Cost 2 Reference Manual

Joe English

Version 2.0a3, 16 Jan 1996

Contents

1 Introduction

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;

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

2 Running Cost

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 Tcl command `loadsgmls` 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 "filename" r]
loadsgmls $fp
close $fp
```

`loaddoc` invokes `sgmls` as a subprocess:

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

3 Getting Started

> Cost is a powerful but somewhat complex system. The `Simple` module provides a simplified, high-level interface for developing translation specifications.

A large number of SGML translation tasks involve nothing more than

- inserting some text around each element,
- replacing each `SDATA` entity reference with a suitable output format representation, and
- "escaping" certain characters or sequences of characters that might be interpreted as markup in the target language.

The `Simple` module is designed to handle these types of translations. It makes a single pass through the document, inserting text and optionally calling a user-specified script at the beginning and end of each element. The translated document is written to standard output.

To load this module, put the command

```
require Simple.tcl
```

at the beginning of the specification script. Next, define a *translation specification* as follows:

```
specification translate {
    specification-rules...
}
```

The *specification-rules* is a paired list matching *queries* with *parameter lists*. The queries are used to select elements, and are typically of the form

```
{element GI}
```

or

```
{elements "GI GI..."}
```

where each *GI* is the generic identifier or element type name of the elements to select.

Any Cost query may be used, including complex rules like

```
{element TITLE in SECTION withattval SECURITY RESTRICTED}
```

or simple ones like

```
{el}
```

The latter query – '`el`' – matches all element nodes; it can be used to specify default parameters for elements which don't match any earlier query.

The parameter lists are also paired lists, matching *parameters* to *values*. The `Simple` module translation process uses the following parameters:

`startAction`
: Tcl statements to execute at the beginning of the element

`endAction`
: Tcl statements to execute at the end of the element

`before`
: Text to insert before the element (before evaluating `startAction`)

`prefix`
: Text to insert at the beginning the element (after evaluating `startAction`)

`suffix`
: Text to insert at the end of the element (before evaluating `endAction`)

`after`
: Text to insert after the element (after evaluating `endAction`)

`cdataFilter`
: A *filter procedure* for character data

`sdataFilter`
: A filter procedure for system data (`SDATA` entity references).

Tcl variable, backslash, and command substitution are performed on the `before`, `after`, `prefix`, and `suffix` parameters. This takes place when the element is processed, not when the specification is defined. The value of these parameters are *not* passed through the `cdataFilter` command before being output.

> Remember to "protect" all Tcl special characters by prefixing them with a backslash if they are to appear in the output. The special characters are: dollar signs `$`, square brackets `[]`, and backslashes `\`. See the Tcl documentation on the `subst` command for more details.

The `cdataFilter` parameter is the name of a *filter procedure*. This is a one-argument Tcl command. Cost passes each chunk of character data to this procedure, and outputs whatever the procedure returns. The default value of `cdataFilter` is the `identity` command, which simply returns its input:

```
proc identity {text} {return $text}
```

The `sdataFilter` parameter works just like `cdataFilter`, except that it is used for *system data* (the replacement text of `SDATA` entity references.) The default `sdataFilter` is also `identity`.

The `Simple` module saves and restores the current `cdataFilter` and `sdataFilter` at each element node.

Example

The following specification translates a subset of HTML to nroff -man macros. (Well, actually it doesn't do anything useful, it's just to give an idea of the syntax.)

