CoST version 0.2
Copenhagen SGML Tool

Klaus Harbo

CoST version 0.2
Copenhagen SGML Tool

This is the current draft documentation for the Beta version of CoST, the Copenhagen SGML Tool. As this documentation is merely a draft, it may contain errors and omissions. An updated version of this document will appear at a later time.

This manual was written by Klaus Harbo.

Printed and published in Denmark.

Version Beta 0.2, December 1993.

Klaus Harbo
Dept. of Computer Science & Euromath Center
University of Copenhagen
Universitetsparken 5
2100 København Ø
Denmark
E-mail: `Klaus.Harbo@euromath.dk`

Contents

1

Introduction to CoST

1.1 What is CoST?

CoST (Copenhagen SGML Tool) is a structure-controlled SGML application programming tool. It is built on top of a public domain SGML tool: the SGMLS parser made by James Clark.

With CoST you can write translation specifications for SGML document instances. CoST is purely structure driven, i.e. it gives you access to the structure of the SGML document instance. It won't, however, let you access the lexical and syntactical details in the SGML entities that represent the document instance in storage.

You can write CoST programs that will translate SGML document instances or perform other processing in response to SGML documents.

You program CoST using TCL - Tool Command Language.

1.2 About this manual

The current release of CoST is a Beta version. This means that CoST isn't finished and may yet contain bugs.

This manual is intended to give its readers enough information to get started with CoST, but the manual - like CoST itself - is still a Beta version. Hopefully there will be a day, when a more complete manual will be made available.

1.2.1 This manual and SGML

To appreciate fully the contents of this manual you'll need to know about SGML. This manual is not a tutorial in SGML, far from it.

If you want to know about SGML, read literature such as Charles Goldfarb's "The SGML Handbook" or Eric van Herwijnen's "Practical SGML". Another valuable source of SGML information (and advice if you're really stuck!) is the USENET newsgroup comp.text.sgml.

1.2.2 This manual and TCL and derivatives

The core of the programming language of CoST is an embeddable programming language called TCL - Tool Command Language - created by John Osterhout of UC Berkeley. To use CoST in any serious context, you'll need to learn the TCL programming language.

This manual will (eventually) contain examples of uses of CoST which will give you an idea of what TCL is like. It won't teach TCL. To use CoST you'll need to get the documentation for TCL. (Check the CoST installation material for availability of TCL and its documentation.)

In addition to TCL, CoST uses an object-oriented extension to TCL, called [incr TCL]. You won't have to learn [incr TCL] to use CoST, but it may be a good idea to look at it anyway, since many things in CoST can be done more elegantly if you use the object-oriented facilities than if you don't.

The CoST programming model is object-oriented.

1.2.3 This manual and the parser SGMLS

Since part of CoST is the SGMLS parser, you'll need to get acquainted with SGMLS as well. This manual will not tell you how to run SGMLS or how to set SGMLS up to find SGML or document type declarations. You'll have to figure that out by reading the SGMLS documentation. (Check the CoST installation material for availability of SGMLS and its documentation.)

1.3 The current implementation of CoST

Speed is not the strength of the current implementation of CoST. As mentioned, you program CoST using TCL. However, at this point all of CoST is implemented in TCL too because TCL lets you develop software quickly. Since TCL is an interpreted language, its execution speed leaves something to desired.

However, TCL lets you convert part or all of a system to a compiled language (usually C or C++).

CoST development still focus on correctness, functionality and flexbility. It doesn't make sense to implement the CoST kernel in C before the system as such is fairly stable. When that happens, and if there is sufficient interest, porting the CoST kernel to C or C++ will be considered.

It is believed that this will increase execution speed significantly. It is difficult to estimate the improvement (it depends very much on the application, since user code will still be interpreted TCL), but it is not unrealistic to expect (well, hope for) a speed-up of an order of magnitude or so.

1.4 Organization of this manual

The material of this manual is divided into the following chapters:

1. This introduction
2. Overview of a CoST application (including a simple example)
3. Declarations in a CoST application
4. Available commands in a CoST declaration

2

Overview of CoST

2.1 A structure-controlled tool

CoST is built on top of the SGMLS parser. This means that CoST applications have only access to purely structural information (ESIS, Element Structure Information Set) about the SGML document instance it is working on. The relation between SGMLS and CoST is best described in a figure, see figure 2.1.The relation between SGMLS and CoST.

