String) → Maybe a → [Attribute]
```

```
show_IMPLIED key function = maybe [] (show_REQUIRED key function)
  -- |
read_REQUIRED :: Monad m ⇒ String → (String → Result a) → STM m [Attribute] a
read_REQUIRED key func =
  read_IMPLIED key func ⋙=
  maybe (fail ("I expect " ++ key ++ " as required attribute")) return
  -- |
show_REQUIRED :: String → (a → String) → a → [Attribute]
show_REQUIRED key function =
  (:[]) · (λx → (key, x)) · AttValue · (:[]) · i₁ · function
  -- |
read_DEFAULT :: Monad m ⇒
  String → (String → Result a) → a → STM m [Attribute] a
read_DEFAULT key func def =
  read_IMPLIED key func ⋙=
  maybe (return def) return
  -- |
show_DEFAULT :: String → (a → String) → a → [Attribute]
show_DEFAULT = show_REQUIRED
  -- |
show_FIXED :: String → (a → String) → a → [Attribute]
show_FIXED = show_REQUIRED
  -- |
read_FIXED :: Monad m ⇒
  String → (String → Result a) → a → STM m [Attribute] a
read_FIXED key func def =
  read_IMPLIED key func ⋙=
  maybe (return def) return


  -- |
read_ELEMENT' :: String → [Content i] → ([Content i], Result (Element i))
read_ELEMENT' tag ((CElem (e@(Elem key _ _)) _) : t) | key ≡ tag = (t, Ok e)
read_ELEMENT' tag ((CString _ s _) : t) | Prelude.and (map isSpace s) =
  read_ELEMENT' tag t
read_ELEMENT' tag (((CMisc _ _) : t)) = read_ELEMENT' tag t
read_ELEMENT' tag l =
  (l, Error ("I expect " ++ tag ++ " element" ++ moreinfo))
  where moreinfo :: String
    moreinfo = ": [" ++ concat (map conts l) ++ "]"
      -- |
    conts :: Content i → String
    conts (CElem (Elem k _ _) _) = "<" ++ k ++ "/>"
    conts (CString _ s _) = s
    conts (CRef _ _) = "(ref)"
    conts (CMisc _ _) = "(misc)"
  -- |
read_ELEMENT :: String → STM Result [Content i] (Element i)
read_ELEMENT tag = liftSTM (ST (λs → read_ELEMENT' tag s))
  -- |
show_ELEMENT :: String → [Attribute] → [Content ()] → [Content ()]
show_ELEMENT tag attr contents = [CElem (Elem tag attr contents) ()]
  -- |
attributes :: Element i → [Attribute]
attributes (Elem _ x _) = x
  -- |
childs :: Element i → [Content i]
childs (Elem _ _ x) = x
  -- |
```

```
type PCDATA = Prelude.String
  -- |
read_PCDATA' :: [Content i] → ([Content i], Result PCDATA)
read_PCDATA' []                    = ([], return [])
read_PCDATA' ((CString _ y _) : t) =
  let (a, b) = read_PCDATA' t
  in (a, b >>= return · (y++))
read_PCDATA' ((CRef y _) : t) =
  let (a, b) = read_PCDATA' t
  in (a, b >>= return · (read_REF y++))
read_PCDATA' (l@((CElem _ _) : _)) = (l, return [])
read_PCDATA' (_ : t)               = read_PCDATA' t
  -- |
read_REF :: Reference → PCDATA
read_REF (RefEntity x) = '&' : x ++ ";"
read_REF (RefChar x) = '#' : show x
  -- |
read_PCDATA :: STM Result [Content i] PCDATA
read_PCDATA = liftSTM (ST (λs → read_PCDATA' s))
  -- |
show_PCDATA :: PCDATA → [Content ()]
show_PCDATA pcdata = [CString False pcdata ()]


