CoST 2 – design notes

Joe English

14 Sep 1995

Contents

1 Status

(13 September 1995) This is a work in progress... pardon the rough edges. Please let me know if you have any questions or comments; thanks. -JE

2 Introduction and Hype

CoST is a general-purpose SGML post-processing tool. It is a structure-controlled SGML application; that is, it operates on the *element structure information set* (*ESIS*) representation of SGML documents.

CoST is implemented as a Tcl extension, and works in conjunction with the `sgmls` and/or `nsgmls` parsers.

CoST provides a flexible set of low-level primitives upon which sophisticated applications can be built. These include

- A powerful *query language* for navigating the document tree and extracting ESIS information;
- An *event-driven* programming interface;
- A *specification* mechanism which binds properties to nodes based on queries;

In addition, CoST includes the full power of Tcl and [incr tcl], an object-oriented extension to Tcl.

CoST is a low-level programming tool. A working knowledge of SGML, Tcl, and [incr tcl] are necessary to use it effectively.

3 Changes from the B4 release

`costsh` is a standalone process which reads the output from SGMLS; it is not a modified version of SGMLS as the B4 version was. `costsh` can be run as an interactive shell, which has proven to be very useful for debugging and for exploring the document structure.

The CoST kernel has been completely reimplemented in C, and is, except in spirit, almost completely different from the B4 release.

> I had planned to reimplement all documented facilities of the B4 release on top of the new primitives. This is turning out to be rather difficult to do, so the B4 release will still be available and maintained as a separate package.

In CoST B4, all tree nodes were represented as [incr tcl] objects. The new release stores the document internally and provides access to data through queries.

The previous version of CoST processed documents in a single pass by default, with an optional "tree mode" that allowed two passes over specific subtrees. In the new release, documents may be processed in any order with any number of passes.

The new release is considerably faster than before. It's still not *blazingly* fast, but it's reasonable. There is still room for improvement; specifications and queries could be optimized in several ways. Tcl and [incr Tcl] still seem to be the main speed bottleneck. [incr Tcl] 2.0 will reportedly be much faster than 1.5, and that should help as well.

4 Invocation

Normally, `costsh` is used in a pipeline with `sgmls`:

```
sgmls [ options ] sgml-document ... |
    costsh -S specfile [ script-options ... ]
```

The -S flag specifies that `costsh` is to operate as a *filter*: it reads a parsed document instance from standard input, then evaluates the Tcl script *specfile*. The remaining *script-options ...* are available in the global list '`argv`'. Finally, `costsh` calls the Tcl procedure `main` if one was defined in *specfile*, then exits. `main` should take zero arguments.

Calling `costsh` with no arguments starts an interactive shell:

```
costsh
```

The `loadsgmls` command reads a document into memory:

`loadsgmls` *`filehandle`*

Reads an ESIS event stream in `sgmls` format from *filehandle* and constructs the internal document tree. The current node is set to the root of the document. *filehandle* must be a Tcl file handle such as '`stdin`' or the return value of `open`.

CoST provides two convenience functions as wrappers around `loadsgmls`. '`loadfile` *`file`*' reads a pre-parsed ESIS stream from a file and is essentially the same as

```
set fp [open "\textit{filename}" r]
loadsgmls $fp
close $fp
```

`loaddoc` invokes `sgmls` as a subprocess:

`loaddoc` *`args...`*

Invokes `sgmls` with the arguments *args...* and loads the ESIS output stream. If the `$SGML_DECLARATION` environment variable is set, passes that as the first argument to `sgmls`.

5 Element Structure

In CoST, a document is represented internally as a hierarchical collection of *nodes*. Each node has an ordered list of *children*, and an unordered set of named *attributes*. Every node except the root node has a unique *parent*.

There are several types of nodes, each with a different set of characteristics:

`SD`
: An SGML document or subdocument

`EL`
: An element

`PEL`
: A "pseudo-element" or data container

`CDATA`
: A sequence of data characters (excluding record-ends)

`RE`
: A record-end character

`SDATA`
: System data, from an '`SDATA`' entity reference

`ENTREF`
: A data entity reference

`PI`
: A processing instruction

`ENTITY`
: An entity

`AT`
: An attribute or data attribute

The root node of a document is always an `SD` (SGML document or subdocument) node. Elements are represented by `EL` nodes. Data content matched by a '`#PCDATA`' content model token is represented by a `PEL` node. Collectively, these three node types are called *tree nodes*.

Sequences of characters other than record-ends are represented by `CDATA` nodes, and record-end characters appear as `RE` nodes.

> Technically, record-ends *are* character data, but it is often useful to process them separately so CoST creates distinguished nodes for them.

`PI` nodes represent processing instructions (and references to '`PI`' entities).

`SDATA` nodes represent internal system data entity references, and `ENTREF` nodes represent external data entity references. (References to other types of entities are expanded by the parser and are not directly represented as tree nodes.)

`CDATA`, `RE`, `SDATA`, and `ENTREF` nodes always appear as children of `PEL` nodes; `PI` nodes may appear anywhere in the tree.

`AT` and `ENTITY` nodes do not appear as children of any node in the tree; instead, they are accessed by name.

Node properties are accessed with *queries*.