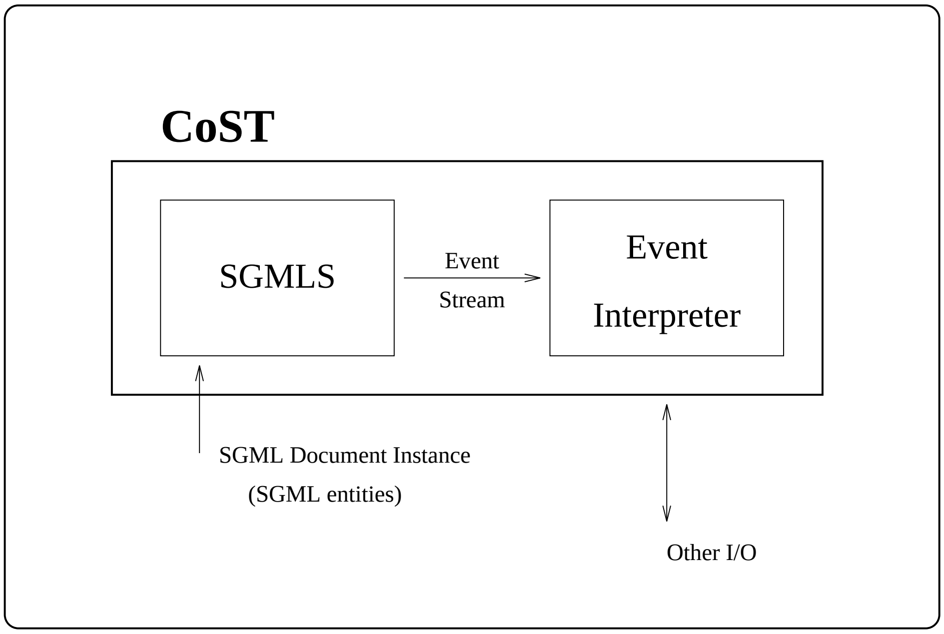

Figure 2.1: Relationship between CoST and SGMLS.

2.1.1 Handling SGML entities (input)

You can see in the figure that it is SGMLS that handles everything related to the actual SGML document entities. CoST does not interfere at all with this. A consequence of this is that CoST will process a document only if SGMLS can parse it. A corollary of this fact is that you have to have a working setup of SGMLS to process documents with CoST. In particular the SGML_PATH environment variable must be set correctly so SGMLS can find the entities needed for parsing. Check with the documentation of SGMLS.

2.1.2 The Event Stream

From the figure it is obvious that the event stream flowing between the parser and the interpreter is essential for the operation of CoST. To use CoST, it is necessary to understand the nature of the contents of the stream and the actions taken by the event interpreter in response to it.

If you are familiar with the output of the SGMLS parser, the event stream is basically this output transformed to be suitable input for the event interpreter. For those who are not so familiar with SGMLS and its output, a brief description of the event stream follows...

When an SGML parser reads through the character stream of the SGML entities making up the SGML document instance in storage, it interprets the characters according to the SGML declaration and the DTD. In the abstract, the result of this interpretation is a parse tree of the document instance. The parse tree contains information about the elements of the document instance structure and their attributes.

The event stream flowing from SGMLS to the event interpreter is this information. The sequence of the information is as if doing in-order traversal of the parse tree. At each node in the parse tree information is given to the interpreter about the name of node (the element's Generic Identifier, GI) and information about its attributes (their values or absence of such in case the attribute was declared #IMPLIED and its value wasn't specified on the start tag).

An example is appropriate. The following SGML document instance

```
<!DOCTYPE memo [
  <!ELEMENT memo    - -    (title,body)>
  <!ELEMENT body    - O    (section+)>
  <!ATTLIST memo
      style    CDATA  #REQUIRED
      author   CDATA  #REQUIRED
  >
  <!ELEMENT title   --     (#PCDATA)>
  <!ELEMENT section - -    (header,para*,section*)>
  <!ELEMENT (header!para)
                    -      (#PCDATA)>
]>
<?USESPEC memo-latex.tcl>
<memo author="Klaus Harbo" style=simple>
```

```
<title>Parsing SGML documents</title>
<body>
<section>
<header>First section title</header>
<para>This is the first paragraph in the first secion
<para>This is the second paragraph in the first section
</section>
</memo>
```

would produce the following event stream from the SGML parser:

```
MEMO     START (STYLE=simple, AUTHOR=Klaus Harbo)
TITLE    START
#PCDATA  "Parsing SGML documents"
TITLE    END
BODY     START
SECTION  START
HEADER   START
#PCDATA  "First section title"
HEADER   END
PARA     START
#PCDATA  "This is the first paragraph in the first section"
PARA     END
PARA     START
#PCDATA  "This is the second paragraph in the firstsection"
PARA     END
SECTION  END
BODY     END
MEMO     END
```

Note that even though there are no end tags on PARA elements in the SGML document, the end of the PARA elements are inferred by the parser and end events are generated accordingly.

