Transformers 5 ways

Copenhagen SGML Tool

SGML '93

Boston, 6-9 December 1993

Klaus Harbo

1 Foreword

This paper is a slightly modified version of a paper presented at the SGML '93 conference that was held in Boston in December 1993.

The paper presents the solutions to several SGML translation problems using the Copenhagen SGML Tool (CoST). The problems were submitted by the presenters in the Translation session at the conference. The idea of the session was that each of the presenters should solve the submitted problems using the translation technology of their organization, giving the conference participants a chance to compare the products more directly than is usually possible.

The products represented at the session were Balise (AIS), CoST (University of Copenhagen), OmniMark (Exoterica), SGML Hammer (Avalanche), and TagWrite (Zandar). This paper presents the solutions to the problems using CoST.

The version of this paper that was presented at the conference did not include the problem descriptions, since these were made available to all conference participants. For convenience, the problem descriptions for each problem has been added to obtain the current version. With the exception of the addition of this foreword and a few cosmetic modifications, nothing has been changed.

2 Background

Converting your documents to SGML promises you more document flexibility. But converting to SGML is not much use in itself. The potential of SGML is really that it lets you perform more flexible processing on the documents.

There is no doubt that the availability of public domain validating parsers (ARCSGML, SGMLS) is good for the SGML market. These packages let you acquire SGML experience without investing heavily in SGML software. Once you acquire more experience, you may want to upgrade to commercial software. The public domain software, however, allows you to get started. With a parser you can verify that the SGML documents you have created are indeed valid.

But without suitable tools, processing the documents is not exactly easy. Some people try to process the SGML files directly with tools of their own making. This is usually not a good approach since SGML documents are meant to be parsed by SGML parsers that know about DTDs, entities, etc. Another - better - approach is to make specialized tools that *parse the output of the validating parsers*, but this still leaves a lot of work to the programmer; the programs are usually hard to write and even harder to maintain.

CoST (Copenhagen SGML Tool) is a public domain, flexible SGML-aware programming environment which attempts to overcome these difficulties.

The intention of this paper is to show how you can use CoST to write programs that process SGML data. The programs shown here are all translations but there is no reason why CoST couldn't be used for other types of processing as well.

3 Programming CoST

It makes sense to describe CoST briefly, since this paper is the first publically available description of CoST.

3.1 Overview

CoST is object-oriented. During processing the *CoST kernel* creates the objects that actually handle the processing of the *parsing events* that are generated by the SGML parser. It follows from this fact that most processing in CoST is controlled by the actions of the parser. You program CoST by writing pieces of code that are executed by the CoST kernel under certain, well-defined circumstances.

When you write CoST processing specifications, you *control what objects are created to handle the parsing events*. The advantage of CoST is that you don't have to write all the functionality of the instantiated objects yourself. The objects that are created all share a common set of basic functionality (which depends on the parsing event that the object is supposed to handle). Among other things, the basic functionality lets you access document structure (ESIS) information.

The CoST kernel handles basic object management, i.e. the creation and deletion of objects. The CoST programmer can override the default object management, e.g. that some part of the parse tree (represented by the objects) should not be deleted.

A CoST processing specification consists of a set of declarations of *element object classes* that extend the basic object functionality to handle processing of specific element types (GIs) in the application. You can specify processing for individual or several GIs, TEXT (i.e. PCDATA, CDATA, RCDATA), SDATA, Record Ends, and processing instructions. Default processing is available too.

If several element object classes need to share functionality or access to certain variables, you can create *general object classes* (`itcl_class`) from which element object classes can `inherit` functionality. Classes can have `protected` variables (each object get its own copy of the variable) and `common` variables (every object of the class can access the same variable). A special variant of `protected` variables are `public` variables which can be accessed read-write from outside the object.

The most basic operation is to specify what processing should take place at the start and end of an element. This is done with `start` and `end` declarations in element object class declarations. If it is necessary to have *two-pass processing* (`Treemode`), you can specify what should happen in the first pass using the `pass1Start` and `pass1End` declaration. The `start` and `end` methods are special because they are *automatically invoked* by the CoST kernel. The same applies to `pass1Start` and `pass1End`.

You can specify invokable methods (`method`) on classes, as well as procedures (`proc`). Procedures are usable only within objects of the class. Special are the object `constructor` and `destructor` methods, which are automatically invoked at object creation and dele-

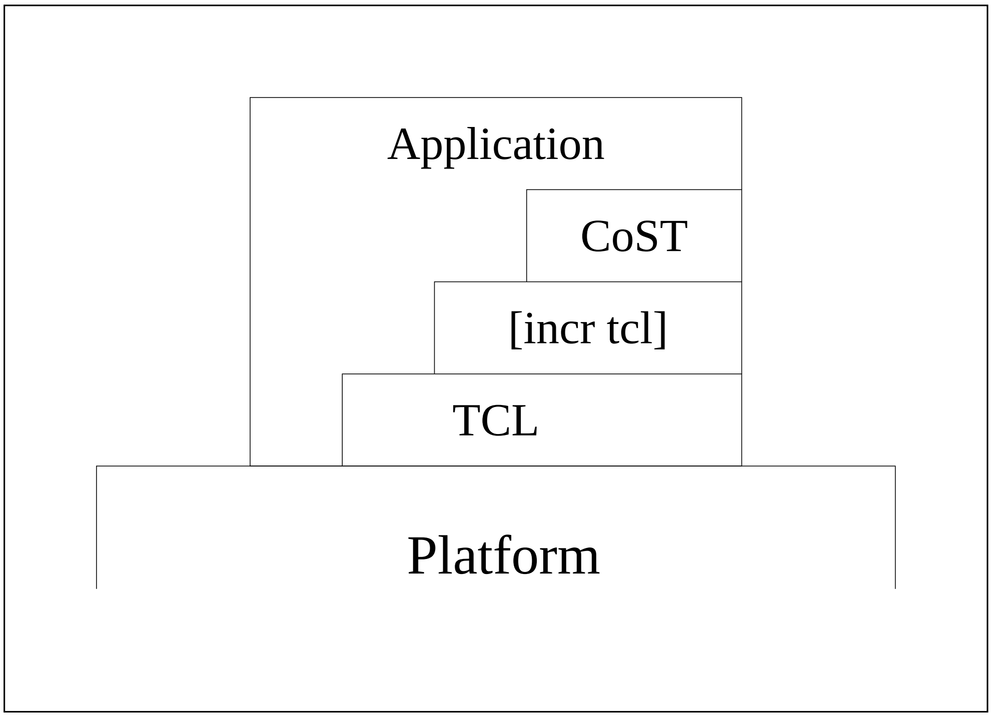

Figure 1: Relationship between CoST and its tools.

tion, respectively.

3.2 Syntax

Variables (scalar and arrays) can have any name. The value of a variable is accessed with a $ prefix. Grouping is like C, with curly braces ("{}"). Like C, all statements return a value which can be ignored. Where a *value* is valid, a statement can be inserted using square brackets ("[]"), which force evaluation of the statement. The command `puts` writes a string to a file, often `stdout` or `stderr`. The default file for `puts` is `stdout`. Comments are indicated by "#".

3.3 Tools used in CoST

CoST is an interpreter of the parsing events generated by the public SGML parser *SGMLS* created by James Clark. The basic programming language of CoST is *TCL* - Tool Command Language - created by John Osterhout, UC Berkeley. The object-oriented extension of TCL is *[incr TCL]* created by Michael McLennan, AT&T.