> In the following sections, node properties are described as subcommands of the `query` command; however, they may be used wherever a query clause is appropriate. For example, '`query gi`' returns the generic identifier of the current node, and '`query parent gi`' returns the generic identifier of the current node's parent.

5.1 General properties

`query nodetype` Returns the node type of the current node.

Specific node types may be selected with the `sd`, `el`, `pel`, `cdata`, `sdata`, `re`, and `pi` query clauses. These test the type of the current node, and fail if it does not match.

5.2 Element nodes

`query? el` Tests if the current node is an `EL` node.
`query gi` Returns the generic identifier (element name) of the current node. Fails if the current node is not an `EL` node.
`query withgi` *gi* Tests if the current node is an `EL` node with generic identifier *gi*. Matching is case-insensitive.
`query element` *gi* Synonym for '`query withgi` *gi*'
`query elements "`*gi...*`"` The argument *gi...* is a space-separated list of name tokens. Succeeds if the current node's generic identifier is any one of the listed tokens. Matching is case-insensitive.

Element nodes may also have a *data content notation* or `dcn`. The DCN of an element is the value of the attribute (if any) with declared value '`NOTATION`'.

5.3 Data nodes

Data nodes are those which directly contain data. This includes `CDATA`, `SDATA`, `RE`, `PI`, and `AT` nodes (but not `PEL` nodes, which are *containers* for data nodes).

```
query content
```

Returns the character data content of the current node. For `RE` nodes, this is always a newline character ('`"\n"`'). For `SDATA` nodes it is the *system data* of the referenced entity. For `PI` nodes it is the *system data* of the processing instruction. For `AT` nodes it is the *attribute value*. Fails for all other node types.

Internal `ENTITY` *nodes?*

The `content` query clause only returns the content of data nodes. The `content` *command* returns the character data content of any node.

```
content
```

If the current node is a data node, equivalent to '`query content`'. Otherwise, equivalent to '`join [query* subtree textnode content] ""`', i.e., returns the text content of the current node.

The `textnode` clause filters out data nodes which are not part of the document's "primary content" (e.g., processing instructions). It is also useful in matching clauses in specifications.

```
query textnode
```

Tests if the current node is a `CDATA` (character data), `RE` (record end), or `SDATA` (system data) node.

5.4 Entities

```
query dataent
```

Tests if the current node is an `ENTITY` (data entity) or `ENTREF` (entity reference) node.

The difference between `ENTITY` and `ENTREF` nodes is that `ENTREF` nodes appear in the tree at the point of an entity reference, whereas `ENTITY` nodes represent the entity itself. All the properties of `ENTITY` nodes (including the content and data attributes) are accessible from the `ENTREF` nodes which reference them.

The `entity` query clause navigates directly to an `ENTITY` node:

```
query entity ename
```

Selects the `ENTITY` node corresponding to the entity named *ename* in the current subdocument, if any. The entity name is case-sensitive.

Note that `ENTITY` nodes will only be present for data entities which are referenced in the document or are named in an attribute with declared value 'ENTITY' or 'ENTITIES'.

> The parent of an `ENTITY` node is the `SD` node corresponding to the document or subdocument in which the entity was defined.

`query ename`

Returns the entity name of the current node if it is a `ENTITY` or `ENTREF` node; fails otherwise.

Note that the entity name is not available for `SDATA` nodes, only the entity text.

External entities have a *system identifier*, a *public identifier*, or both. Internal data entities have neither.

`query sysid`

If the current node is an `ENTITY` or `ENTREF`, node, returns the *system identifier* of the entity if one was declared; fails otherwise.

`query pubid`

Like `sysid` but returns the *public identifier* of the entity referenced by the current node.

The replacement text of internal entities is accessible with the `content` command.

Make sure this works for ENTREF and ENTITY nodes

External data entities (but not internal entities) also have an associated *data content notation*.

Elements (`EL` nodes) may also have a data content notation; this is determined by the value of an attribute with declared value '`NOTATION`', if one is specified for the element.

`query dcn`

Returns the name of the current node's *data content notation*, if any.

`query withdcn` *name*

Tests if the current node's data content notation is defined and is equal to *name*. Comparison is case-insensitive.

External data entities may also have *data attributes* if any are declared for the entity's associated data content notation. Data attributes are treated the same as regular attributes.

Need to add '`etype`', *entity type; returns either* '`CDATA`', '`SDATA`', '`NDATA`', *or empty string.*

Need (?) notation public / system identifiers. Need: entref command; follow ENTREFs and ENTITY attributes, implement NMLOC for EL and PEL nodes.

5.5 Attributes

`AT` nodes do not appear in the tree directly; instead, they are accessed by name from their parent node.

Only `EL` nodes and `ENTITY` nodes have attributes.

`query attval` *`attname`*

Returns the value of attribute *attname* on the current node. If the attribute has an implied value, returns the empty string. Fails if *attname* is not a declared attribute.

`query hasatt` *`attname`*

Tests if the current node has an attribute named *attname* with a non-implied value for that attribute (i.e., the attribute specified in the start-tag or a default value appeared in the '`<!ATTLIST>`' declaration).

`query withattval` *`attname value`*

Tests if the value of the attribute *attname* on the current node has the value *value*. Comparison is case-insensitive.