Note that the processing instruction USESPEC controls which CoST translation spec to use for the translation.

More events than those presented here are generated (e.g. Record End events), but it is not necessary to discuss them here. The documentation on the usage of CoST constructs should provide enough information to use the tool without going to this level of detail.

2.2 The Event Interpreter

We need to describe what the event interpreter does in response to the events produced by the parser.

To use CoST effectively, it is important to understand how the document structure is represented in CoST during processing. The main idea is that the event interpreter builds a parse structure from the events produced by the parser.

Basically, the event interpreter pushes a node object on the parse stack when a START element event occurs, and pops it off the stack when an END element event occurs.

We'll see later that it does more than that, but this is the most fundamental operation. The consequence is that the object representing a given element in the document instance will be on top of the objects of its parents in the stack. The programming constructs of CoST let the programmer access information about objects on the parse stack (ie. the ancestors of the current node in the parse tree).

CoST provides more than simply ancestor information. In particular, you can control whether an object is destroyed after it is popped off the stack. If the object is not destroyed, you can access it later (using the methods defined on the object) when you need the information it contains.

The object management features are discussed later.

2.2.1 The objects on the parse stack

Objects are pushed on the parse stack according to the events produced by the parser. The word "objects" is meant literally, since CoST uses an object-oriented programming model. The object classes of the objects pushed on the stack (and thus the variables in the objects and the methods that can be applied to the objects) are controlled by the CoST programmer in coorporation with the CoST class generator.

Programming CoST, among other things, means specifying the actions to be taken when parse events occur. This is done by declaring the element object classes that will handle the events. Thus, if you want action X to be taken at the start of element A and action Y to be taken at the end of element A, you declare an object class (element class) whose name is A, whose start method is X and end method is Y:

```
element A {
  start {
    <whatever actions X consist of>
  }
  end {
    <whatever actions Y consist of>
  }
}
```

This declaration creates an object class in CoST. When the event element A START occurs, an object from this class is instantiated and pushed on the parse stack.

You are not limited to declaring start and end methods (which are special anyway, because they are invoked by the interpreter whereas any other method you declare must be invoked explicitly by some other method, e.g. the start or end methods).

In general, you can declare:

start method This method is invoked just after the object is pushed on the node object stack. The start method should never be called explicitly by statements in user code; it is automatically called by the event interpreter.

end method This method is invoked just before the node object is popped off the parse stack. The end method should never be called explicitly by statements in user code; it is automatically called by the event interpreter.

Other methods Any other methods than start and end methods. These methods are always explicitly called by user code (e.g. the code inside start or end methods). They are especially useful when there is a need to move information from one point in the parse tree to another (e.g. if you want to collect all notes in a document at the end of the document). Methods can be declared with any number of parameters.

Procedures Procedures can be declared either inside element classes, in which case they are callable only from within in objects that belong to the class or outside in which case they can be called from anywhere in the CoST spec. Check the TCL documentation for more information.

Variables There are several kinds of variables declarable inside element object classes:

Protected Variables that are only accessible from within that particular object.

Public Variables that are accessible from outside the object (ie. its value can both be read and written from outside the object).

Common Variables that are shared between every member of an object class.

The object-oriented facilities of CoST leverages directly off the facilities of [incr tcl]. To get more information about the programming options, see the documentation of TCL and [incr tcl].

2.2.2 An example

To give you a better idea of what CoST specs are like, we present a small example. The spec presented is a complete CoST translation specification.