```
require Simple.tcl
specification translate {
	{element H1} {
		prefix		"\n.SH "
		suffix		"\n"
		cdataFilter		uppercase
	}
	{element H2} {
		prefix		"\n.SS "
		suffix		"\n"
	}
	{elements "H3 H4 H5 H6"} {
		prefix "\n.SS"
		suffix "\n"
		startAction {
		    # nroff -man only has two heading levels
		    puts stderr "Mapping [query gi] to second-level heading"
		}
	}
	{element DT} {
		prefix		"\n.IP \""
		suffix		"\"\n"
```

```
        }
        {element PRE} {
                prefix "\n.nf\n"
                suffix "\n.fi\n"
        }
        {elements "EM I"} {
                prefix "\\fI"
                suffix "\\fP"
        }
        {elements "STRONG B"} {
                prefix "\\fB"
                suffix "\\fP"
        }
        {element HEAD} {
                cdataFilter nullFilter
        }
        {element BODY} {
                cdataFilter nroffEscape
        }
}
proc nullFilter {text} {
    return ""
}
proc nroffEscape {text} {
    # change backslashes to '\e'
    regsub -all {\\} $text {\\e} output
    return $output
}
proc uppercase {text} {
    return [nroffEscape [string toupper $text]]
}
```

Notes

The specification order is important: queries are tested in the order specified, so more specific queries must appear before more general ones.

Parameters are evaluated independently of one another. For example,

```
specification translate {
    {element "TITLE"} {
        cdataFilter uppercase
    }
    {element TITLE in SECT in SECT in SECT} {
        prefix "<H3>"
        suffix "</H3>\n"
```

```
    }
    {element TITLE in SECT in SECT} {
        prefix "<H2>"
        suffix "</H2>\n"
    }
    {element TITLE in SECT} {
        prefix "<H1>"
        suffix "</H1>\n"
        startAction {
            puts $tocfile [content]
        }
    }
}
```

The parameter '`cdataFilter uppercase`' applies to all `TITLE` elements, regardless of where they occur, and the '`startAction`' parameter applies to any `TITLE`s which are children of a `SECT`, even if an earlier matching rule specified a '`prefix`' or '`suffix`'.

As its name implies, the `Simple` module is not very sophisticated, but it should be enough to get you started. To do more powerful things with Cost, read on...

4 Element Structure

An SGML document is represented in Cost 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` 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 handle 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.

4.1 General properties

`query nodetype`

Returns the node type of the current node (`SD`, `EL`, `PEL`, et cetera).

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.

4.2 Element nodes

`query? el`

Tests if the current node is an `EL` node.

```
query gi
```

Returns the generic identifier (element type 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 `dcn` (*data content notation*) property. The DCN of an element is the value of the attribute (if any) with declared value `NOTATION`.

4.3 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.

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).

```
query textnode
```

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

4.4 Entities

```
query dataent
```

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

`ENTREF` nodes appear in the document tree at the point of a data entity reference. `ENTITY` nodes represent the entity itself and do not appear as children of any tree node.

All properties of `ENTITY` nodes (including their content and data attributes) are accessible from `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.

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

```
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.

The `content` command returns the replacement text of internal data entity nodes.

External entities have a *system identifier*, a *public identifier*, or both.

```
query sysid
```

Returns the *system identifier* of the entity referenced by the current node if one was declared; fails otherwise.

```
query pubid
```

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

External data entities 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 accessed in the same way as regular attributes.

4.5 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 of the current node.

```
query hasatt attname
```

Tests if the current node has an attribute named *attname* with a non-implied value (i.e., the attribute was 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 named *attname* of the current node. Fails if no such attribute is present.

```
query attlist
```

Selects each attribute (`AT` node) of the current node, in an unspecified order.

```
query attname
```

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

The `content` query clause returns the attribute value of the current node if it is an `AT` node.

5 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 Prolog statements or "generators" in the Icon programming language.