The `attribute` and `attlist` clauses navigate to `AT` nodes.

`query attribute` *`attname`*

Selects the attribute of the current node with name *attname*, if any.

`query attlist`

Selects each attribute of the current node.

The `content` of an `AT` node is the attribute value.

`query attname`

Returns the *attribute name* of the current node, if it is an `AT` node.

Need a way to determine if an attribute is declared but has an implied value

6 Queries

The CoST query language is used in several places:

- Accessing and testing node properties with the `query`, `query*`, and `query?` commands
- Locating nodes for processing with the `withNode` and `foreachNode` commands
- In the "match" part of *specification* clauses

CoST queries are similar to statements in Prolog, or "generators" in the Icon programming language.

6.1 Syntax

A query consists of a sequence of *clauses*. Each *clause* begins with an identifying keyword, and may contain further arguments. Clause keywords are case-insensitive. Arguments may or may not be case-sensitive depending on the clause.

```
query ::= clause [ clause ... ] ;
clause ::= keyword [ arg ...] ;
```

Note that there is no "punctuation": clauses and arguments are delimited by spaces as per the usual Tcl parsing rules. Since each clause takes a fixed number of arguments, there is no ambiguity.

Queries are evaluated from left to right, evaluating each clause in turn. Each clause is evaluated in the context of a *current node*.

Clauses may take one of four actions:

- *succeed*, possibly selecting a new current node
- *fail*, causing the query to backtrack
- *return* a value to the caller
- *abort*, signalling an error

If a clause succeeds, evaluation continues with the next clause. If it fails, evaluation *backtracks* to the previous clause, which will in turn either fail or select a new current node and continue again.

When the query is complete, the original current node is restored.

For example, the command

```
query ancestor attval "ID"
```

(which returns the 'ID' attribute value of the nearest ancestor which has one) is evaluated as follows:

- the '`ancestor`' clause passes the current node to the '`attval ID`' clause;
- if the current node has an '`ID`' attribute, `attval` returns that to the `query` command;
- otherwise, the clause fails and evaluation backtracks to the `ancestor` clause, and
- `ancestor` sets the current node to the original node's parent and continues again; then its parent's parent, and so on until the rest of the query succeeds or the root node is reached.
- If `ancestor` reaches the root node, then the whole query fails.

Note that failure does not signal an error – the `query` command just returns the empty string in this case.

6.2 Query commands

`query` *`clause...`* Evaluates the query *clause...*, and returns the first successful result. If the query fails or does not return a value, returns the empty string. `q` is a synonym for `query`.
`query?` *`clause...`* Evaluates the query *clause...*, and returns '`1`' if the query succeeds, '`0`' otherwise.
`query*` *`clause...`* Returns a list of all values produced by the query *clause...*.
`withNode` *`clause...`* `{` *`stmts`* `}` Evaluates *stmts* as a Tcl script with the current node set to the first node produced by the query *clause...*. If the query fails, does nothing.
`foreachNode` *`clause...`* `{` *`stmts`* `}` Evaluates *stmts* with the current node set to every node produced by the query *clause...* in order.

`withNode` and `foreachNode` both restore the original current node when evaluation is complete. The `selectNode` command sets the current node in the calling context:

`selectNode` *`clause...`*

Sets the current node to the first node produced by evaluating the query *clause...*.

6.3 Navigational clauses

Ancestors

`query docroot`

Selects the root node of the document.

The root node of a document is always an `SD` node. The top-level document *element* may be selected with '`query docroot child el`'.

`query doctree`

Selects every node in the document; equivalent to '`query docroot subtree`'.

`query parent`

Selects the current node's parent.

`query ancestor`

Backtracking version of `parent`: continues with the current node, the current node's parent, then the parent's parent, and so on until the root node is reached. Note that a node is considered to be an ancestor of itself.

Siblings

`query left`

Selects the current node's immediate left (preceding) sibling. Fails if the current node is the first child of its parent.

`query right`

Selects the current node's immediate right (following) sibling. Fails if the current node is the last child of its parent.

`left` and `right` only select a single node; `prev` and `next` select all of a node's siblings:

`query prev`

Selects all earlier siblings of the current node, starting with the immediate left sibling and continuing backwards to the first child.

`query next`

Selects all later siblings of the current node.

The `prev` query clause selects nodes in "reverse order"; the `esib` ("elder siblings") clause selects them in the same order as they appear in the document:

`query esib`

Selects all earlier siblings of the current node, starting with the first child node and ending with the immediate left sibling.

The `ysib` ("younger siblings") clause is present for symmetry with `esib`. It is a synonym for `next`.

`query ysib`

Selects all later siblings of the current node.

To select *all* of a node's siblings (including the node itself), use '`query parent child`'.

Descendants

`query child`

Selects all children of the current node.

`query subtree`

Selects all descendants of the current node in preorder traversal (document) order. Note that a node is considered to be a member of its subtree.

`query descendants`

Preorder traversal. This is like `subtree`, but does not include the current node.

6.4 Miscellaneous clauses

`query in` *gi*

Continues with the parent node, if the parent node has generic identifier *gi*; otherwise fails. Shorthand for '`parent withGI` *gi*'.

`query within` *gi*

Continues with the nearest enclosing parent element with the generic identifier *gi*. Equivalent to '`ancestor withGI` *gi*'.