The following CoST translation spec would convert the example SGML document instance presented already (in the section describing the event stream) to LaTeX:

```
element MEMO {
  start {
    puts "\\documentstyle\{[attrValue STYLE]\}"
    puts "\\author\{[attrValue AUTHOR]\}"
  }
}

element TITLE {
  start { puts stdout "\\title\{" nonewline }
  end   { puts stdout "\}" }
}
```

```
element BODY {
  start { puts "\\begin\{document\}" }
  end   { puts "\\end\{document\}" }
}

element SECTION {
  start { puts stdout "\\chapter" nonewline }
}

element HEADER {
  start { puts stdout "\{" nonewline }
  end   { puts stdout "\}" nonewline }
}

element PARA {
  start {
    puts ""
    puts ""
  }
}
```

This CoST spec would, given the SGML document instance example presented above (and assuming the CoST spec is in a file called "memo-latex.tcl"), produce the following output:

```
\documentstyle{book}
\author{Klaus Harbo}
\title{Parsing SGML documents}
\begin{document}
\chapter{First section title}

This is the first paragraph in the first section

This is the second paragraph in the first section
\end{document}
```

Of course this is a very simple example, but it does illustrate the basic idea of CoST specs. Note that attribute values are accessed with an [attrValue <attrName>] construct.

Later we'll show that CoST allows you to control how text and SDATA entities, etc are handled (the handling is implicit in the translation spec given here).

3

Declarations in CoST

This chapter is divided into two parts:

1. The declarations that are related to procesing performed on the elements of SGML document instance structure.
2. The other declarations, i.e. declarations to control handling of Record End events, SDATA entities, processing instructions, etc.

3.1 Element declarations

There are two ways to specify element specific actions, one in which you specify actions for a specific element type (GI), and one in which you specify what should be done if no action has been declared for the element type (a default declaration). We'll look at these to the options in the next two subsections.

3.1.1 The element declaration

Basically the element declarations are like this:

```
element <GI> {
  start {
    <TCL-stmts>
  }

  end {
    <TCL-stmts>
  }
}
```

As you can see, the actions you want performed at the start and end of the element are specified separately.

In the current version, the name of the GI must be specified with capital letters.

start and end declarations

The start and end declarations in element specifications are lists of TCL statements that are executed at the time you encounter the beginning or end respectively of the element in question. For example the declaration:

```
element TITLE {
  start {
    puts stdout "\{\\LARGE\\bf " nonewline
  }
  end { puts stdout "\}" nonewline }
}
```

...formats TITLE elements as large bold-font in TeX.

Declaration of variables

As briefly described in the Overview chapter, you can declare variables in your CoST specifications.

Variables outside element specifications

Just like TCL and [incr tcl], you can declare normal variables and objects in CoST. If declared outside the element specifications, there is no difference between creating variables and objects in CoST and regular TCL and [incr tcl] programs. For example, you can declare counters that are accessible everywhere in the CoST spec.

To see how this is done, check the TCL and [incr tcl] documentation.

Variables inside element specifications

Declaring variables inside element specification is like declaring variables in itcl_class declarations in [incr tcl]. As mentioned in the overview, there are three types of variables in CoST element declarations, protected, public and common. See the overview and the [incr tcl] documentation for details about the meaning and use of the different kinds of variables.

You declare variables with the following syntax:

```
element myelement {
  protected { provar 1 }
  public    { pubvar "Mystring" }
  common    { comvar {my list of things} }
  protected   noinitvar
  start { ... }
  end { ... }
}
```

In the above example four variables are declared:

1. The protected variable provar is initialized to the value "1"

2. The public variable pubvar to the value "Mystring"

3. The common variable comvar to the list value "my list of things"

4. The protected variable noinitvar is not initialized at all.

NOTE: There is one important syntactical difference between the declaration of variables in [incr tcl] and CoST: In [incr tcl] variables are declared without braces around the name and initial value, whereas in CoST the braces must be present when variables are given initial values. The reason for this difference is technical and is related to the way CoST parses element class declarations.

Declaration of methods

In addition to the special methods start and end, it is possible to declare element classes to have user defined methods.

Here's an example declaration of an element class which contains several methods:

```
element myelement {
  protected { myval 0 }
  method incrbyone {} {
    incr myval
  }
  method add value {
    incr myval $value
  }
  method multadd {value times} {
    for {set i 1} {$i<=$times} {incr i} {
      incr myval $value
    }
  }
  start { ... }
  end { ... }
}
```

The myelement element class contains a protected variable, myval, which can be modified from outside the object via three element class methods.

1. The method incrbyone is simple, having zero arguments. The empty argument list is indicated by the empty list after the name of the method.

2. The method add has one formal parameter, value. It is legal to let this single formal parameter be a single-member list ("method add value ...", in this case). This makes declaring single parameter methods more consistent with the declaration of methods with zero or several parameters, since they must always use lists.

3. The method multadd has two formal paramenters, value and times, indicated by the two member list after the method name. This method uses a TCL for-loop to increase myval several times.