5.1 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"
```

is evaluated as follows:

- the current node becomes the *source node* for the '`ancestor`' clause;
- the '`ancestor`' clause passes the source 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 source 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.

5.2 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. `q?` is a synonym for `query?`.

`query*` *`clause...`*

Returns a Tcl list of all values produced by the query *clause...*. `q*` is a synonym for `query*`.

`countq` *`clause...`*

Returns the number of nodes selected or results returned 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. The Tcl `break` and `continue` commands exit the loop and continue with the next selected node, respectively.

`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...*.

5.3 Navigational clauses

Ancestors

`query parent`

Selects the source node's parent.

`query ancestor`

Selects all ancestors of the source node, beginning with the source node and ending with the root node.

`query rootpath`

Selects all ancestors of the source node, beginning with the root node and ending with the source node.

Note that a node is considered to be an ancestor of itself.

Siblings

`query left`

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

`query right`

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

`left` and `right` only select a single node. `prev` and `next` select multiple siblings:

`query prev`

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

`query next`

Selects all later siblings of the source 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 source 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 source 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 source node in order.

`query subtree`

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

`query descendant`

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

5.4 Addressing

Every tree node (`EL` and `PEL` nodes) has a unique *node address.* This is an opaque string by which the node may be referenced.

`query address`

Returns the node address of the current node. Fails if the current node is not a tree node.

`query node` *`addr`*

Selects the node whose address is *addr.*

`query nodes` *`addrlist`*

addrlist is a space-separated list of node addresses as returned by `addresses`. Selects each node in *addrlist*, in the order they appear in the list.

5.5 Miscellaneous clauses

```
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 in gi
```

Selects the parent node if it is an `EL` node with generic identifier *gi*, fails otherwise. Shorthand for '`parent withGI gi`'.

```
query within gi
```

Selects all ancestor `EL` nodes with generic identifier *gi*. Equivalent to '`ancestor withGI gi`'.

6 Event handlers

Cost supports an *event-driven* processing model. This essentially reconstructs the source ESIS event stream for a particular subtree.

Tree traversal procedures are defined with the `eventHandler` command.

```
eventHandler -global name {
    event { script }
    event { script }
    ...
}
```

Defines a new *traversal procedure* named *name* which, when invoked, traverses the subtree rooted at the current node and evaluates the specified *script* for each ESIS event *event*. Ignores events for which no *script* is defined. If **-global** is specified, the *script*s are evaluated in the top-level Tcl environment; otherwise they are evaluated in the calling context. If any *script* calls the Tcl `break` command, stops the traversal.

The following events are generated:

`START`
: Invoked when entering an `EL` (element) node. The current node is set to the `EL` node.

`END`
: Invoked when leaving an `EL` node. The current node is set to 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 (events are generated for each data node child, however).

`process` *cmd*
Performs a preorder traversal of the subtree rooted at the current node, calling *cmd* for each ESIS event. *cmd* is invoked with one argument, the name of the event, with the *current node* set to the active node.

The `process` command traverses the tree and calls a user-specified event handler procedure at each event. 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 the name of the event.

[incr tcl] classes which are to be used as event handlers should inherit from the `EventHandler` base class, which defines a do-nothing method for each event type.

Example

```
# File: printtree.spec
# Sample event handler
# Prints an indented listing of the tree structure

global level; set level 0

proc main {} { printtree }

eventHandler printtree -global {
    START {
        indent $level;
        puts "<[query gi]>";
        incr level;
    }
    END {
        incr level -1;
```

```
        indent $level;
        puts "</[query gi]>";
    }
    CDATA   { indent $level; puts "\"[query content]\"" }
    SDATA   { indent $level; puts "|[query content]|" }
    RE      { #indent $level; puts "RE" }
    DATAENT { indent $level; puts "&[query ename];" }
}

proc indent {n} {
    while {$n > 0} { puts stdout "   " nonewline; incr n -1 }
}
```

7 Specifications

Specifications assign parameters to document nodes based on queries.

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

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

Evaluating a specification tests each *query* in sequence, and looks for a matching *name* in the parameter list associated with every query that succeeds. Comparison is case-sensitive. All the *name*s in a single parameter list must be unique.

```
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 in *specName*. 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 specification commands to be used as event handlers by the `process` command.