7 Event handlers

The `process` command traverses a subtree of the source document beginning with the current node, calling a user-specified *event handler* at each element and data node. This essentially reconstructs the source ESIS event stream.

The event handler may be any Tcl command, including an [incr tcl] object or a specification command. The handler is called with one argument, which is one of the following event types:

`START`
: Invoked when entering an `EL` (element) node. Current node is the `EL` node.

`END`
: Invoked when leaving an `EL` node. Current node is the `EL` node.

`CDATA`
: Invoked for each `CDATA` (character data) node.

`RE`
: Invoked for each `RE` (record end or "newline") node.

`SDATA`
: Invoked for each `SDATA` (system data entity reference) node.

`PI`
: Invoked for each `PI` (processing instruction) node.

`DATAENT`
: Invoked for each `ENTREF` (data entity reference) or `ENTITY` (data entity) node.

Most event types correspond directly to data node types. Two events are generated for each `EL` node, one at the start of the element and one at the end. No events are generated for `PEL` nodes (instead, an event is generated for each data node child).

> `process` *cmd*
>
> Performs a preorder traversal of the current subtree, calling *cmd* for each ESIS event. *cmd* is invoked with one argument, the name of the event.

[incr tcl] classes which are to be used as event handlers should inherit from the `EventHandler` base class. This class defines a no-op method for each event type. This will ensure forward-compatibility if new event types are defined in the future.

8 Specifications

A *specification* assigns parameters to tree nodes based on queries. Specifications are named, and contain a list of queries paired with lists of name-value pairs.

```
specification specName {
    { query ... }  { name value ... }
    { query ... }  { name value ... }
    ...
}
```

Defines a new specification matching each *query* to the *name-value* pairs. Creates a Tcl access command named *specName*.

Evaluating a specification tests each query in sequence, and looks for a binding in the parameter list associated with each query that succeeds.

If a query matches but no binding is found in the associated parameter list, subsequent queries are tested.

The order of entries in a specification is significant. More specific queries should appear before more general ones. For example, '`{element P} {...}`' must appear before '`{el} {...}`' or else the '`{el}`' query will always take precedence).

> Tcl-style comments – beginning with a '`#`' and extending to the end of the line – may *not* be used inside the specification definition.

`specName has name`

Tests if there is a binding for *name* associated with the current node in *specName*. Returns '`0`' if no such binding exists, '`1`' otherwise.

`specName get name [ default ]`

Returns the *value* paired with *name* associated with the current node. If there is no such binding, then if a *default* argument was supplied, returns *default*; otherwise signals an error.

Parameter bindings may also be Tcl scripts. The `do` subcommand is a convenient way to define "methods" for document nodes.

`specName do name`

Equivalent to '`eval [specName get name ""]`' – retrieves the binding (if any) of *name* in *specName* associated with the current node and evaluates it as a Tcl expression. If no match is found, does nothing.

As a special case, '*`specName event`*' is equivalent to '*`specName`* `do` *`event`*' for each *event type* ('`START`', '`END`', '`CDATA`', etc.). This allows specifications to be used as event handlers by the `process` command.

9 Application Properties

Any document node may be annotated with application-defined *properties*. Property values are strings (like everything in Tcl), and are accessed by name, using the `setprop` command and the `propval` and `hasprop` query clauses.

`setprop` *`propname propval`*

Assigns *propval* to the property *propname* on the current node.

Property values are retrieved with queries:

`query propval` *`propname`*

Returns the value of the property *propname* on the current node; fails if no such property has been assigned.

`query hasprop` *`propname`*

Succeeds if the current node has been assigned a property named *propname*, fails otherwise.

Property names are case-sensitive. Property names beginning with a hash sign ('`#`', the SGML RNI delimiter) are reserved for internal use by CoST.

10 Miscellaneous utilities

CoST provides some extra utilities which are useful for text processing.

10.1 Environments

Tcl provides very primitive variable scoping mechanisms: a variable can only be global or local to a single procedure. CoST includes a facility for multiple *dynamic environments.*

An *environment* is a set of name-value pairs, much like an associative array. Bindings may be saved and restored dynamically, similar to TeX's grouping mechanism. In CoST, it is possible to create multiple independent environments.

`environment` *`envname`* `[` *`name value`* `...]`
Creates a new environment and a Tcl command named *envname* to access it. The optional *name* and *value* argument pairs define initial bindings in the environment.

Environment commands have several subcommands:

`envname` `set` *`name value`* `[` *`name value`*`... ]`
Adds the *name-value* pairs to the environment *envname*, overwriting the current binding of each *name* if it is already present.

`envname` `get` *`name`* `[` *`default`* `]`
Returns the value currently bound to *name* in the environment *envname*. If no binding for *name* currently exists in *envname* and the *default* argument is present, returns that instead; otherwise signals an error.

`envname` `save [` *`name value`* `... ]`
Saves the current set of name-value bindings in *envname*. If *name* and *value* argument pairs are supplied, adds new bindings to the environment after saving the current bindings.

`envname` `restore`
Restores the bindings in *envname* to their settings at the time of the last call to '*`envname`* `save`'.