NOTE: The same effects could have been achieved with much simpler means using public variables. The example serves to illustrate the idea of methods.

Declaration of procedures

Procedures outside element specifications

Outside element specifications, procedures can be declared just like in regular TCL programs. See the TCL documentation for details.

Procedures inside element specifications

Like methods, procedures can be declared inside element object declarations. The only differences between procedures and methods is that methods can be invoked from outside the object, whereas procedures cannot.

A declaration of a procedure could look like this:

```
element ... {
  ...
  proc noArgProc {} {
    ...
  }
  proc oneArgProc myarg {
    ...
  }
  ...
}
```

See the TCL and [incr tcl] documentation for details.

Element class inheritance

It is possible to have complex object class hierarchies by defining regular [incr tcl] classes that are inherited by the element object class. That way it is possible to avoid specifying methods that are defined for many element object classes more than once. The inheritance also makes it possible for objects of different element object classes to share part of their name space by means of common variables in the class they both inherit.

Example:

```
itcl_class fooClass {
  common foovar
  protected myvar
  <whatever variables, procedures, methods>
}

itcl_class barClass {
  common barvar
  <whatever variables, procedures, methods>
}

element SOMEGI {
  inherit {fooClass}
  ...
}
```

```
element THISGI {
  inherit {fooClass barClass}
  ...
}

element THATGI {
  inherit {fooClass barClass}
  ...
}
```

The example shows that SOMEGI element objects inherit from fooClass and that THISGI element objects inherit both from fooClass and barClass (multiple inheritance). Element objects of type SOMEGI, THISGI, and THATGI all share the variable foovar, whereas only THISGI and THATGI have access to barvar.

These facilities provide a high level of flexibility.

Note: The variables foovar and barvar are shared (the objects share the variable, i.e. they have access to the same value), whereas myvar is a protected variable (each object has its own copy of myvar with its own value).

3.1.2 The default declaration

In addition to declaration of specifications for actions taken when start and end events occur for specific element types, it is possible to specify actions for every element that hasn't been specified specifically. That way you can make specifications in which most elements are processed the same way simply by specifying this standard processing. The elements that need special processing are specified separately.

The default specification is declared as follows:

```
default {
  <variable, procedure and method declarations...>

  start {
    <whatever special start actions should be taken bydefault>
  }
  end {
    <whatever special end actions should be taken bydefault>
  }
}
```

If no default specification is provided in the CoST spec, the following declaration is used:

```
default {
  start {}
  end   {}
}
```

In other words, no special processing will take place for elements that aren't specified in the CoST spec.

3.2 General declarations

In addition to declare the actions that handle events related to elements and associated attributes, there are certain other kinds of events whose handling are declarable.

The events in question are related to the handling of SDATA entities, processing instructions, and Record Ends.

3.2.1 Handling of text

The characters of the TEXT pseudo-element is accessed through the special variable data in the CoST TEXT declaration.

A very simple declaration for handling character data (declared as #PCDATA, #CDATA, or #RCDATA) could look as follows:

```
TEXT {
  when {
    {in NOTE} {
      noteMgr registerNewNote $data
    }
    default {
      puts stdout $data nonewline
    }
  }
}
```

In this case the character content of NOTE elements is given to an object (accessed through a global variable) which handles notes, whereas character content of other elements is printed on stdout. The note manager may format and print the note content right away, or it may collect all the notes in the document and print them all at the end; we don't know, since its declaration is not shown here.

(This example gives us a preview of the contextual facilities of CoST. All document context facilities (described elsewhere) are available in TEXT objects.)

Specification of handling of TEXT is optional. The following TEXT specification is used as default:

```
TEXT {
  puts stdout $data nonewline
}
```

3.2.2 Handling SDATA entity events

SDATA entity references are handled by the SDATA declaration. In the SDATA declaration, the characters of the SDATA replacement text are accessed through the special variable data.

A simple SDATA declaration handling the Danish national special characters could be declared as follows:

```
SDATA {
  case $data in {
    \[aelig\]    { set out \{\\ae\} }
    \[AElig\]    { set out \{\\AE\} }
    \[oslash\]   { set out \{\\o\}  }
    \[Ooblique\] { set out \{\\O\}  }
    \[aring\]    { set out \{\\aa\} }
    \[Aring\}    { set out \{\\AA\} }
    default      { set out ? }
  }
  puts stdout $out nonewline
}
```

This SDATA declaration would produce suitable TeX commands for the entity references to Danish national characters.