  -- * Elements
  -- |
read_MAYBE :: STM Result [Content i] a →
  STM Result [Content i] (Maybe a)
read_MAYBE st = STM (λs →
  ((stateM st s) >>= (λ(z1, z2) → return (z1, return z2)))
  `mplus` return (s, Nothing))
  -- |
show_MAYBE :: (a → [Content ()]) → Maybe a → [Content ()]
show_MAYBE f = maybe [] f
  -- |
read_LIST :: Eq i ⇒ STM Result [Content i] a → STM Result [Content i] [a]
read_LIST st = STM (λs →
  let x = stateM st s
  in case x of
    Ok (x1, x2) → if s ≡ x1 then return (s, [x2])
      else let y = stateM (read_LIST st) x1
        in case y of
          Ok (y1, y2) → return (y1, x2 : y2)
          Error _ → return (x1, [x2])
    Error _ → return (s, [])
  )
  -- |
show_LIST :: (a → [Content ()]) → [a] → [Content ()]
show_LIST f = concat · map f
  -- |
read_LIST1 :: Eq i ⇒ STM Result [Content i] a → STM Result [Content i] [a]
read_LIST1 st = STM (λs →
  let x = stateM st s
  in case x of
    Ok (x1, x2) → if s ≡ x1 then return (s, [x2])
      else let y = stateM (read_LIST1 st) x1
        in case y of
          Ok (y1, y2) → return (y1, x2 : y2)
          Error _ → return (x1, [x2])
```

```
    Error _ → fail "empty list"
  )
  -- |
show_LIST1 :: (a → [Content ()]) → [a] → [Content ()]
show_LIST1 = show_LIST


  -- * Read
  -- |
read_1 :: Monad m ⇒ STM m s a → s → STM m s' a
read_1 st1 s = returnSTM (stateM st1 s ≫= (λ(_, x) → return x))
  -- |
read_2 :: Monad m ⇒ STM m s a → STM m s b → s → STM m s' (a, b)
read_2 st1 st2 s = returnSTM (do
    (s1, a) ← stateM st1 s
    (_, b) ← stateM st2 s1
    return (a, b))
  -- |
read_3 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    s → STM m s' (a, b, c)
read_3 st1 st2 st3 s = returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (_, c) ← stateM st3 s2
    return (a, b, c))
  -- |
read_4 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    STM m s d → s → STM m s' (a, b, c, d)
read_4 st1 st2 st3 st4 s = returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (_, d) ← stateM st4 s3
    return (a, b, c, d))
  -- |
read_5 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    STM m s d → STM m s e → s → STM m s' (a, b, c, d, e)
read_5 st1 st2 st3 st4 st5 s = returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (_, e) ← stateM st5 s4
    return (a, b, c, d, e))
  -- |
read_6 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    STM m s d → STM m s e → STM m s f → s → STM m s' (a, b, c, d, e, f)
read_6 st1 st2 st3 st4 st5 st6 s = returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (_, f) ← stateM st6 s5
    return (a, b, c, d, e, f))
  -- |
read_7 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    STM m s d → STM m s e → STM m s f → STM m s g →
    s → STM m s' (a, b, c, d, e, f, g)
```

```
read_7 st1 st2 st3 st4 st5 st6 st7 s = returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (s6, f) ← stateM st6 s5
    (_, g) ← stateM st7 s6
    return (a, b, c, d, e, f, g))
  -- |
read_8 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    STM m s d → STM m s e → STM m s f → STM m s g →
    STM m s h → s → STM m s' (a, b, c, d, e, f, g, h)
read_8 st1 st2 st3 st4 st5 st6 st7 st8 s = returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (s6, f) ← stateM st6 s5
    (s7, g) ← stateM st7 s6
    (_, h) ← stateM st8 s7
    return (a, b, c, d, e, f, g, h))
  -- |
read_9 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    STM m s d → STM m s e → STM m s f → STM m s g →
    STM m s h → STM m s i → s → STM m s' (a, b, c, d, e, f, g, h, i)
read_9 st1 st2 st3 st4 st5 st6 st7 st8 st9 s = returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (s6, f) ← stateM st6 s5
    (s7, g) ← stateM st7 s6
    (s8, h) ← stateM st8 s7
    (_, i) ← stateM st9 s8
    return (a, b, c, d, e, f, g, h, i))
  -- |
read_10 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    STM m s d → STM m s e → STM m s f → STM m s g →
    STM m s h → STM m s i → STM m s j →
    s → STM m s' (a, b, c, d, e, f, g, h, i, j)
read_10 st1 st2 st3 st4 st5 st6 st7 st8 st9 st10 s =
  returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (s6, f) ← stateM st6 s5
    (s7, g) ← stateM st7 s6
    (s8, h) ← stateM st8 s7
    (s9, i) ← stateM st9 s8
    (_, j) ← stateM st10 s9
    return (a, b, c, d, e, f, g, h, i, j))
  -- |
read_11 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
```