Environments are useful in conjunction with event handlers which need to maintain a "stack" of information, where properties of a node should propagate to its children but not affect its younger siblings or ancestors. Typically an event handler will call `save` for each '`START`' event and `restore` for each '`END`' event.

10.2 Substitutions

When writing translation utilities to convert SGML from one format to another, it is often necessary to "escape" or translate character data that might be interpreted as markup in the result language. For example, translation to HTML requires all occurrences of '<', '>' and '&' to be replaced with entity references '`<`', '`>`' and '`&`'. TeX and LaTeX have many special characters which need to be "escaped".

The `substitution` command provides an easy and efficient way to apply fixed-string substitutions.

`substitution` *substName pairs*

pairs is a list of alternating keys and values. Defines a new command *substName* that, when applied to a string argument, returns a new string with each occurrence of any key in *pairs* replaced with the corresponding value. If multiple keys match a substring, the earliest and longest match takes precedence.

```
substitution htmlescape {
        {<} {&lt;}
        {>} {&gt;}
        {&} {&amp;}
        {<=} {&le;}
        {>=} {&ge;}
}
htmlescape "a < b && b > c"
# returns "a &lt; b &amp;&amp; b &gt; c"
```

11 Examples

Many of these examples are based on HTML; some familiarity with that document type is assumed.

Need to explain the HTML features that are used.

11.1 Query examples

Here is a simple query which returns a list of all of the hyperlinks (HREF attribute values) in an HTML document:

```
query* doctree element A attval HREF
```

A slightly better version is:

```
query* doctree element A hasatt HREF attval HREF
```

The 'hasatt HREF' clause selects only those elements which have a non-implicit HREF attribute; without this clause, the list will contain empty members for each A element which is a destination anchor (<A NAME=...> instead of <A HREF=...>).

The next example builds a cross-reference list: it prints the anchor name and text of each destination anchor, and the target URL of each source anchor. It uses two foreachNode passes to traverse the document.

```
puts stdout "Destination anchors:"
foreachNode doctree element A hasatt NAME {
    puts stdout "\t#[query attval NAME]: [content]"
}
puts stdout "Source anchors:"
foreachNode doctree element A hasatt HREF {
    puts stdout "\tURL:[query attval HREF]: [content]"
}
```

A similar listing could also be produced with an event-driven specification:

```
specification printAnchors {
    {element A withatt HREF} {
        START   { puts stdout "<URL:[query attval HREF]>: " nonewline }
    }
    {element A withatt NAME} {
        START   { puts stdout "Anchor #[query attval NAME]: " nonewline }
    }
    {element A} {
        END     { puts "" }
    }
    {textnode within A} {
        CDATA   { puts stdout [query content] nonewline }
        RE      { puts stdout " " nonewline }
    }
}
```

This would be invoked as:

```
withNode docroot { process printAnchors }
```

11.2 A spell-checker

The Unix utility `ispell` is programmed to understand LaTeX and nroff markup, but it doesn't know anything about SGML. If you've ever tried to run `ispell` on an SGML document, you've probably noticed that it doesn't do a very good job, since it tries to "correct" the spelling of all the tags and other markup.

This example, implemented as an [incr tcl] event handler, simply extracts the character data from the input document and filters it through `ispell`, producing a list of potentially misspelled words on standard output.

It's not as fancy as the interactive `ispell` mode, but it works well enough. It has one extra feature which is very useful for technical documentation, though: you can specify a list of elements which should *not* be spell-checked.

The SpellChecker event handler class works with any document type. It recognizes one processing instruction: '<?spelling ...>' will add the listed words to the private dictionary. For example, this document contains '<?spelling HyTime CoST LaTeX>'.

Here is the main specification, used to spell-check this document:

```
require specs/Spell.tcl

SpellChecker spellChecker \
    -suppress "AUTHOR DATE EDNOTE LISTING EXAMPLE SYNOPSIS LIT SAMP VAR
       ATTR CLASS CMD ELEM ENV EVENT NODETYPE QC SUBCMD TAG
       ARG OPTARG OPTION"

proc main {} {
    spellChecker begin
    process spellChecker
    spellChecker end
}
```

And here is the implementation of the SpellChecker class:

```
# Spell.tcl
# CoST wrapper around 'ispell'

itcl_class SpellChecker {
    inherit EventHandler;

    public dictfile ""
    public suppress ""; # list of elements not to spellcheck
    public tmpfile "/tmp/costspell.tmp";

    protected ispellpipe; # pipe to 'ispell' process
    protected suppressing 0;
    protected wordlist "";# extra words for dictionary

    constructor {config} {
       # make sure suppress GI list is all uppercase
       set suppress [string toupper $suppress]
    }

    method suppress? {} {
       # suppress checking for current element?
       return [expr [lsearch $suppress [query gi]] != -1]
    }
```

```
    # START and END tag handlers just set the 'suppressing' flag.
    method START {} {
       if [suppress?] { incr suppressing }
    }
    method END {} {
       if [suppress?] { incr suppressing -1 }
       # make sure there's whitespace between element boundaries:
       puts $ispellpipe ""
    }

    method CDATA {} {
       if !$suppressing { puts $ispellpipe [content] }
    }

    method PI {} {
       # Is this a <?spelling ...> instruction?
       if {[lindex [query content] 0] == "spelling"} {
           # Yep; add to word list
           append wordlist " [lrange [query content] 1 end]"
       }
    }

    method begin {} {
       set cmd "ispell -l"
       if {$dictfile != ""} {
           append cmd " -p $dictfile"
       }
       set ispellpipe [open "|$cmd | sort | uniq > $tmpfile" w]
       set suppressing 0;
    }

    method end {} {
       close $ispellpipe
       # Read words back from temporary file
       set fp [open $tmpfile r]
       while {[gets $fp word] > 0} {
           # see if it's in local dictionary:
           if {[lsearch $wordlist $word] == -1} {
       # nope; report it:
       puts stdout $word
           }
       }
       close $fp
    }
}
```