The feasibility of CoST as a one-person development project relies on the extensive use of these tools. Without them, developing CoST would have taken much more effort than was actually the case. See figure.

The overall design of CoST and the use of the above mentioned tools make CoST a highly flexible and extensible system. It is quite easy to write CoST programs that call routines written in other (compiled or interpreted) programming languages such as C, C++, Common Lisp, Scheme, etc. Is is also possible to re-write whole CoST programs in compiled languages for performance.

Programming libraries for TCL are available for processing that require access to UNIX system programming utilities, distribted programming, development of Graphical User Interfaces (GUIs), etc.

3.4 Current implementation

With the exception of the C code used to interface to SGMLS, the current version of CoST is implemented entirely in TCL, which is an interpreted language. A natural result of this is that speed isn't CoST's greatest strenth. However, TCL is designed to integrate smoothly with compiled code. It is the intention to port performance critical parts of CoST to C.

3.5 Availability

CoST currently runs under UNIX. It is known to run on Sun Sparcstation (SunOS 4), Digital DECstation (Ultrix) and Alpha (OSF/1), and the HP 700-series (HP-UX). The underlying packages (SGMLS, TCL) will run in most UNIX environments, so porting CoST to most UNIX platforms should be a minor effort. Porting CoST to DOS is not inconceivable, but isn't part of current plans.

3.6 More information

More information about CoST is available from:

- Klaus Harbo
  Dept. of Computer Science / Euromath Center
  University of Copenhagen
  Universitetsparken 5
  DK-2100 Copenhagen
  Denmark
  E-mail: Klaus.Harbo@euromath.dk
  Phone: +45 3532 0713

4 Problems submitted for the Translation session

4.1 General remarks

In the following sections CoST solutions to the problems submitted by (some of) the participants in the translation session are presented. Some of the algorithms shown do not solve all of the posed problem. Indeed, some of the submitted problems are not solved at all. The intention of the solutions presented is to give an impression of the CoST programming model.

Many of the problems are solved with *two-pass* solutions. Please note that the two passes are handled by CoST internally. The document is not parsed twice. This two-pass approach is termed *internal two-pass translation*. An internal two-pass translation requires that the part of the document which is processed in two passes must be held entirely in memory. For some document processing problems (notably processing of very large documents) a *external two-pass translation* may be a better approach. This approach means that the document instance is parsed and processed twice using two different programs. The processing in pass two presumably uses data (which is probably written to a file) collected during pass one.

None of the solutions presented in this document use external two-pass translation.

Record handling in SGML documents is known to be problematic for many users of SGML. Understanding the finer details of this area of SGML processing isn't easy. I have not spent much time thinking about this aspect in the solutions presented here, so there may be a few problems in this area here and there. Since the programs presented will never be used for any practical purpose it didn't seem worthwhile to spend time on this issue.

Regrettably there wasn't time to solve all of the problems submitted by the participants of the translation session. In particular, there wasn't time to come up with a solution for the ICADD problem submitted by Yuri Rubinsky. That it turned out be the ICADD problem is purely coincidental and doesn't indicate lack of support for the ICADD project.

For each of the presented problems, the problem description is presented as given to the presenters at the translation session.

4.2 Problems submitted by David Sklar

4.2.1 Extraction/sorting (EBT #1)

Problem description (by David Sklar, EBT)

Consider a DTD that allows FIGURE elements to be placed anywhere in the body of the document. Some figure elements have caption attributes, but not all of them do:

```
<!ELEMENT ROOT                    (FRONTMATTER,BODY,ENDMATTER)>
```

```
<!ELEMENT FIGURE         - O     (HOTSPOT*)>
<!ATTLIST FIGURE
          caption        CDATA   #IMPLIED
          filename       CDATA   #REQUIRED>
```

Create a transformation that appends an element called FIGREVIEW to the end of the ENDMATTER's children list:

```
<!ELEMENT FIGREVIEW      - -     (FIGURE+)>
```

FIGREVIEW simply contains copies of all the FIGURE elements found in the body of the document, sorted lexicographically based on the figure's caption. (Assume all text is ISO 8859-1.) Uncaptioned figures should not appear at all in the FIGREVIEW section.

Note that each FIGURE element is actually the root of a subtree of arbitrary size (I intentionally don't show the content model for HOTSPOT); make sure the entire subtree of each captioned FIGURE is duplicated in the FIGREVIEW section. (I am interested in knowing if that is possible, in your technology, without full knowledge of the content model of HOTSPOT.)

CoST solution

Strategy Two-pass internal translation.

General approach This problem requires that certain parts of the document instance (the `FIGURE` elements) are output more than once. This can be handled either by saving the output generated for these elements, or by saving the objects representing the elements, so output can be reproduced at the end. The latter approach was chosen. See figure.

In pass one, the `CAPTION` attribute values are inserted as keys in a global array and inserted into a list. The associated values in the array are the root objects of the parse (sub-) tree representing figures. In pass two the list of captions is sorted and the array is used for accessing the `FIGURE` objects.

Notes

- A `NAMELEN` of more than 8 was necessary to parse documents with the DTD given.
- The problem description seems to indicate that ISO 8859-1 implies a sort order. That is not the case. The default sort order of the platform was used.

```
element FIGURE {
  Treemode
  pass1Start {
    dontDelete
    if {![isImplicit CAPTION]} {
      global captions
      set captions([attrValue CAPTION]) $this
```