SDATA declarations can access all CoST document context facilities.

If no SDATA declaration is specified in the CoST spec, the following declaration is used as default:

```
SDATA {
  puts stdout " " nonewline
}
```

In other words, by default a space character is printed on stdout for every SDATA entity reference.

3.2.3 Handling Record End events

The RECORDEND declaration declares the action to handle Record End Events.

If no RECORDEND declaration is specified in the CoST spec, the following declaration is used as default:

```
RECORDEND {
  puts ""
}
```

In other words, by default a newline character is printed on stdout.

3.2.4 Handling Processing Instruction events

Handling of Processing instructions is declared with the PI declaration:

```
PI {
  <statements handling the Processing Instruction>
}
```

The data of the processing instruction is accessible in the variable data.

If no PI declaration is specified in the CoST spec, the following declaration is used as default:

```
PI { return }
```

4

Commands in CoST declarations

This chapter describes briefly the CoST specific commands that are available inside CoST declarations, i.e. the commands that have been added to the basic TCL and [incr tcl] commands.

The chapter has 2 main parts, a part that describes the commands that are used to access attribute information, and a part that describes the commands that are used to access information about the context of the element object that executes the commands, e.g. information about the element type of its parent.

4.1 Accessing attribute information

Attribute information is an important aspect of the Element Structure Information Set (ESIS). In the following sections, commands are described that readthe value of a named attribute (attrValue), learn whether a value was specified on the start tag of the element (isImplicit), learn whether a named attribute was declared for the element (isDeclared), and give the list of declared attribute names (declaredAttrNames).

Note: Perhaps a little clarification is needed on the selection of commands available in CoST declarations: It is not possible to learn if an attribute was declared #IMPLICIT or #REQUIRED, or whether it was, say, declared CDATA or TOKEN, since this type of information is not part of ESIS. All we can learn is if the attribute exists at all for the element type and, if that is the case, if the attribute has an explicit value (i.e. specified on the start tag of the element, a default value, or a value inherited with a #CURRENT declaration), or an implicit value (i.e. not specified on the start tag). If the attribute has an explicit value, we can find out what it is with the attrValue command.

4.1.1 attrValue

The attrValue command is used for accessing the value of attributes for which a value exists explicitly. In accordance the rules prescribed by ESIS, it doesn't make any difference whether it was specified on the start tag or defaulted (i.e. the attribute has a declared default value).

attrValue is a regular command, but it is often used as an expression in puts or if statements (as above) in which case it is in brackets ("[]") which force evaluation on TCL (check with the TCL documentation).

attrValue produces an error if its argument is not a declared attribute on the element or if the value of the attribute is implicit.

The argument to attrValue (the name of the attribute) must always be given in uppercase.

Example:

```
element ... {
  start {
    ...
    if {[attrValue STYLE]=="BOOK"} {
      puts "Style attribute = book"
    }
    ...
  }
  ...
}
```

4.1.2 isImplicit

The purpose of the isImplicit command is to find out whether or not a named attribute has an explicit value or not.

isImplicit returns 1 if the value of the attribute has an implicit value (i.e. was declared #IMPLIED and not specified on the start tag), and returns 0 if the attribute has an explicit value.

isImplicit produces an error if its argument (the name of the attribute) is not a declared attribute on the element.

Example:

```
element ... {
  ...
  start {
    ...
    if {[isImplicit STYLE]} {
      puts "No value for STYLE attributespecified."
    } else {
      puts "STYLE attribute = [attrValueSTYLE]."
    }
    ...
  }
  ...
```

```
}
```

4.1.3 isDeclared

The purpose of isDeclared is to find out whether a named attribute exists on the element or not. Of course, this information is available in the DTD, so if a CoST spec is only used with one DTD, this command is not of much use. However, if the CoST specs are modularized and some of the modules are used with more than one DTD, it may be convenient to find whether or not a given attribute is available on some element. (The same is the case if Link Process Definitions (LPDs) are used, but this is slightly irrelevant now, since SGMLS does not support LPDs. Hopefully this will change in a future version of SGMLS.)

isDeclared returns 1 if its argument (the name of the attribute) is a declared attribute on the element in the DTD, 0 otherwise.