11.3 Outline and index

This is a utility which I've found useful in preparing this reference manual: it builds an outline from the section titles, and produces a command and query clause index.

```
#
# outline.spec
# Build a table of contents and command/query clause index
# from the main document.
#

proc main {args} {
#
# The first pass prints the table of contents
# and annotates each SECT element with properties
# that are used in the second phase:
#
    process outline
    nl; nl;
#
# The second phase builds and prints an index of each command
# (CMD elements) and query clause (QC elements) used in the document,
# including the section number(s) where they appear.
#
    puts stdout "Commands:"
    listall CMD
    nl;
    puts stdout "Query clauses:"
    listall QC
}


#
# Pass 1: table of contents
# Lists all <SECT>ion <TITLE>s and <H>eadings,
# assigning section numbers on the way.
#
global secdepth ;# current nesting level
global secctrs ;# array: secdepth -> current section number

set secdepth 1
set secctrs($secdepth) 0

specification outline {
```

```
    {element SECT} {
       START {
           global secdepth secctrs
           incr secctrs($secdepth)

           # Set node properties:
           setprop secdepth $secdepth
           setprop secctr $secctrs($secdepth)
           set secnumbers [query* ancestor propval secctr]
           setprop secnum [join [lreverse $secnumbers] "." ]

           # Setup for subsections:
           incr secdepth
           set secctrs($secdepth) 0
       }
       END {
           incr secdepth -1
       }
    }

    {textnode within TITLE} {
       CDATA { output [content] }
    }
    {element TITLE} {
       START {
           global secdepth
           indent $secdepth
           output "[query parent propval secnum] "
       }
       END {
           nl
       }
    }

    {textnode within H} {
       CDATA { output [content] }
    }
    {element H} {
       START { indent [expr $secdepth + 1] }
       END { nl }
    }

}


#
```

```
# Utility for pass 2:
# build and print an index of terms;
# 'gi' is the generic identifier of the element to be indexed.
#

proc listall {gi} {
    foreachNode doctree element $gi {
        set term [content]
        set where [query ancestor propval secnum]
        lappend tindex($term) $where
    }

    foreach term [lsort [array names tindex]] {
        set tindex($term) [luniq $tindex($term)]
        indent 1
        output "$term ([join $tindex($term) ", "])"
        nl
    }
}

#
# Miscellaneous utilities:
#
proc output {data} { puts stdout $data nonewline; }
proc nl {} { puts stdout ""; }
proc indent {n} {
    while {$n > 0} {
        output "    ";
        incr n -1;
    }
}
```

Explain how this works; recursive section structure.

11.4 LaTeX conversion

This is a comprehensive example which uses all of the core CoST features.