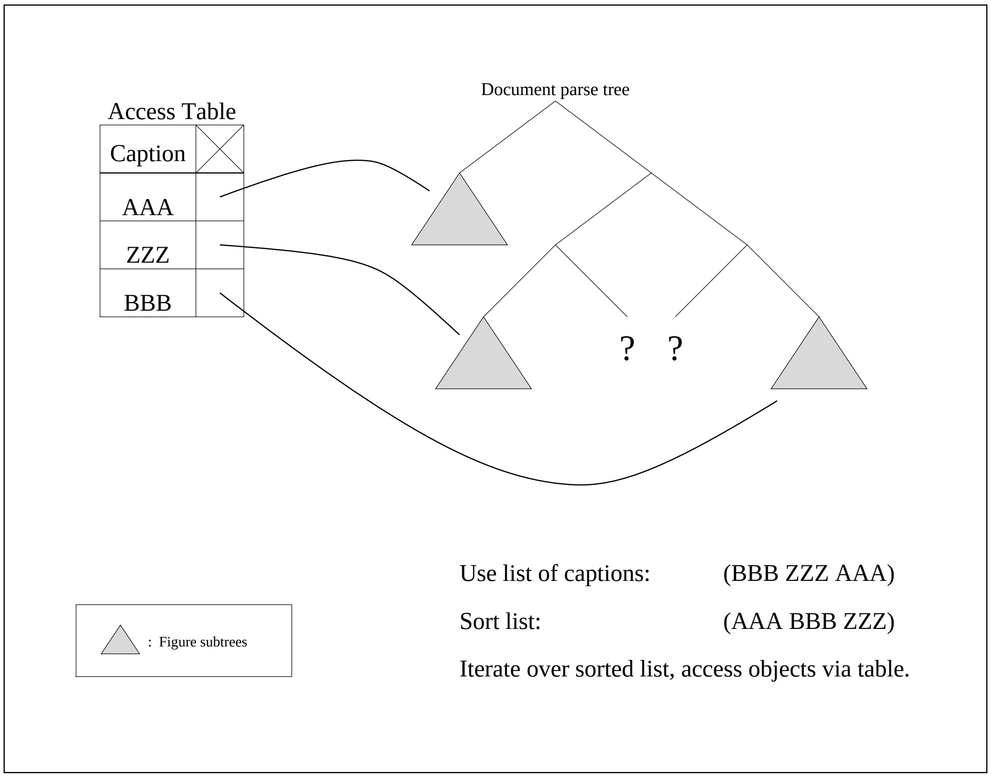

Figure 2: Saving certain parts of a document instance.

```
      }
    }
    start {
      puts stdout "<FIGURE" nonewline
      if {![isImplicit CAPTION]} {
        puts stdout " CAPTION=\"[attrValue CAPTION]\"" nonewline
      }
      puts stdout " FILENAME=\"[attrValue FILENAME]\">" nonewline
    }
    end { puts stdout "</FIGURE>" }
  }

  element ENDMATTER {
    start { puts "<ENDMATTER>" }
    end {
      global captions
      foreach i [lsort [array names captions]] {
        $captions($i) pass2
      }
      puts "</ENDMATTER>"
    }
  }

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

  SDATA { start { puts stdout "&data;"   nonewline } }
  TEXT  { start { puts stdout "$data"    nonewline } }
  PI    { start { puts stdout "<?$data>" nonewline } }
```

Notable program features

- It isn't necessary to assume anything about the content model of `FIGURE` or `HOTSPOT` elements to reproduce output.
- `FIGURE` elements are processed in `Treemode`, which means that they are processed in two passes, and that children are not automatically deleted after their end tag has been seen (or inferred) by the parser.
- `dontDelete` means that `FIGURE` elements are not deleted either. Of course all objects of the subtree must exist if output is to be reproduced.
- The global array variable `captions` is used for `FIGURE` object access in pass two.
- All elements except `FIGURE` and `ENDMATTER` are processed according to the `default` declaration. Only attributes that have explicit values are output.

4.2.2 Duplication of destination text at source of cross reference (EBT #2)

Problem description (by David Sklar, EBT)

Consider a DTD that contains SECTREF elements that represent cross-references to SECTION elements.

```
<!ELEMENT    SECTREF        - O     EMPTY>
<!ATTLIST    SECTREF
             REFID   IDREF   #REQUIRED>
<!ELEMENT    SECTION        - -     (TITLE,SUBSECT+) >
<!ATTLIST    SECTION
             ID      ID      #IMPLIED>
<!ELEMENT    TITLE          - -     (#PCDATA|EMPH)* >
```

The transformation should replace each input-DTD SECTREF with an enhanced output-DTD SECTREF:

```
<!ELEMENT    SECTREF        - -     (#PCDATA)>
<!ATTLIST    SECTREF
             REFID   IDREF   #REQUIRED>
```

When a SECTREF is encountered during the transformation, the engine should automatically generate this text as the content of the output SECTREF:

```
see 'XXXXXX'
```

where XXXXXX is a copy of the contents of the TITLE of the SECTION that is referenced by the SECTREF. Note that TITLE is not atomic; a TITLE instance can contain an arbitrary number of EMPH subelements. Also note that it is possible for a SECTREF to appear before the SECTION to which it refers.

The transformation should perform error checking to ensure that exactly one SECTION is referenced, report problems appropriately and in a 'non-fatal' way (i.e. should continue processing and produce a useable document instance).

CoST solution

Strategy Two-pass internal translation.

General approach `TITLE` elements have non-trivial content (in this case it is `PCDATA` or `EMPH` elements, but it could have been anything). Again, we can either save the output that is generated by `TITLE` and its content, or we can make sure that we can reproduce it on request. As in the previous exercise, the latter approach is chosen.

Two passes are used (cf. `Treemode` in the declaration of `DOC`). In the first pass, each `TITLE` element in `SECTION` elements register itself (with its *object ID* `$this`) at its parent `SECTION` element object (the reference is to the `SECTION`, not to the `TITLE`). The object ID of the `TITLE` element object is inserted into a global array with the `ID` of the SECTION as key. In the second pass, when the output of a TITLE element is needed, the array is used to request the TITLE object and its children to reproduce their output.

Notes

- The whole document is processed in Treemode.

```
itcl_class SHARED { common titleobjs  }

element DOC {
  Treemode
  start {}
}

element TITLE {
  pass1Start {
    if {[in SECTION]} { [parentID] registerTitle $this }
  }
  start { puts "<TITLE>"  }
  end   { puts "</TITLE>" }
}

element SECTION {
  inherit SHARED
  method registerTitle obj {
    if {![isImplicit ID]} {
      set titleobjs([attrValue ID]) $obj
    }
  }
  start {
    puts stdout "<SECTION" nonewline
    if {![isImplicit ID]} {
      puts stdout " ID=[attrValue ID]>" nonewline
    }
  }
  end { puts "</SECTION>" }
}

element SECTREF {
  inherit SHARED
  start {
    puts stdout "<SECTREF>see '" nonewline
    foreach obj  [$titleobjs([attrValue REFID]) listChildren] {
      $obj pass2
    }
    puts stdout "'" nonewline
  }
  end { puts stdout "</SECTREF>" nonewline }
}

element EMPH {
  start { puts stdout "<EMPH>" nonewline }
  end   { puts stdout "</EMPH>" nonewline }
}
```

Notable program features

- The `titleobjs` array is a `common` variable in the `SHARED` object class, which both `SECTION` and `SECTREF` objects `inherit`. CoST supports multiple inheritance as well.
- In the declaration of `TITLE`, the method `parentID` is used to invoke a method on the object representing the parent in the parse tree. In this case the method `registerTitle` is invoked; this method is defined in the declaration of `SECTION`. As you can see, this method is invoked by `TITLE` elements only if their parent is a `SECTION` element object (i.e. `[in SECTION]` evaluates to true) so there is no risk that this method is invoked on objects for which it isn't defined.
- The method `pass2` (which is part of CoST objects' basic functionality and therefore not shown here) is invoked on `TITLE` objects to generate output. Under normal circumstances, this method is called by the CoST kernel (in fact it is called by the kernel for generating the output for `TITLE` elements where they occur), but - as shown - it can also be called explicitly by the CoST programmer.
- The method `listChildren` can be used to access the children of an object. It can be used only when CoST is in `Treemode`, because the data needed for it is generated just before pass one in `Treemode`.

4.3 Problems submitted by John McFadden

General remarks made for all the problems:

> "Exoterica has selected the following problems as things that are encountered in "real world" applications, while at the same time being simply stated. Each problem is described both by a general statement and a specific example, to allow the general aspects of a problem to be considered and to provide a case that can be used to illustrate specific solutions to the problem."

4.3.1 Row counting in tables (EXO #1)

Problem description (by John McFadden, Exoterica)

While converting an SGML marked-up table to some other form (e.g. a typesetting language), count the number of rows and colums in the table and put the total in a specified position at the start of the table.