```
      STM m s d → STM m s e → STM m s f → STM m s g →
      STM m s h → STM m s i → STM m s j → STM m s k →
      s → STM m s' (a, b, c, d, e, f, g, h, i, j, k)
read_11 st1 st2 st3 st4 st5 st6 st7 st8 st9 st10 st11 s =
  returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (s6, f) ← stateM st6 s5
    (s7, g) ← stateM st7 s6
    (s8, h) ← stateM st8 s7
    (s9, i) ← stateM st9 s8
    (s10, j) ← stateM st10 s9
    (_, k) ← stateM st11 s10
    return (a, b, c, d, e, f, g, h, i, j, k))
  -- |
read_12 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
      STM m s d → STM m s e → STM m s f → STM m s g →
      STM m s h → STM m s i → STM m s j → STM m s k →
      STM m s l → s → STM m s' (a, b, c, d, e, f, g, h, i, j, k, l)
read_12 st1 st2 st3 st4 st5 st6 st7 st8 st9 st10 st11 st12 s =
  returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (s6, f) ← stateM st6 s5
    (s7, g) ← stateM st7 s6
    (s8, h) ← stateM st8 s7
    (s9, i) ← stateM st9 s8
    (s10, j) ← stateM st10 s9
    (s11, k) ← stateM st11 s10
    (_, l) ← stateM st12 s11
    return (a, b, c, d, e, f, g, h, i, j, k, l))
  -- |
read_13 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
      STM m s d → STM m s e → STM m s f → STM m s g →
      STM m s h → STM m s i → STM m s j → STM m s k →
      STM m s l → STM m s n → s → STM m s' (a, b, c, d, e, f, g, h, i, j, k, l, n)
read_13 st1 st2 st3 st4 st5 st6 st7 st8 st9 st10 st11 st12 st13 s =
  returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (s6, f) ← stateM st6 s5
    (s7, g) ← stateM st7 s6
    (s8, h) ← stateM st8 s7
    (s9, i) ← stateM st9 s8
    (s10, j) ← stateM st10 s9
    (s11, k) ← stateM st11 s10
    (s12, l) ← stateM st12 s11
    (_, m) ← stateM st13 s12
    return (a, b, c, d, e, f, g, h, i, j, k, l, m))
```

```
  -- |
read_17 :: Monad m ⇒ STM m s a → STM m s b → STM m s c →
    STM m s d → STM m s e → STM m s f → STM m s g →
    STM m s h → STM m s i → STM m s j → STM m s k →
    STM m s l → STM m s n → STM m s o → STM m s p →
    STM m s q → STM m s r → s →
    STM m s' (a, b, c, d, e, f, g, h, i, j, k, l, n, o, p, q, r)
read_17 st1 st2 st3 st4 st5 st6 st7 st8 st9
  st10 st11 st12 st13 st14 st15 st16 st17 s =
  returnSTM (do
    (s1, a) ← stateM st1 s
    (s2, b) ← stateM st2 s1
    (s3, c) ← stateM st3 s2
    (s4, d) ← stateM st4 s3
    (s5, e) ← stateM st5 s4
    (s6, f) ← stateM st6 s5
    (s7, g) ← stateM st7 s6
    (s8, h) ← stateM st8 s7
    (s9, i) ← stateM st9 s8
    (s10, j) ← stateM st10 s9
    (s11, k) ← stateM st11 s10
    (s12, l) ← stateM st12 s11
    (s13, n) ← stateM st13 s12
    (s14, o) ← stateM st14 s13
    (s15, p) ← stateM st15 s14
    (s16, q) ← stateM st16 s15
    (_, r) ← stateM st17 s16
    return (a, b, c, d, e, f, g, h, i, j, k, l, n, o, p, q, r))