Example:

```
element ... {
  ...
  start {
    ...
    if {[isDeclared IMPORTANT]} {
      if {![IsImplicit IMPORTANT]} {
        if {[attrValueIMPORTANT]==YES} {
          puts stdout "NB!!IMPORTANT: " nonewline
        } else {
          puts stdout "NB!!UNIMPORTANT: " nonewline
        }
      } else {
        # IMPLIED behaviour
        puts stdout "StandardImportance: " nonewline
      }
    }
    ...
  }
  ...
}
```

This example illustrates the most general way of coding attribute access: First find out if the attribute was declared for the element, then find out if it has an explicit value, and finally use the explicit value if it exists, otherwise imply the action to be taken.

4.1.4 declaredAttrNames

The purpose of declaredAttrNames is to get a list of names of the declared attributes. declaredAttrNames doesn't take any arguments and returns the list of declared attribute names. If no declared attributes exist on the element, the empty list () is returned.

Example:

```
element ... {
  ...
  start {
    ...
    set attrlist [declaredAttrNames]
    foreach i $attrlist {
      if {[isImplicit $i]} {
        puts "$i: IMPLIED"
      } else {
        puts "$i: [attrValue$i]"
      }
    }
    ...
  }
  ...
}
```

4.2 Accessing contextual element information

4.2.1 Current element information

Information about the element is available from within the declaration of the element.

GI

The command GI returns the element's Generic Identifier of the current element. Of course, if you are specifying the actions for a TITLE element, you don't need to know what the GI of the current element (it is contextually obvious), but in default declarations it is often convenient to have access to GI information.

Example:

```
default {
  start {
    puts stdout "<[GI]" nonewline
    set attrlist [declaredAttrNames]
    foreach i $attrlist {
      if {![isImplicit]} {
        puts stdout " $i=\"[attrValue$i]\"" nonewline
      }
    }
    puts stdout ">" nonewline
  }
  end {
   puts stdout "</[GI]>" nonewline
  }
}
```

The above example is a sketch of a normalizing SGML application, which takes an SGML document instance and produces a normalized version of it, with all end tags inserted. However, much could be done to improve this sketch.

objID

objID returns the name of the object in which the statements are executed. This is most useful in situations where the object shouldn't be destroyed after being popped off the parse stack. In that case, information about the object (e.g. its objID) will usually be stored somewhere so the information in the object can be used (and the object be gracefully deleted).

Example:

```
element NOTE {
  ...
  start {
    ...
    dontDelete
    noteHandler registerNote [objID]
    ...
  }
  ...
}
```

This example sketches how an object registers itself at some object manager. The reason for registration may be that certain actions are to be taken after parsing of the document has completed. In this example this post-processing is handled by having the note element object register themselves to the object that will perform the post-processing, which will invoke appropriate methods on the objects later on.

Of course this strategy requires that the CoST memory management doesn't delete these objects. This is done by executing the dontDelete command at some point during the lifespan of the object.

4.2.2 Parent information

This section describes how to access information about elements' ancestors.

parentGI

The parentGI command returns the GI of the parent of the current element. By "parent" is meant the parent of the current element in the parse tree.

An error is produced if the element doesn't have any parent (i.e. it is the document element) .

Example:

```
element TITLE {
  ...
```

```
  start {
    ...
    case [parentGI] in {
      DOCUMENT { set out title }
      SEC1     { setout section }
      SEC2     { setout subsection }
      SEC3     { setout subsubsection }
      default  { ... }
    }
    puts stdout $out nonewline
    ...
  }
  ...
}
```

parentID

The parentID command returns the object name of the object representing the current element's parent in the parse tree. If the current element doesn't have any parent (i.e. it is document element) the empty string is returned.

Access to objects through their object names is potentially very powerful.

Example:

The example in parentGI could just as well have been implemented using parentID:

```
...
case [[parentID] GI] in {
  ...
}
```

We see in this example that GI is not only invokable from within the object; it is a general method defined on element objects that can be invoked from the outside if the object name is known.

4.2.3 Sibling information

No text available at this time.

ySibGI

No text available at this time.

ySibID

No text available.

4.2.4 in

The in command is a predicate. It takes one argument (supposedly the name of an element in the DTD), and returns 1 if the GI of the parent of the current element is equal to the argument, otherwise it returns 0.

Example:

```
element ... {
  ...
  start {
    ...
    if {[in TITLE]} {
      puts "[GI]: Parent was TITLE"
    }
    ...
  }
  ...
}
```

4.2.5 within

The within command behaves almost exactly like the in command, except that it returns 1 if the name of some ancestor of the current element is equal to the argument to within.