Given the input:

```
<!DOCTYPE    doc [
<!ELEMENT    doc     0 0     (table+)>
<!ELEMENT    table   - -     (row+)>
<!ELEMENT    row     - 0     (col+)>
<!ELEMENT    col     - 0     (#PCDATA)>
]>
<table>
```

```
<row><col>xxx<col>xxx<col>xxx
<row><col>xxx<col>xxx<col>xxx
<row><col>xxx<col>xxx<col>xxx
<row><col>xxx<col>xxx<col>xxx
<row><col>xxx<col>xxx<col>xxx
</table>
<table>
<row><col>xxx<col>xxx<col>xxx
<row><col>xxx
<row><col>xxx<col>xxx
</table>
```

Produce the output:

```
TABLE 1 (5 rows, 3 columns)
{xxx|xxx|xxx}
{xxx|xxx|xxx}
{xxx|xxx|xxx}
{xxx|xxx|xxx}
{xxx|xxx|xxx}
TABLE 2 (3 rows, 1-3 columns)
{xxx|xxx|xxx}
{xxx}
{xxx|xxx}
```

CoST solution

Strategy Local tree-mode, internal two-pass translation.

General approach Before the first line of output for a `TABLE` is generated, the whole `TABLE` must be processed (we need to know the total number of row in the table, etc.), i.e. a two-pass solution is required. But since the information we need is wholly contained in a single `TABLE`, the `Treemode` is used only for processing these elements; it is not necessary to process all of the document before generating any output. Otherwise the processing required is fairly simple. Column separators ("`|`") should not be generated for the first column in each row.

Notes

- Treemode is local to `TABLE`.

```
itcl_class SHARED {
  common tabno 1
  common colmin
  common colmax
  common colno
  common rowno
}

element TABLE {
  Treemode
  inherit SHARED
```

```
  pass1Start {
    foreach i { colmax colno rowno } { set $i 0 }
    set colmin ""
  }
  start {
    puts stdout "TABLE $tabno ($rowno rows, " nonewline
    if {$colmin!=$colmax} {
      puts stdout "$colmin-" nonewline
    }
    puts stdout "$colmax columns)"
    incr tabno
  }
  end { deleteChildren }
}

element ROW {
  inherit SHARED
  pass1End {
    if {$colmin=="" || $colno<$colmin} { set colmin $colno }
    if {$colno>$colmax} { set colmax $colno }
    incr rowno; set colno 0
  }
  start { puts stdout "\{" nonewline }
  end   { puts stdout "\}" }
}

element COL {
  inherit SHARED
  pass1End { incr colno }
  start { if {[rank]>1} { puts stdout "|" nonewline } }
}
```

Notable program features

- All the counters etc. that are used are `common` variables.
- After processing a `TABLE`, all children of the `TABLE` object are deleted. The `TABLE` element object itself is deleted automatically by the CoST kernel (no `dontDelete` is issued).

4.3.2 List Mark Formatting (EXO #2)

Problem description (by John McFadden, Exoterica)

Given a <LIST> element, that can have other <LIST> elements within its content, which has an optional attribute whose presence determines the manner in which items in each list are marked or numbered, and whose absence indicates that the mark or numbering form is to be deduced from the ancestry of the <LIST> (e.g. a <LIST> within a decimal-numbered list is to use lower-case letters), correctly mark or number each list item.

Given the input

```
<!DOCTYPE doc [
```

```
<!ELEMENT doc       O O     (list+)>
<!ELEMENT list      - -     (item+)>
<!ATTLIST list
     type    CDATA   #IMPLIED>
<!ELEMENT item      - O     (text, list?)>
<!ELEMENT text      O O     (#PCDATA)>
]>
<list>
<item>xxx
<list>
<item>xxx
<item>xxx
</list>
<item>xxx
<list type="*">
<item>xxx
<item>xxx
</list>
</list>
<item>xxx
<list type="R">
<item>xxx
<item>xxx
</list>
</list>
```

Produce the output:

```
1. xxx
   a) xxx
   b) xxx
2. xxx
   * xxx
   * xxx
3. xxx
   I) xxx
   II) xxx
```

CoST solution

Strategy One-pass translation.

General approach This problem is solved by defining af set of accepted *counter patterns* ("A" for arabic numerals, "L" for lower-case letters, "U" for upper-case letters and "R" for roman numerals; the list is arbitrary). The counter patterns are used in *list mark patterns*, e.g. you can specify " R)" to get roman numerals with two preceeding spaces and a close parenthesis as the marker for an item in a list. The item's rank in the list is then substituted for the counter pattern during processing.

Additionally an *inheritance scheme* is necessary. Exactly which which scheme is selected isn't too important, a circular inheritance is used. See figure. When an explicit value for the TYPE attribute overrides the default (inherited) marker pattern, the inheritance scheme is reset, starting with an arabic number marker. This is done by setting the count pattern to the empty string. It is indicated by an empty circle in the figure.

Notes

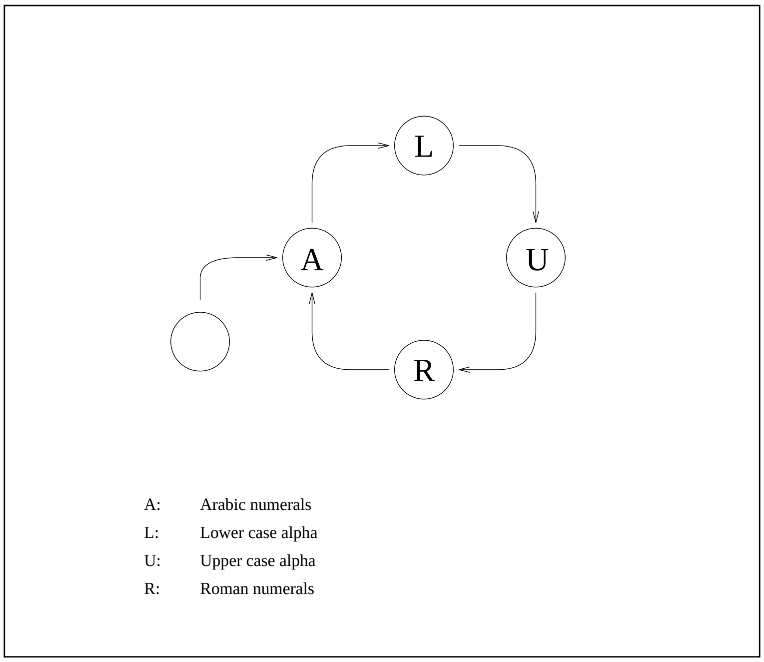

Figure 3: Inheritance for list markers.

- A stack of the list markers is maintained explicitly. This seemed to be a more convenient approach than accessing the attribute values by navigating the parse tree (the element object tree).
- The characters that match a counter pattern in the `TYPE` attribute values (in this case "A", "L", "U", or "R") are assumed to be patterns.
- Default mark patterns are set in the `format` array for each of the available counter patterns.

```
itcl_class SHARED {
  public  type;                public  fstr
  common  markstack { "" };    common  indent 0
  common  heirform;            common  format
  common  patlist;

  constructor {} {
    set patlist {A L R U}
    set heirform() A;          set heirform(A) L
    set heirform(L) U;         set heirform(U) R
    set heirform(R) A
    set format(A) "\%s. ";     set format(R) "\%s) "
    set format(U) "\%s) ";     set format(L) "\%s) "
  }
}

element DOC { end { puts "" } }

element LIST {
```

```
  inherit SHARED
  start {
    if {[isImplicit TYPE]} {
      set type $heirform([lindex $markstack 0])
      set fstr $format($type)
    } else {
      set type ""
      set fstr [attrValue TYPE]
    }
    if {$type!=""} {
      regsub $type $fstr "%s" fstr
    } else {
      foreach pat $patlist {
        if {[regsub $pat $fstr "%s" fstr]} break
      }
    }
    set markstack [linsert $markstack 0 $type]
    incr indent
  }
  end {
    set markstack [lreplace $markstack 0 0]
    incr indent -1
  }
}

element ITEM {
  inherit SHARED
  start {
    puts ""
    for {set i 1} {$i<=$indent} {incr i} {
      puts stdout "   " nonewline
    }
    set rankstr [rankToString [rank] \
                 [[parentID] info public type -value]]
    puts stdout "[format [[parentID] info public \
                 fstr -value] $rankstr]" nonewline
  }
}

proc rankToString {rank listtype} { return $rank }
```

