Cost 2 Reference Manual

Joe English

Version 2.2, 26 Jun 1999

Contents

1 Introduction

Cost is a structure-controlled SGML application programming tool. It is implemented as a Tcl extension, and works in conjunction with James Clark's nsgmls and/or sgmls 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 can be dynamically **load**ed into a Tcl application with the usual **package** mechanism, or it can be statically linked into a custom Tcl interpreter. There is also a command-line interface, **costsh**, which can be used interactively or as part of a command pipeline. A windowing interface, **costwish**, is also available for building GUI applications with Cost and Tk.

Cost provides a relatively low-level interface: it's a toolkit for building SGML applications rather than an SGML application itself. One application, `Simple.tcl`, is provided as an example and as a starting point.

2 Running Cost

Cost can be included in a Tcl shell in the usual way, by calling '`Cost_Init(interp);`' from **Tcl_AppInit** and linking the executable with the Cost library (`libcost2.2.a` under Unix). It can also be dynamically loaded into an existing Tcl shell with the Tcl command:

```
package require Cost
```

2.1 Environment Variables

Cost uses the following environment variables, all of which are optional:

`$COSTLIB`
: Points to the directory containing `costinit.tcl` and other library scripts.

`$COSTPATH`
: A colon-separated list of directories to be searched by the **cost:require** command; see 9.5.

TCL_LIBRARY

TCLLIBPATH
: Used by Tcl to find its script libraries; see the Tcl documentation for details.

SGML_DECLARATION
: If set, the **loaddoc** command passes its value on the command line when running a subprocess to parse SGML documents. It should point to a file containing the default SGML declaration for your site. (This is primarily useful if you're using **sgmls**, which uses the Reference Concrete Syntax by default. Most document types won't work with this SGML declaration, so you usually have to provide a different one. **nsgmls** uses a more sensible set of defaults.)

SGML_CATALOG_FILES
: Used by **nsgmls** and later versions of **sgmls** to find catalog files to resolve `PUBLIC` identifiers.

2.2 Loading documents

The Cost 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**. Returns the *document handle* of the new document.

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**.

loadxml parses and loads an XML document:

`loadxml` *`filehandle`*

Reads an XML document from *filehandle* and constructs the internal document tree, where *filehandle* is a Tcl file handle. Sets the current node is set to the root of the document, and returns a new *document handle*.

All of the above commands return a *document handle*, which may be passed to **selectDocument** or **withDocument** to change the current document:

`selectDocument` *`documentHandle`*

Sets the current document to the document referred to by *documentHandle*, and sets the current node to the root node of that document.

`withDocument` *`documentHandle`* `{` *`script`* `}`

Sets the current document to the document referred to by *documentHandle*, sets the current node to the root node of that document, then evaluates *script* as a Tcl script. When the script returns – either normally or with an error, **return**, **break**, or **continue** statement – restores the previous current document and current node. Returns: whatever *script* does (including Tcl result codes).

`currentDocument`

Returns the document handle of the currently active document.

2.3 Running costsh

Normally **costsh** is used in a pipeline with **[n]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 in sgmls format 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`

3 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 collection 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

`ILINK`
: A multi-ended link between nodes. (ILINK nodes do not come from the source document: they are created by application code. See 8)

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 SGML text 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 the **query** and **query*** commands; see 4 for full details.

3.1 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.

3.2 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.

`EL` 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`.

3.3 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.

3.4 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.

3.5 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.

Note: the `attribute` clause navigates to the attribute node; it does not return anything. To return an attribute value, use '`query attval` *`attribute-name`*' or '`query attribute` *`attribute-name`* `content`'.

`query* attlist`

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

Note: the `attlist` clause navigates to attribute nodes; it does not return anything. To return a list of attribute names, use '`query* attlist attname`'.

`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.

The `attlist` clause selects all attributes declared for an element, even those which have an `#IMPLIED` value. To select only those attributes for which a value is specified, you can use

```
foreach attname [query* attlist attname] {
        if {[query? hasatt $attname]} {
            ... process attribute here
        }
}
```

4 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.

4.1 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.

Some clauses are *predicates*, which test some property of the current node and either succeed or fail based on the outcome of the test. Other clauses are *navigational*; these traverse the document tree. Some navigational clauses select a single node (e.g., `parent`) and others generate a list of nodes (e.g., `ancestor`). Finally, *value* clauses are used to return information about the current node (e.g., `gi`, `attval`, `content`).

Queries are evaluated from left to right, evaluating each clause in turn. Each clause 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. It is not an error if the overall query fails; the **query** command just returns an empty string in this case.

4.2 Examples

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.

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
```

The next example demonstrates a multi-step navigational query. (Each query clause is listed on a separate line for clarity.) (This could be used to generate cross-reference text from an ID reference, for example.)