Example:

```
element ... {
  ...
  start {
    ...
    if {[within TABLE]} {
      set fontsize extrasmall
      ...
    }
    ...
  }
  ...
}
```

4.2.6 when

The commands in and within are useful in themselves, but often you'll want to check many contextual things in your spec, and you'll want not only to find out whether the current element is in or within some other element, but rather find out something about the element hierarchy as such. The when command can be used for this purpose.

when is very much like a case statement, its structure is like this:

```
when {
  {in MATH within TABLE} {
    <special actions taken for math in tables>
  }
  {within MATH in DISPLAYMATH} {
    <special actions taken displayed math>
  }
  default {
    <normal actions for math>
  }
}
```

The semantics of in and within in when constructs is not different from their semantics as stand-alone commands.

The semantics of the example is:

1. IF [in MATH] evaluates to 1 in the current element, AND [within TABLE] evaluates to 1 in that parent MATH element, THEN execute the special actions for math in tables;

   ELSE

2. IF [within MATH] evaluates to 1 in the current element, AND [in DISPLAY-MATH] evaluates to 1 in that ancestor MATH element, THEN execute the special acions for displayed math.

   OTHERWISE

3. Execute the normal actions for math.

It should be obvious that the when construct is very efficient in terms of how much code you need to write.

Example: Given above.

4.2.7 ancestorList

The command ancestorList returns the list of ancestors of the element object that issues the command. The parent of the object is the first member of the result list, the parent's parent is the second member, and so on.

4.3 Controlling object management

Note: The commands describe in this subsection are partly obsolete due to the introduction of more sophisticated object management features of CoST.

Under normal operation, CoST destroys all element objects after their end action has been invoked and they have been popped off the parse stack. However, in many situations it is desirable to keep some of the objects when they otherwise would be deleted, usually because they contain (or havc collected) information that is needed later on.

CoST provides a set of commands that allows the CoST programmer to tell the object manager not to delete certain objects. These commands are described in the following sections.

4.3.1 dontDelete

The dontDelete command instructs the CoST object manager not to delete the object issuing the command. The object manager won't delete the object under any circumstances (unless the object subsequently issues doDelete, see below). It is therefore the responsibility of the CoST programmer to destroy the objects that are not destroyed by the object manager. This is done with the [incr tcl] command delete (check the [incr tcl] documentation).

Example:

```
element ... {
  ...
  start {
    dontDelete
    ...
  }
  ...
}
```

4.3.2 doDelete

The doDelete command overrides an earlier dontDelete command, i.e. it is only necessary if a dontDelete was issued but the object is to be destroyed anyway. The doDelete will only have its intended effect if it is issued no later than the last statement of the end action. The doDelete command instructs the object manager to delete the object as if the dontDelete hadn't been issued. As a result doDelete won't have any effect if it is issued after the point in time when the object would have been destroyed under normal circumstances.

If the CoST programmer wants to delete an object, he can use the [incr tcl] command delete. Check the [incr tcl] documentation.

4.3.3 stopDelete

The stopDelete command instructs the CoST object manager to stop destroying objects (unlike the dontDelete command which instructs the objects not to destroy that particular object).

stopDelete can be used to make the parser build a parse tree of some part of the document. The parse tree can subsequently be navigated and information collected from the tree.

Example:

```
itcl_class any_object {
```

```
  protected nextSib

  proc register {} {
    if {[within TABLE]} {
      Queue children
      if {[ySibID]!=""} {
        [ySibID] registerNextSib[ID]
      }
      if {[parentID]!=""]} {
        [parentID] registerChild[ID]
      }
    }
  }
  method registerNextSib sib {
    set nextSib $sib
  }
  method registerChild id {
    children put $id
  }
}

default {
  inherit any_stack_object
  start {
    register
    ...
  }
  end {
    ...
  }
}

element TABLE {
  inherit any_object
  ...
  start {
    Queue children
    stopDelete
    ...
  }
  end {
    startDelete
    ...
  }
  method registerNextSib sib {}
  ...
}
```

The example illustrates how you can augment the contextual information (which by default only contains information about yonger siblings) with information that is sufficient to allow free traversal in the parse tree. In the example we assume that we need to perform some reorganization of the data in a TABLE. In the example we use default, but it could simply have been the list of elements that can appear in TABLE elements (e.g. ROW or/and COLUMN elements).

Warning: If you use specification of multiple GIs at the same time, common variables will only be common with each in element class. If all the element classes should share certain variables, use inherit instead.

4.3.4 startDelete

The startDelete command instructs the object manager to resume normal operations.

Example: See above (stopDelete).