Notable program features

- CoST supports matching by regular expressions. Here regular substitution (`regsub`) is used for substituting counter patterns with place holders for formatting ("`%s`" for strings, as in C's `printf`).
- As mentioned, a stack is used to maintain information about parent mark patterns. In this example, the stack is implemented as a simple list, but CoST has built-in general classes such as `Stack`, `Set`, and `Counter`.
- The implementation shown does not support the formatting of counters as roman numerals. This is not a built-in feature of CoST, and it did not seem worthwhile to implement it in general for this exercise. As a result, the procedure which was supposed to perform this formatting (`rankToString`) simply returns the value as an arabic numeral in all cases.

4.3.3 Date (EXO #3)

Problem description (by John McFadden, Exoterica)

Output the current date in a form determined by an attribute value in the SGML document.

```
<!DOCTYPE doc [
<!ELEMENT doc       O O    (date+)>
<!ELEMENT date      - O    EMPTY>
<!ATTLIST date
     form    CDATA   #REQUIRED>
]>
<date form="M*,/ /,D,/, /,Y*">
<date form="Y,2M,2D">
<date form="D,3MU,2Y">
```

Produce the output (and given that the date is 7 Dec. 1993):

```
December 7, 193
19931207
7DEC93
```

CoST solution

Strategy Not much.

General approach The real issue in this exercise is how to parse the value of the FORM attribute. It isn't possible simply to split the string on comma characters, since commas can appear in *formatting literals* (strings between solidus ("/") characters).

The approach used is to parse the formatting string a token at a time. This is done by checking if the token is a *literal* (marked by "/" at the beginning and end) or a date *symbol*. This way a *format string* is gradually built which is then used to generate output at the end.

Notes

- There was a small error in the problem description, since in the DTD the DATE element has a FORM attribute, whereas in the example instance the attribute is called TYPE. For this exercise the correct attribute name was arbitrarily assumed to be FORM.

It seemed reasonable to divide the code needed for this exercise into two, a piece of code which parses the FORM attribute value and generates a date formatting string, and the CoST program which uses the formatting string to generate output. Conceivably, the code which parses date formats could be part of a larger programming library.

The library code looks as follows:

```
set fmtpat(D)   "%e"
set fmtpat(2D)  "%d"
set fmtpat(M*)  "%B"
set fmtpat(2M)  "%m"
set fmtpat(3MU) "%h"
set fmtpat(Y)   "%Y"
set fmtpat(2Y)  "%y"

proc buildDateFormatString str {
  global fmtpat
  set pat ""
  while {$str!=""} {
    if {[regexp {^/} $str]} {
      regexp {^/([^/]+)/} $str m match
    } else {
      regexp {^[^,]+} $str m
      set match $fmtpat($m)
    }
    set str [string range $str [string length $m] end]
    if {[string index $str 0]==","} {
      regsub "," $str "" str
    }
    set pat $pat$match
  }
  return $pat
}
```

Notable program features of the parsing code

- Current time and date is obtained with the `getclock` command.
- The date and time formatting supported by CoST corresponds to the capabilities of the `strftime()` C library function of the platform. The "%"-patterns above correspond to those of `strftime()`.

Given the above CoST library code functionality, the formatting of dates in the CoST program itself is fairly trivial:

```
load format-date.cost

element DATE {
  start {
    set format [buildDateFormatString [attrValue FORM]]
    puts [fmtclock [getclock] $format]
  }
}
```

Notable program features

- Access to date formatting is built-in functionality of CoST using the `fmtclock` command. In this exercise, the problem was to generate an appropriate formatting string from the FORM attribute value.

4.3.4 Line Breaking (EXO #4)

Problem description (by John McFadden, Exoterica)

In the process of converting an SGML document to some other form (or even SGML), produce output that has a given number of text characters on each line, not counting any commands or tags in that other form in the total, and adjusting the length of each line so that breaks only occur at "allowed" points (such as at spaces and existing line breaks).

Given the output

```
<!DOCTYPE doc [
<!ELEMENT doc       O O     (p+)>
<!ELEMENT p         - O     (#PCDATA | i )*>
<!ELEMENT i         - O     (#PCDATA)>
]>
<p>The fast <i>red</i> and <i>blue</i> goat
<p>Waded in the mirky moat.
<p>When it got out and sat down
<p>Its coat was all a foxy brown
```

Produce the output (given a line width of 12):

```
<DOC>
<P>The fase <I>red</I>
and <I>blue</I>
goat
<P>waded in the
mirky moat.
<P>When it got
out and sat
down
<P>Its coat was
all a foxy
brown
</DOC>
```

Require that there be no trailing spaces at the points of the breaks (i.e. none of the preceding lines have trailing spaces).

CoST solution

Strategy Not much.

General approach Probably some of the products presented at the Translation session have line length management built in. This is not the case with CoST. It is the intention to provide only the *core functionality* needed to process SGML documents. The list of facilities you might otherwise provide is endless. Instead programming libraries are encouraged. The implementation shown here could serve as a starting point of a generally available library function which provides management of line lengths.

The approach taken is to use an object to maintain information about the number of significant characters (characters in tags don't count) output on each line, and produce newlines when needed. The "price" for this functionality is that you can't just use `puts` to produce output. Instead methods are invoked on the line length manager (`lmgr`) to output strings, generate newlines, etc.

Understanding the code which implements the line length manager is left to the reader as an exercise.

Notes

- The CoST programmer could quite easily modify the `puts` command to use the line length manager for output. That would probably be a sensible thing to do if the length management code was included in a library, so programs wouldn't have to be modified to use line length management.

  In general, CoST supports ad-hoc modification of any built-in command.

```
itcl_class lengthMgr {
  protected used 0
  protected linelength
  protected last DATA
  protected saveTag ""
  constructor len {
    set linelength $len
  }
  method put data {
    set tmp [string trimleft  $data " "]
    set tmp [string trimright $tmp  " "]
    foreach tk [split $tmp " "] {
      set tklen [string length $tk]
      if {[expr $used+$tklen+1]>$linelength} {
          puts ""
        set used 0
      } else {
        if {$used>0} { puts stdout " " nonewline }
      }
      if {$last=="TAG"} {
        puts stdout $saveTag nonewline
        set last "DATA"
        set saveTag ""
      }
      puts stdout $tk nonewline
      if {[expr $used+$tklen+1]>$linelength} {
        if {$tklen>=$linelength} {
          puts ""
          set used 0
        } else { set used [expr $used+$tklen] }
      } else {
        set used [expr $used+$tklen]
      }
    }
  }
  method putStartTag tagstr {
```

```
      set saveTag "$saveTag$tagstr"
      set last TAG
    }
    method putEndTag tagstr {
      puts stdout "$tagstr" nonewline
    }
    method putNL {} {
      set used 0
      puts ""
    }
}
```

Notable program features

- A problem in coding line length management is to make sure that a start tag is not left at the end of a line, since this could produce an unwanted newline in the content of the element. This, in turn, may produce a parse error or unwanted effects in the application.