The order of entries in a specification is significant. More specific queries should appear before more general ones. For example, '`{element P withattval SECURITY TOP} {hide=1}`' must appear before '`{element P} {hide=0}`' or else the '`{hide=0}`' binding will always take precedence.

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

8 Application Properties

Document nodes may be annotated with application-defined *properties*. Property values are strings (like everything in Tcl), and are accessed by name.

`setprop propname propval`

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

`unsetprop propname [ propname ... ]`

Removes the properties *propname...* on the current node. It is not an error if any of the *propnames* are not currently set.

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.

`query withpropval propname propval`

Succeeds if the current node has a *propname* property with value *propval*. The value comparison is case sensitive.

Property names are case-sensitive.

Property names beginning with a hash sign (#, the SGML RNI delimiter) are reserved for internal use by Cost.

9 Links and relations

> This facility is still experimental and subject to change.

Links and relations provide a way to correlate arbitrary tree nodes.

An *ilink* is a collection of one or more named *anchors*. Each *anchor* is a reference to a node in the tree. Ilinks also have an *origin* node; this is the node which was current when the ilink was created. Every ilink belongs to a named *relation*; all ilinks in the same relation have the same structure (number and names of anchors).

Ilinks are stored as nodes in the document tree. They are accessed by queries and may be assigned properties just like other nodes.

The `relation` and `addlink` commands create a relations and ilinks. Relations must be created before ilinks are added.

```
relation relname  \
        anchname1 [ anchname2 ... anchnameN ]
```

Creates a new relation named *relname*, with anchors named *anchname1* ... *anchnameN*.

```
addlink relname [ anchname "query" ... ]
```

Adds a new `ILINK` node to the relation *relname*. The ilink's origin is set to the current node. A *query* must be specified for each anchor name *anchname* in the relation. The anchor's endpoint is set to the first node produced by the query. If the query fails, then the anchor is not created. Each *query* is evaluated with the newly created `ILINK` node as the source node.

Anchors are created in the order specified. The query clause for an anchor may refer to previously created anchors or to the ilink's `origin`.

For example,

```
# create a new relation with three anchors:
relation crossref source target targetsection
# create links:
foreachNode doctree element XREF {
    set refid [query attval REFID]
    addlink crossref \
            source "origin" \
            target "doctree el withattval ID $refid" \
            targetsection "anchor target ancestor element SECT"
}
```

Once ilinks are created, they may not be removed or changed.

The `ilink` and `anchor` query clauses navigate to and from `ILINK` nodes:

`query ilink` *relname srcanch*
Selects each `ILINK` in the relation *relname* in which the anchor named by *srcanch* refers to the current node.

`query anchor` *dstanch*
The current node must be an `ILINK` node. Selects the node referenced by the *dstanch* anchor.

`query origin`
The current node must be an `ILINK` node. Selects the ilink node's origin node.

For example,

```
foreachNode doctree element XREF {
    puts [query ilink CROSSREF SOURCE anchor TARGET  propval title]
}
```

The clause '`ilink crossref source`' selects the `ILINK` nodes in the `crossref` relation having the current node as their `source` anchor. The clause '`anchor target`' traverses to the `target` anchor, and the query returns the value of that node's `title` property.

The `anchtrav` query clause navigates across ilinks; it combines the `ilink` and `anchor` clauses into one step.

`query anchtrav` *relname srcanch dstanch*
Selects the target node of the *dstanch* anchor in every ilink in the relation *relname* in which the source node is the *srcanch* anchor.

```
foreachNode doctree element XREF {
    puts [query anchtrav CROSSREF SOURCE TARGET  propval title]
}
```

Ilinks may be accessed independently of any of their anchors:

`query relation` *relname*
Selects each `ILINK` node in the relation *relname*.