instance (Show a, Show b, Show c, Show d, Show e, Show f, Show g, Show h,
    Show i, Show j, Show k, Show l, Show m, Show n, Show o, Show p, Show q)
      ⇒ Show (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q) where
  show (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q) = "(" ++ show a ++ "," ++
    show b ++ "," ++ show c ++ "," ++ show d ++ "," ++ show e ++ "," ++
    show f ++ "," ++ show g ++ "," ++ show h ++ "," ++ show i ++ "," ++
    show j ++ "," ++ show k ++ "," ++ show l ++ "," ++ show m ++ "," ++
    show n ++ "," ++ show o ++ "," ++ show p ++ "," ++ show q ++ ")"
instance (Eq a, Eq b, Eq c, Eq d, Eq e, Eq f, Eq g, Eq h,
    Eq i, Eq j, Eq k, Eq l, Eq m, Eq n, Eq o, Eq p, Eq q)
      ⇒ Eq (a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q) where
    (a1, b1, c1, d1, e1, f1, g1, h1, i1, j1, k1, l1, m1, n1, o1, π1, q1) ≡
      (a2, b2, c2, d2, e2, f2, g2, h2, i2, j2, k2, l2, m2, n2, o2, π2, q2) =
      (a1 ≡ a2) ∧ (b1 ≡ b2) ∧ (c1 ≡ c2) ∧ (d1 ≡ d2) ∧ (e1 ≡ e2) ∧
      (f1 ≡ f2) ∧ (g1 ≡ g2) ∧ (h1 ≡ h2) ∧ (i1 ≡ i2) ∧ (j1 ≡ j2) ∧
      (k1 ≡ k2) ∧ (l1 ≡ l2) ∧ (m1 ≡ m2) ∧ (n1 ≡ n2) ∧ (o1 ≡ o2) ∧
      (π1 ≡ π2) ∧ (q1 ≡ q2)
```

3 Test

```
  -- |
module Main where
import Text.XML.MusicXML hiding (String)   -- MusicXML package
import System.IO
import Data.Maybe
```

```
import System.Environment
import System.Console.GetOpt
import Prelude


  -- |
data Option = List FilePath
   | Help
   | Version
     deriving (Eq, Show)
options :: [OptDescr Option]
options = [
   Option ['v', 'V'] ["version"] (NoArg Version) "show version number"
   , Option ['h', 'H', '?'] ["help"] (NoArg Help) "show help"
   , Option ['l', 'm'] ["manifest"] (ReqArg List "MANIFEST") "manifest file"
   ]
   -- |
header :: String → String
header prog = "Usage: " ++ prog ++ " [OPTIONS...] FILES..."
   -- |
proc :: [Option] → [String] → IO ()
proc [] files = main' (zip ([1 ..] :: [Int]) files)
proc ((List file) : t) files = do
   list ← readFile file
   proc t (lines list ++ files)
proc (_ : t) files = proc t files


mkoutput :: FilePath → FilePath
mkoutput = reverse · ("lmx.tuptuo-"++) · drop 4 · reverse


   -- |
put :: String → IO ()
put msg = putStr msg ≫ hFlush stdout
putLn :: String → IO ()
putLn msg = putStrLn msg ≫ hFlush stdout
putBool :: Bool → IO ()
putBool True = putStrLn "[Ok]"
putBool False = putStrLn "[Failed]"


   -- |
inout :: FilePath → IO ()
inout file = do
   putLn ("file: " ++ show file) ≫ put "Reading "
   contents ← readFile file
   r1 ← return (read_CONTENTS read_MusicXMLDoc file contents)
   r2 ← return (case isOK r1 of
      True → Just (mkoutput file, fromOK r1); False → Nothing)
   putBool (isOK r1)
   put "Writing "
   r3 ← return (fmap (λ(a, b) →
      (a, show_CONTENTS show_MusicXMLDoc b)) r2)
   maybe (return ()) (uncurry writeFile) r3
   putBool (isJust r3)


   -- |
main :: IO ()
main = do
```

```
  argv ← getArgs
  prog ← getProgName
  case getOpt Permute options argv of
    (o, n, [ ]) | Help ∈ o → putStrLn (usageInfo (header prog) options)
        | Version ∈ o → putStrLn (unwords [prog])
        | otherwise → proc o n
    (_, _, errs) → putStrLn (unlines errs ++ usageInfo (header prog) options)
  -- |
main′ :: [(Int, FilePath)] → IO ()
main′ [ ] = return ()
main′ ((a, b) : t) = do
  putLn ("\nNumber: " ++ show a) ≫ inout b ≫ main′ t
```

4 Conclusion

This library handle music notation at musicxml format. This Haskell library is translation from DTD specification.

References

[1] Michael Good. Lessons from the adoption of musicxml as an interchange standard, 2006.

[2] Michael Good and Geri Actor. Using musicxml for file interchange. *IEEE*, 2003.

[3] Paul Haudak, John Hudges, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. *ACM*, 2007.

[4] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*, 2002.

[5] David Brian Williams. Musicxml: The new link for sharing sibelius and finale files. 2008.