```
proc xreftext {refid} {
    return [join [query* \
        doctree \
        element SECT \
        withattval ID $refid \
        child \
        element TITLE \
        subtree \
        textnode \
        content]]
}
```

The '`doctree`' clause selects every node in the document. '`element SECT`' is a predicate which selects all the `SECT` elements. '`withattval ID $refid`' tests if the source node has the desired ID. '`child`' selects all the children of the selected `SECT` element. '`element TITLE`' selects the `TITLE` subelement of the selected `SECT`. '`subtree`' selects all the descendant nodes of the `TITLE` (i.e., `PEL` nodes and possibly subelements and all of their descendants). '`textnode`' selects all the data nodes, and finally '`content`' returns the data content.

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

4.3 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***.

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

Returns the number of nodes selected or results returned by the query *clause...*. **q#** and **countq** are synonyms for **query#**.

`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...*.

4.4 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 a synonym for `next`; it is present for symmetry with `esib`.

`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`'.

The `forward`, `backward`, `later`, and `earlier` clauses select all nodes in the subtrees before and after the current element. `backward` and `earlier` select the same set of nodes, but in a different order (`earlier` traverses nodes in the same order as `subtree`; `back-wards` tracerses them in the reverse order). `forward` and `later` are synonyms; both names are included for symmetry with `backward` and `earlier`.

`query* forward`

Selects all nodes in the tree which appear after the source node.

`query* backward`

Selects all nodes in the tree which appear before the source node. Nodes are selected in the reverse order, beginning with the source node's immediate predecessor and ending with the root node.

`query* later`

Synonym for `forward`.

`query* earlier`

Selects all nodes in the tree which appear before the source node. Nodes are selected in the same order as a preorder traversal, beginning with the root node and ending with the node's immediate predecessor.

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.

4.5 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 `address`. Selects each node in *addrlist*, in the order they appear in the list.

4.6 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`*'.

5 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.

Tcl-style comments – beginning with a `#` and extending to the end of the line – may appear in specifications, but only at the *beginning* of the name-value pairs, and at the very beginning of the specification. For example:

```
specification foo {
    #
    # This comment is legal
    #
    {element P} {
```

```
            # LEGAL
            name1         value1
            # ILLEGAL
            name2         value2
        }
        # ILLEGAL
        {element Q} {
            # LEGAL
            name1 value1
        }
}
```

6 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 *propname*s 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.

7 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 }
}
```

8 Links and relations

This facility is still experimental and subject to change.

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

A *relation* is a named collection of *ilink*s. An *ilink* is a collection of one or more named *anchors*. Each *anchor* is a reference to a node in the tree; the referenced node is the *endpoint* of the anchor. Ilinks also have an *origin* node; this is the node which was current when the ilink was created. All ilinks in the same relation typically have the same structure (number and names of anchors). It is possible to traverse to an ilink from any of its endpoints, and to any endpoint from an ilink. Cost ilinks are similar to HyTime ilinks and XML *XLINK*s; the chief difference is that in Cost an anchor may only reference a single node: aggregate anchors and character data spans are not (yet) supported.

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 to them.

> **relation** *relname* \
>   [*anchname1 anchname2* ... *anchnameN*]
>
> Creates a new relation named *relname*. The *anchname1* ... *anchnameN* parameters are currently ignored; they may be used to document the intended structure of the relation.

> **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 queries defining each 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 last clause returns the value of that node's `TITLE` property. In other words, the above example prints the `TITLE` of the node referenced by each `XREF` element in the document.

The `anchtrav` query clause navigates across ilinks; it combines the `ilink` and `anchor` clauses into one step. Thus the above example can be expressed more concisely as:

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

`query* anchtrav` *relname srcanch dstanch*

Selects the target node of the *dstanch* anchor in every ilink in the relation *relname* for which the current node is the *srcanch* anchor. In other words: traverses a link.

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]" }
}
```

9 Miscellaneous utilities

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

9.1 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.

9.2 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"
```

9.3 Numerals

```
require Numerals.tcl
arabic        number
lcroman        number
ucroman        number
lcalpha        number
ucalpha        number
```

The file $COSTLIB/Numerals.tcl provides a set of routines for converting integers to different styles. These are useful for generating numbered lists, section and appendix numbers, etc. **lcroman** and **ucroman** return lower-case and upper-case Roman numerals, respectively. **lcalpha** (**ucalpha**) converts the numbers 1 ... 26 to the letters ’a’ ... ’z’ (’A’ ... ’Z’). **arabic** returns its argument unchanged; it is provided for symmetry.

Example:

```
set counter 0
foreachNode child element LI {
    incr counter
    puts "[lcroman $counter]. " nonewline
    ... process content
}
```

9.4 Miscellaneous Miscellaneous

`cost:undefined` *`class value`*

The command **cost:undefined** is a debugging and maintenance aid. The first time it is called with a particular *class*-*value* pair, it prints a warning message on `stderr`. Subsequent calls with the same arguments are ignored.

It useful to have a default rule in each translation script that simply calls **cost:undefined**; for example:

```
specification mySpec {
    element FOO        { ... }
    element BAR { ... }
    ...
    el {
        startAction { cost:undefined GI [query gi] }
    }
}
```

This will print a warning if the specification fails to account for any element types encountered in the document – for example, if new element types are added to the DTD.

The purpose of the *class* argument is to keep track of different namespaces, e.g., `cost:undefined NOTATION [query dcn]`, etc.

9.5 Script management

`cost:require` *`filename`*

The **cost:require** command looks in the Cost search path (defined by the environment variable `$COSTPATH`) for *filename* and sources it as a Tcl script. It keeps track of which files have been loaded, and will only read a file once. Multiple calls to '`cost:require` *`filename`*' are therefore safe.

The shell command '`costsh -S` *`specfile`*' uses **cost:require** to find *specfile*.

require is an alias for **cost:require**, for backwards-compatibility with previous releases of Cost.