For example,

```
foreachNode relation CROSSREF {
    withNode anchor SOURCE { puts "[content]: " }
    withNode anchor TARGET { puts "[query propval title]" }
}
```

10 Miscellaneous utilities

10.1 Environments

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

```
environment envname [ name value ...]
```

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

```
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'.

If the set and save subcommands are passed one extra argument, it is treated as a list of name-value bindings.

10.2 Substitutions

When translating SGML documents to other formats (including other SGML document types), it is often necessary to "escape" or "protect" character data that might be interpreted as markup in the result language. For example, HTML requires all occurrences of <, > and & to be entered as entity references <, > and &. TEX and LATEX have *many* special characters which must be entered as control sequences.

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

```
substitution substName {
    string replacement
    string replacement
    ...
}
```

Defines a new Tcl command *substName* which takes a single argument and returns a copy of the input with each occurrence of any *string* replaced with the corresponding *replacement*. If multiple *string*s match, the earliest and longest match takes precedence.

Example

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

11 Examples

11.1 Query examples

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

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 filters out the elements which have an implied `HREF` attribute. Without this clause, the returned list would 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 from an HTML document, printing the anchor name of each destination anchor and the target URL of each source anchor, along with the anchor text:

```
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 "\t<URL:[query attval HREF]>: [content]"
}
```

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

```
specification printAnchors {
    {element A hasatt HREF} {
        START        { puts stdout "<URL:[query attval HREF]>: " nonewline }
    }
    {element A hasatt 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 }
    }
}
process printAnchors
```

The next example demonstrates a multi-step navigational query. (Each query clause is listed on a separate line for clarity.)

```
proc xreftext {refid} {
    withNode \
            doctree \
            element SECT \
            withattval ID $refid \
            child \
            element TITLE  \
    {
        return [content]
    }
    error "No such section $refid"
}
```

'doctree element SECT' selects all the SECT elements. 'withattval ID $refid' tests if the source node has the right ID. 'child element TITLE' navigates to the first TITLE subelement, and then the 'withNode' body returns the content of that element. (This could be used to generate cross-reference text from an ID reference, for example.)

Another way to do this is:

```
join [query* doctree element SECT withattval ID $refid \
        child element TITLE subtree textnode content]
```

> The join command is necessary if the TITLE element contains subelements or SDATA nodes, in which case 'query* ... subtree textnode content' returns a list with more than one member.

11.2 A spell-checker

If you've ever tried to run the Unix utility 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. (It's programmed to understand LaTeX and nroff markup, but it doesn't know anything about SGML.)

This example, which demonstrates how to use [incr tcl] objects as ESIS event handlers, 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 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, modulo the list of suppressed elements. It recognizes one processing instruction: '<?spelling *word* *word*...>' adds the listed *word*s to a local dictionary.

Here is the 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'

needExtension ITCL

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;     # flag: currently suppressing output?
    protected wordlist "";       # local 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]
    }

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

    # Feed character data to 'ispell':
    method CDATA {} {
        if !$suppressing { puts $ispellpipe [content] }
    }
```

```
    method PI {} {
        # Is this a <?spelling ...> instruction?
        if {[lindex [query content] 0] == "spelling"} {
            # Yep; add to local dictionary:
            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 an index of every command and query clause mentioned in the document, cross-referenced to the section in which it appears.

The DTD uses a recursive model for sections:

```
<!element sect        - O (title,(%m.sect;)*,subsecs?) >
<!element subsecs        - - (sect+)>
```

Each `SECT` element contains a `TITLE` (the section heading), followed by any number of block-level elements (`%m.sect;`), and an optional `SUBSECS` element, which in turn contains other sections.

Commands are tagged with the `CMD` element, and query clauses are tagged with the `QC` element.

```
#
# 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 pass:
#
    process outline
    nl; nl;
#
# The second pass builds and prints an index of each command
# (CMD elements) and query clause (QC elements) used in the document,
# printing 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 number properties ('secnum') 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)
            setprop secnum [join [query* rootpath propval secctr] "." ]

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

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

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


#
# 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;
    }
}
```

12 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.