Explain how it works; split in two parts, generic "architectural form" processor implented as [incr tcl] event handler, specification mapping elements to forms and supplying form parameters. uses substitutions, environments, specification, event handlers.

```
require specs/Latex.tcl
```

```
LatexProcess tolatex -specification {

    {element COSTDOC} {
       arcform document
       documentclassarticle
       classoptions{}
       packages{t1enc alltt myfmt}
       preamble{ [doPreamble] }
       prefix {
       \maketitle\tableofcontents
       }
    }

    {element P} { arcform paragraph }
    {element SECT} { arcform division }

    {element TITLE in SECT} {
       arcform divheading
    }
    {element H} {
       arcform macro
       macro "paragraph*"
    }

    {elements "EMPH DFN DPH"} {
       arcform macro
       macro "emph"
    }

    {elements "CMD SUBCMD QC ENV NODETYPE EVENT CLASS ATTR ELEM"} {
       arcform macro
       macro "texttt"
    }

    {element TAG} {
       arcform macro
       macro texttt
       prefix {<}
       suffix {>}
    }

    {element ENVAR} {
       arcform macro
       macro texttt
       prefix {\$}
```

```
}

{elements "OPTION OPTARG ARG"} { arcform macro }
{element OPTION} { macro textbf }
{element OPTARG} { macro textit }
{element ARG} { macro textit }

{elements "SAMP LIT"} {
   arcform macro
   macro "texttt"
   before "‘"
   after "’"
}

{element CMDDEF} {
   arcform environment
   envname "framed"
}

{elements "TIP NOTE"} {
   arcform environment
   envname "quote"
}

{element EDNOTE} {
   arcform environment
   envname ednote
}

{element DLIST} {
   arcform list
   envname "description"
}
{element DT} {
   arcform itemtag
}
{element DD} {
   arcform itembody
}

{element LIST} {
   arcform list
   envname "itemize"
}
{element IT} {
   arcform item
```

```
    }

    {element SYNOPSIS} {
       arcform verbatim
       envname alltt
       cdataFilter allttFilter
    }

    {elements "EXAMPLE LISTING" hasatt SOURCE} {
       arcform special
       before {\begin{verbatim}}
       content {[readFile [query entity [query attval SOURCE] sysid]]}
       suffix {\end}
       after  {{verbatim}}
    }

    {elements "EXAMPLE LISTING"} {
       arcform verbatim
    }

    {elements "HEAD TITLEPG"} { arcform ignore }
}

proc main {} { process tolatex }

proc doPreamble {} {
    set txt ""
    withNode docroot subtree element HEAD {
       withNode child element title  { append txt "\\title{[content]}\n" }
       withNode child element author { append txt "\\author{[content]}\n" }
       withNode child element date   { append txt "\\date{[content]}\n" }
    }
    return $txt
}

proc readFile {filename} {
    if [catch {set fp [open "|newform -i8 -o0 $filename" r]}] {
       warning "Can't load $filename"
       return;
    }
    set content [read $fp]
    close $fp
    return $content
}

substitution allttFilter {
```

```
    "\{""\\\{"
    "\}""\\\}"
    "\\""\\char\"5C{}"
}

# Latex.tcl
# CoST SGML->LaTeX (2e) conversion process utilities
#

# Substitution for text mode:
# %%% This only works with T1 encoding
# %%% Trying to do this with OT1 is too horrifying to think about
substitution latexEscape {
       "\{""\\\{"
       "\}""\\\}"
       "[""\{[\}"
       "]""\{]\}"
       "\\""\\char\"5C{}"
       "#""\\#"
       "$""\\char`$"
       "%""\\%"
       "&""\\char`&"
       "~""$\\sim$"
       "_""\\char`_"
       "^""\\char`^"
}

proc identityFilter {str} { return $str }
proc nullFilter {str} { return "" }
proc sdataFilter {str} { return $str }

# filter for 'verbatim' environment: expand tabs,
# don't substitute anything else.
# %%% this doesn't work: subelements will throw off column count
proc tabexFilter {line} {
    if {[string first "\t" $line] == -1} {return $line}
    set spaces 0
    set tabstop 8
    set result ""
    set col 0
    foreach chunk [split $line "\t"] {
       incr col $spaces
       while {$spaces > 0} { append result " "; incr spaces -1 }
       append result $chunk
       incr col [string length $chunk]
       set spaces [expr $tabstop - $col % $tabstop]
```

```
    }
    return $result
}

#
# Event handler class:
#
itcl_class LatexProcess {
    inherit EventHandler
    protected env
    protected spec ""
    protected undefinedGIs ""
    protected blanklines -1
    protected seclevel -1

    public specname
    public specification  ""
    public verbose 0
    public secheadings {section subsection subsubsection paragraph}
    public preamble ""; # extra preamble, before \begin{document}

    constructor {config} {
       set env [environment "${this}Env" \
       cdataFilter latexEscape \
       sdataFilter sdataFilter \
       markupFilter identityFilter \
       suppress 0 \
       ]
       if [info exists specname] {
           set spec $specname
       } else {
           set spec [specification ${this}Assoc $specification]
       }
    }

#
# Output methods:
# Blank lines are significant in TeX (they cause a paragraph break),
# so these methods keep track of newlines.
# Use 'output' for character data (escaped), 'insert' for LaTeX markup.
# 'nl n' appends 'n' blank lines; if n==0, just break the line.
# NB: 'output' assumes that its argument contains no newlines.
#
    method output {text} {
       if {$text != ""} {
           puts stdout $text nonewline
```

```
            set blanklines -1
        }
    }

    method nl {{n 0}} {
        while {$blanklines < $n} {
            puts stdout "";
            incr blanklines
        }
    }

    method insert {text} {
        output [[$env get markupFilter] $text]
    }

    method insertFile {filename} {
        if [catch {set fp [open $filename r]}] {
            warning "Cannot open $filename"
            return;
        }
        output [[$env get cdataFilter] [read $fp]]
        close $fp
    }

#
# Event handler methods:
#
    method CDATA {} { output [[$env get cdataFilter] [content]] }
    method SDATA {} { output [[$env get sdataFilter] [content]] }
    method RE {} { nl 0 }