  This problem is handled correctly by the implementation shown.

Using the above (imaginary) library function, the CoST program itself is trivial:

```
load length-mgr.cost

lengthMgr lmgr 12

element DOC {
  start { puts stdout "<DOC>" nonewline }
  end   { puts stdout "\n</DOC>" }
}

element P {
  start {
    lmgr putNL
    lmgr putStartTag "<P>"
  }
}

element I {
  start { lmgr putStartTag "<I>"  }
  end   { lmgr putEndTag   "</I>" }
}

TEXT { start { lmgr put $data } }
```

Notable program features

- The line length manager (`lmgr`) is instantiated with the desired maximum line length.

- The above code assumes that line length management is required only in `P` elements, so the tags for the `DOC` elements are generated directly with `puts`.

4.3.5 Resolve External Cross References (EXO #5)

Problem description (by John McFadden, Exoterica)

Given two or more SGML documents each of which contain references to objects and locations in themselves and other documents in the set, replace each reference by an identification fo the reference object or location, whether it be in the document doing the referencing or some other.

Given two inputs, the first of which is:

```
<!DOCTYPE doc [
<!ELEMENT doc        O O     (title, p+)>
<!ELEMENT title      O O     (#PCDATA)*>
<!ELEMENT p          - O     (#PCDATA | ref)*>
<!ATTLIST p
     id      CDATA   #IMPLIED>
<!ELEMENT ref        - O     EMPTY>
<!ATTLIST ref
     id      CDATA   #REQUIRED>
]>
First Document
<p>xxxx <ref id="a"> xxx
<p id="a">xxxx
<p><ref id="d"> xxxxx
```

and the second of which uses the same DTD, with the instance:

```
Second Document
<p id="d">xxxx <ref id="a"> xxxx
```

Produce the output for the first input:

```
First Document
1. xxxx (2) xxx
2. xxxx
3. (1 in "Second Document") xxxxx
```

and for the second input:

```
Second Document
1. xxxx (2 in "First Document) xxxx
```

CoST solution

Safe resolution of external references inherently means parsing several document instances. In other words, the problem at hand is basically one of *coordination*. The system (parser + processing environment) is run several times, and the result of each run is collected. Finally output is generated.

So far, the development of CoST has been focused on developing a flexible and extensible programming model, and on refining the syntax used to exploit this programming model. A model for coordinating the processing of several instances has not been developed. This may happen later.

As a result, the CoST programmer who needs to process more documents to produce a desired result can coordinate processing by any means available to him, e.g. using shell or Perl scripts. (Perl is a programming language, created by Larry Wall, aimed primarily at system programming.)

No solution for this problem is shown here. It does not make sense to present an arbitrary set of tools that might be used to solve the problem of coordinating the CoST processing necessary to resolve external cross references in SGML documents.

4.4 Problems submitted by Francois Chahuneau & Christophe Lecluse

4.4.1 Instance normalization (AIS #1)

Problem description (by Christophe Lecluse, AIS)

Starting from an arbitrary source SGML instance with potential markup minimization, generate a non-minimized result instance :

- in which all SGML information according to ESIS has been made explicit
- which can still be parsed using the same DTD.

The program should be written to be DTD independent.

This kind of processing is extremely frequent. The result instance can be esaily post-processed by non-SGML systems (e.g. typesetting systems loaders) or by minimal SGML systems.

Precise statement:

- the result instance uses the reference concrete syntax
- all start tags should be present
- all end tags should be present (except for EMPTY declared elements)
- all declared attribute values should be present in start tags (whether specified in the source instance or inferred from the DTD default values), except for unspecified #IMPLIED attributes
- all entity references should be preserved for SDATA or PI internal entities, and for CDATA, SDATA or NDATA external entities

Variant:

To improve legibility and limit line length in the output, a non-significant new-line character is generated before each tag close delimiter on end tags.

For instance, `</PARA>` is coded as:

```
</PARA
>
```

Notes:

- for the result instance to be parsable, it is necessary that EMPTY declared elements have no end tag
- if "<" and "&" characters appear as data content in the source instance, it is assumed they have been represented as < and & SDATA entity references.
- if double quotes (") appear in CDATA attribute values, it is assumed they have been represented as " SDATA entity references.

CoST solution

The requirements for this exercise present two difficulties for CoST:

1. SDATA entity references should not be expanded in the normalized instance.
2. Normalization of elements that are declared EMPTY should be without end tag.

Both requirements are perfectly reasonable. In practice, normalization is often necessary when it is desirable to process (parts of) SGML documents with software that isn't fully SGML-aware (i.e. isn't using a real SGML parser to parse the entities of the instance).

The problem is that information about the *entity reference string* and whether or not an element was declared EMPTY is not part of the *Element Structure Information Set* (ESIS). Since CoST is a structure-controlled SGML application (all of which are driven exclusively by ESIS information), these two normalizations are not possible with CoST in general.

The problem with SDATA entities can be handled by assuming that the expansion of the entity reference is the entity reference itself. This is not an unreasonable thing to do, since SDATA entities - by definition - are specific to the local processing environment. The expansion can thus be modified locally to meet the assumption. This approach was chosen for the implementation shown.

The normalization of the EMPTY elements could be handled by handing a list of the empty elements to CoST at runtime. This approach was chosen for the implementation shown.

These approaches are both less than ideal. However, a more general solution requires tighter integration between the parser and the application than is currently the case with SGMLS and CoST. It is probably not desirable to have such tighter integration between SGMLS and CoST, because that would lead to inter-dependencies between the two systems. With the current looser integration CoST can take advantage of new versions of SGMLS without any modification.

Strategy Direct, one-pass translation.

General approach All elements are processed the same way (as specified in the `default` declaration). The start tag is generated with all attributes that have explicit values. The end tag is generated if the element isn't a member of the set of elements that are declared empty (it is this list that must be handed explicitly to CoST, since no information is available to CoST from which it can be derived).

Notes

- In the implementation shown, the list of elements that are declared EMPTY is simply defined in a statement. In another context, it may be more reasonable to read it from a file or provide the information on the command line.
- The newline characters are generated before the `TAGC` character, as required.

```
Set declEmpty {FIGREF FNREF SECREF}

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

SDATA { start { puts stdout "&$data;"  nonewline } }
TEXT  { start { puts stdout "$data"    nonewline } }
PI    { start { puts stdout "<?$data>" nonewline } }
```

Notable program features

- None.

4.4.2 Dictionary instance split/merge (AIS #2)

Problem description (by Christophe Lecluse, AIS)

Given a dictionary instance made of entries, split this instance into as many files as there are entries, while generating an SGML skeleton which keeps track of the entry file names.

In a second step, perform the inverse operation: use this skeleton to gather all entries (stored one per file) and re-create the original instance.

This exercise is a simplified version of a very common operation in database loading and extraction situations. It stresses the input/output capabilities of the application language.

Precise statement:

Entry for word "word" should be placed in file "word.sgm", where only the 8 first characters of the word are used to create the filename.

Option 1 assume that filename collisions never occur

Option 2 provide for collisions. In case of filename collision with the previous entry, the last character is substituted by 0...9 (it is assumed that collisions never occur more than 10 contiguous times)

The generated filename is stored in an attribute in the skeleton instance (see skeleton DTD).

Source DTD:

The dictionary DTD is the following :

```
<!DOCTYPE DICT [
<!ELEMENT  DICT      - O     (ENTRY+)              -- Dictionary -->
<!ELEMENT ENTRY      - O     (WORD,PHON,GRAM,DEF*)  -- Entry -->
<!ELEMENT WORD       - O     (#PCDATA)              -- Word -->
<!ELEMENT PHON       - O     (#PCDATA)              -- Phonetics -->
<!ELEMENT GRAM       - O     (#PCDATA)              -- Grammar -->
<!ELEMENT DEF        - O     (#PCDATA|EX)+          -- Definition -->
<!ELEMENT ETY        - O     (#PCDATA)              -- Etymology -->
<!ELEMENT EX         - O     (#PCDATA)              -- Example -->
]>
```

A very short instance is the following :

```
<DICT>
<ENTRY><WORD>absinthe </WORD>
<PHON>'ab-.sinth</PHON><GRAM>n </GRAM>
<DEF>: a liqueur flavored esp. with wormwood and  anise</DEF></ENTRY>

<ENTRY><WORD>absolute </WORD>
```

```
<PHON>'ab-s*-.lu_:t</PHON><GRAM>adj</GRAM>
<DEF>1: free from imperfection </DEF>
<DEF>2: free from mixture</DEF>
<DEF>3: free from control, restriction, or qualification</DEF>
<DEF>4: lacking grammatical connection with any other word in a sentence
<EX>absolute construction</EX></DEF>
<DEF>5: POSITIVE <EX>absolute proof</EX></DEF>
<DEF>6: FUNDAMENTAL, ULTIMATE</DEF></ENTRY>

<ENTRY><WORD>absolution </WORD>
<PHON>.ab-s*-'lu_:-sh*n</PHON><GRAM>n </GRAM>
<DEF>: the act of absolving; esp : a remission of sins pronounced
by a priest in the sacrament of penance</DEF></ENTRY>

<ENTRY><WORD>absolutism </WORD>
<PHON>'ab-s*-.lu_:t-.iz-*m</PHON><GRAM>n </GRAM>
<DEF>1: the theory that a sovereign should have unlimited power</DEF>
<DEF>2: government by an absolute ruler</DEF></ENTRY>
</DICT>
```

Extracted entries

The DTD for each isolated entry file is the following:

```
<!DOCTYPE ENTRY [
<!ELEMENT ENTRY      - O     (WORD,PHON,GRAM,DEF*)  -- entry -->
<!ELEMENT WORD       - O     (#PCDATA)              -- Word -->
<!ELEMENT PHON       - O     (#PCDATA)              -- Phonetics -->
<!ELEMENT GRAM       - O     (#PCDATA)              -- Grammar -->
<!ELEMENT DEF        - O     (#PCDATA|EX)+          -- Definition -->
<!ELEMENT ETY        - O     (#PCDATA)              -- Etymology -->
<!ELEMENT EX         - O     (#PCDATA)              -- Example -->
]>
```

A sample instance is the following

```
<ENTRY><WORD>absinthe </WORD>
<PHON>'ab-.sinth</PHON><GRAM>n </GRAM>
<DEF>: a liqueur flavored esp. with wormwood and anise</DEF></ENTRY>
```

Extracted skeleton

The skeleton DTD is the following:

```
<!DOCTYPE SKELETON [
<!ELEMENT SKELETON   - O     (ENTRY+)>
<!ELEMENT ENTRY      - O     (WORD)>
<!ATTLIST ENTRY
     FILE    CDATA   #REQUIRED>
<!ELEMENT WORD       - O     (#PCDATA)>
]>
```

where the "FILE" attribute encodes the file name in which the extracted entry was placed. The sample instance corresponding to the given source instance is the following:

```
<SKELETON>
<ENTRY FILE="absinthe.sgm"><WORD>absinthe </></>
<ENTRY FILE="absolute.sgm"><WORD>absolute </></>
<ENTRY FILE="absoluti.sgm"><WORD>absolution </></>
<ENTRY FILE="absolut0.sgm"><WORD>absolutism </></>
</SKELETON>
```

CoST solution

Strategy Tree-mode, internal two-pass translation.

General approach Pass one is used to collect the WORDs that are to be processed, pass two is used to actually split the document instance. The "skeleton" document is concurrently written to stdout.

Notes

- It is convenient to generate the start tag of ENTRY when processing the WORD element. This is safe because a WORD element is required as the first element in ENTRY elements.

```
itcl_class SHARED {
  common out    ""
  common files
}

element DICT {
  Treemode
  start { puts "<SKELETON>" }
  end   { puts "</SKELETON>" }
}

element ENTRY {
  inherit SHARED
  end { puts $out "</ENTRY>" }
}

element WORD {
  inherit SHARED
  protected { text  "" }
  protected { fname "" }
  proc uniquify word {
    regsub " " $word "" word
    set files(word) 1
    return $word
  }
  method registerText t { set text $t }
  start {
    set fname [uniquify "$text"]
    set out [open $fname "w"]
    puts stdout "<ENTRY FILE=\"$fname\"><WORD>$text</></>"
    puts $out "<ENTRY><WORD>" nonewline
  }
  end { puts $out "</WORD>" }
}
```

```
element {PHON GRAM DEF EX} {
  inherit SHARED
  start { puts $out "<[GI]>" nonewline }
  end   { puts $out "</[GI]>" }
}

TEXT {
  inherit SHARED
  pass1Start {
    if {[in WORD]} {
      [parentID] registerText $data
    }
  }
  start { puts $out $data nonewline }
}
```

Notable program features

- The handling of PHON, GRAM, DEF, and EX elements are the same; this fact is used syntactically so the declaration is written just once.

The code required to merge the files according to a SKELETON document is fairly trivial:

```
element SKELETON {
  start { puts "<DICT>" }
  end   { puts "</DICT>" }
}

element ENTRY {
  start {
    set in [open [attrValue FILE] "r"]
    while {![eof $in]} {
      puts [gets $in]
    }
  }
}

TEXT { start {} }
```

Notable program features

- The implementation shown assumes that the files referred to in the FILE attributes exist and are readable. Production code would require more checks.

4.4.3 Structural transformation (AIS #3)

Problem description (by Christophe Lecluse, AIS)

Starting from a source instance described by a hierarchical type DTD, generate a "flat" instance described by a "flat type" DTD. (This kind of problem is very typical of problems encountered when generating input for a non-SGML word processor or DTP tool; "flattening" to some kind of "transit" DTD is often needed).

In a second step, do the opposite: from the generated "flat" instance, re-generate the original hierarchical instance. (This kind of problem is very common in retroconversion).

Note: The two sample DTDs are designed to illustrate recursive definitions, and are not meant to be really useful in the real world.

Precise statement:

Source (hierarchical) DTD:

```
<!DOCTYPE DOC [
<!ELEMENT DOC       - O    (P+)>
<!ELEMENT P         - O    (#PCDATA|L)+      -- Paragraphs -->
<!ELEMENT L         - O    (I+)              -- Lists      -->
<!ELEMENT I         - O    (P+)              -- List items -->
]>
```

Target (flat) DTD:

```
<!DOCTYPE DOC [
<!ELEMENT DOC       - O    (P+)>
<!ELEMENT P         - O    (#PCDATA)         -- Paragraphs -->
<!ATTLIST P
     INDENT  (0|1|2|3)     0
     MARK    (YES|NO)      NO >
]>
```

When translating from the source DTD, the following rules should be followed:

- paragraphs in the source instance are translated to paragraphs in the target instance
- for leading paragraphs in list items in the source, the MARK attribute should take the value "YES" in the target instance (introductoty marker).
- the "INDENT" attribute should equal "0" for paragraphs outside lists in the source. INDENT=1 for paragraphs in list items at the first level, INDENT=2 for the the second level (embedded lists), INDENT=3 for the third level and beyond.