    method DATAENT {} { specialStart }

#
# START and END methods just call the architectural form handlers,
# inserting the 'before', 'after', 'prefix' and 'suffix' parameters verbatim
#
    method START {} {
        insert [$spec get before ""]
        "[$spec get arcform undefined]Start"
        insert [$spec get prefix ""]
    }

    method END {} {
        insert [$spec get suffix ""]
        "[$spec get arcform undefined]End"
        insert [$spec get after ""]
```

```
    }

#
# LaTeX "architectural form" processing:
# New forms may be defined by subclassing the LatexProcess class,
# defining xxxStart and xxxEnd methods for each new form 'xxx'
#

    # none: no special processing
    method noneStart {} { }
    method noneEnd {} { }

    # ignore: supress content of element
    method ignoreStart {} {
       output [subst [$spec get content ""]]
       $env bgroup cdataFilter nullFilter \
           sdataFilter nullFilter \
           markupFilter nullFilter \
           suppress 1
    }
    method ignoreEnd {} {
       $env egroup
    }

    # undefined: called as a default if no form specified.
    method undefinedStart {} {
       if ![$env get suppress] {
           set gi [query gi]
           if { [lsearch $undefinedGIs $gi] == -1 } {
       lappend undefinedGIs $gi
       puts stderr "No mapping for $gi element"
           }
       }
    }
    method undefinedEnd {} { }
    method queryUndefined {} { return $undefinedGIs }
    method resetUndefined {} { set undefinedGIs "" }

    # special: special processing
    # parameters:
    #  content:content of element.  Evaluated with Tcl 'subst' cmd.
    method specialStart {} {
       output [subst [$spec get content ""]]
    }
    method specialEnd {} { }
```

```
# document: document body
# parameters:
#  documentclass:LaTeX document class
#  classoptions:list of class options
#  packages:list of packages (.sty files) used
#
method documentStart {} {
   set class [$spec get documentclass article]
   set options [$spec get classoptions ""]
   if {$options != ""} { set options "\[$options\]" }
   output "% Generated by CoST"; nl
   output "\\documentclass${options}{$class}"; nl
   foreach package [$spec get packages {}] {
       output "\\usepackage{$package}"; nl
   }
   nl; insert $preamble; nl;
   insert [subst [$spec get preamble ""]]; nl;
   output "\\begin{document}"; nl
}
method documentEnd {} {
   nl; output "\\end{document}"; nl
}

# %%% Need: author, title, date, other metainfo

# division: hierarchical division (section, chapter, etc.)
# In LaTeX, sections are specified by heading elements, not containers.
# The actual heading is produced by the 'divheading' form element
method divisionStart {} {
   nl 1
   incr seclevel;
   setprop seclevel $seclevel
   if [query? hasatt ID] { setprop reflabel [query attval ID] }
}
method divisionEnd {} {
   nl 1
   incr seclevel -1
}

# divheading: hierarchical division title
# parameters:
#  macro:macro name for division heading;
#  defaultdetermined by section level and $secheadings
method divheadingStart {} {
   if [$spec has macro ] {
       macroStart
```

```
    } else {
        insert "\\[lindex $secheadings $seclevel]\{"
    }
}
method divheadingEnd {} {
    if [$spec has macro] {
        macroEnd
    } else {
        insert "\}"
    }
    if [query? parent hasprop reflabel] {
        insert " \\label{[query parent propval reflabel]}"
    }
    nl;
}

# environment: maps to  \begin{xxx} ... \end{xxx}
# parameters:
#  envname:name of environment
#  envargs:extra arguments to \begin{envname}, incl. {}[]s.
method environmentStart {} {
    nl;
    insert "\\begin\{[$spec get envname]\}[$spec get envargs ""]"
    nl 0
}
method environmentEnd {} {
    nl; insert "\\end\{[$spec get envname]\}"; nl;
}

# macro: maps to \xxx{ ... }
# parameters:
#  macro:control sequence name (not including \)
method macroStart {} { insert "\\[$spec get macro]\{" }
method macroEnd {} { insert "\}" }

# paragraph
method paragraphStart {} { nl 1 }
method paragraphEnd {} { nl 1 }

# verbatim: special case of 'environment';
# does not escape special characters in content.
# parameters:
#  cdataFilter:content filter procedure (default: tabexFilter)
#  envname:environment name (default: verbatim)
method verbatimStart {} {
    nl; insert "\\begin{[$spec get envname verbatim]}"; nl;
```

```
    # should markupFilter == nullFilter?
    $env bgroup cdataFilter [$spec get cdataFilter tabexFilter] \
    sdataFilter nullFilter \
    markupFilter identityFilter
}
method verbatimEnd {} {
    $env egroup
    nl; insert "\\end{[$spec get envname verbatim]}"; nl
}

# lists: special case of environment.
# should only contain item, itemtag, and itembody form elements.
method listStart {} { environmentStart }
method listEnd {} { environmentEnd }

# item: members of single-part lists like itemize and enumerate
method itemStart {} { nl; insert "\\item{}"; nl }
method itemEnd {}  { nl 1 }

# itemtag/itembody: members of two-part lists like description
# should occur in pairs.
method itemtagStart {} { nl; insert "\\item\[" }
method itemtagEnd {}  { insert "\]" }
method itembodyStart {} { }
method itembodyEnd {}  { }

# xref: cross-references.
# Don't ask.
method xrefStart {} {
    set linkend [query attval REFID];
    if {$linkend == ""} {
        puts stderr "XREF has no linkend"
    }
    set reflabel ""
    set refprefix ""

    withNode doctree withattval ID $linkend {
        set refprefix [$spec get refprefix [query propval refprefix]]
        set reftype [$spec get reftype [query propval reftype]]
        set reflabel [query propval reflabel]
    }
    set refprefix [$spec get refprefix $refprefix]
    set reftype [$spec get reftype $reftype]
    if {$reflabel == ""} { set reflabel $linkend }

    if {$reflabel == ""} {
```

```
                puts stderr "Warning: XREF to $linkend has no label"
                return;
            }

             insert "$refprefix\\$reftype{$reflabel}"
        }

        method xrefEnd {} { }

    }
```