Sample source instance:

```
<DOC>
<P>Introductory paragraph</P>
<P>Second paragraph :
<L>
<I><P>First case</P><P>Second paragraph in item</P></I>
<I><P>Second case :
```

```
<L>
<I><P>First sub-case</P></I>
<I><P>Second sub-case</P></I>
</P>
</I>
</P>
</DOC>
```

Sample target instance:

```
<DOC>
<P>Introductory paragraph</P>
<P>Second paragraph :</P>
<P MARK=YES INDENT=1>First case</P>
<P MARK=NO INDENT=1>Second paragraph in item</P>
<P MARK=YES INDENT=1>Second case :</P>
<P MARK=YES INDENT=2>First sub-case</P>
<P MARK=YES INDENT=2>Second sub-case</P>
</DOC>
```

CoST solution

Converting from more to less hierarchy

Strategy Simple one-pass translation.

General approach Converting from a highly hierarchical DTD to a more “flat” one is not too difficult. The solution to the problem at hand is to maintain information about the nesting level of P elements; the nesting level is used to generate the value of the INDENT attribute in the target DTD.

Notes

- L elements always mark the end of P elements in the target DTD.
- The hardest problem is to avoid generating too many P end tags. In the algorithm shown, this is simply solved by explicitly making sure that just one end tag is generated for every PCDATA pseudo-element. A more elegant solution could probably be found.

```
itcl_class SHARED { common last }

element DOC {
  start { puts stdout "<DOC>" nonewline }
  end   { puts "\n</DOC>" }
}

element P {
  inherit SHARED
  common { nestlevel 0 }
  start {
    if {[rank]>1 && $last=="DATA"} {
      puts stdout "</P>" nonewline
```

```
      set last ETAG
    }
    puts stdout "\n<P" nonewline
    if {[within L]} {
      if {[rank]==1} {
        puts stdout " MARK=YES" nonewline
      } else {
        puts stdout " MARK=NO " nonewline
      }
      puts stdout " INDENT=$nestlevel" nonewline
    }
    puts stdout ">" nonewline
    incr nestlevel
  }
  end { incr nestlevel -1 }
}

element I {
  inherit SHARED
  end {
    if {$last=="DATA"} {
      puts stdout "</P>" nonewline
      set last ETAG
    }
  }
}

element L {
  inherit SHARED
  start {
    if {$last=="DATA"} {
      puts stdout "</P>" nonewline
      set last ETAG
    }
  }
}

TEXT {
  inherit SHARED
  start { puts stdout $data nonewline; set last DATA }
}

RECORDEND { start { puts stdout " " nonewline } }
```

Notable program features

- The method `rank` is used to set the correct value of the `MARK` attribute. `rank` returns 1 if the current element is the first child of its parent, 2 if it is the second child, etc.

Converting from less to more hierarchy

Strategy Two-pass internal translation.

General approach I found this problem quite difficult. The problem is not really that the facilities of CoST are insufficient to solve the problem (they aren't), but finding the correct algorithm! I believe the algorithm shown is correct, though.

The whole document is processed in `Treemode`, since access to information about both younger and older siblings of `P` elements in the source DTD is essential to the algorithm implemented.

Notes

- None.

```
element P {
  protected indent
  protected mark
  start {
    set indent [attrValue INDENT]
    set mark   [attrValue MARK]
    if {[ySibID]!=""} {
      if {[[ySibID] attrValue INDENT]<$indent} {
        puts "\n<L>"
      }
    }
    if {$mark=="YES"} { puts stdout "<I>" nonewline}
    puts stdout "<P>" nonewline
  }
  end {
    if {[oSibID]!=""} {
      if {[[oSibID] attrValue INDENT]<=$indent} {
        puts stdout "</P>" nonewline
        if {$indent>0 && [[oSibID] attrValue MARK]=="YES"} {
          puts "</I>"
        }
      }
      if {[[oSibID] attrValue INDENT]<$indent} {
        puts "</L>"
      }
    } else {
      puts "</P>"
      for {set i 0} {$i<$indent} {incr i} {
        puts "</I>\n</P>"
      }
    }
  }
}

element DOC {
  Treemode
  start { puts "<DOC>"  }
  end   { puts "</DOC>" }
}
```

Notable program features

- The methods `ySibID` and `oSibID` return the object ID of younger and older siblings respectively. In other words they provide access to all the methods that are defined on the objects that represent the closest siblings of an object. `oSibID` works only in `Treemode`.

- The method `attrValue` provides access to the values of attributes. In this example `attrValue` is used to access the attribute values of both younger and older siblings.

4.4.4 ID type checking & computation (AIS #4)

Problem description (by Christophe Lecluse, AIS)

An SGML parser checks uniqueness of ID attributes in the instance and checks that IDREF attributes are resolved, but this does not guarantee by itself correctness of the cross-references.

In the cases when cross-references are "typed" (i.e. a cross-reference to a figure is not the same as a cross-reference to a table), checking the type of elements associated to the ID/IDREF attribute pairs provides an additional checking level.

This problem examines how to handle such a task.

As an auxiliary task, it is asked to fill implied ID attributes.

Precise statement

The following simplified DTD will be used:

```
<!DOCTYPE DOC [
<!ELEMENT DOC        - O     (TITLE, BODY) +(FIG|FN)>
<!ELEMENT BODY       - O     (SEC1,SEC1+)                -- Body --   >
<!ELEMENT SEC1       - O     (TITLE, P*, (SEC2,SEC2+))   -- level 1 -->
<!ELEMENT SEC2       - O     (TITLE, P*, (SEC3*))        -- level 2 -->
<!ELEMENT SEC3       - O     (TITLE, P*, (SEC4*))        -- level 3 -->
<!ELEMENT SEC4       - O     (TITLE, P+)                 -- level 4 -->
<!ATTLIST (SEC1|SEC2|SEC3|SEC4)
     ID      ID      #IMPLIED>
<!ELEMENT P          - O      (#PCDATA|FNREF|FIGREF|SECREF)*>
<!ELEMENT FIG        - O     (TITLE)                     -- Figure --  >
<!ATTLIST FIG
     ID      ID      #REQUIRED>
<!ELEMENT FN         - O     (P*)    -(FN)               -- Footnote -->
<!ATTLIST FN
     ID      ID      #REQUIRED>
<!ELEMENT (FNREF|FIGREF|SECREF) - O  EMPTY>
<!ATTLIST (FNREF|FIGREF|SECREF)
     REFID   IDREF   #REQUIRED>
]>
```

Exercise 1:

This DTD allows refrences to footnotes (FNREF -> FN), to figures (FIGREF -> FIG), and to sections (SECREF -> SECx). Check the ID/IDREF pairs to ensure that each type of cross-reference points to the adequate target type.

Exercise 2:

In this DTD, ID attributes for SECx elements are implied. If an ID attribute value for some section level is not given in the instance, then fill the hole by generating an ID value. The rule to be followed is : ID="Sn" for level 1 (SEC1), where n is the rank in the document, ID="Sn.p" for level 2 (SEC2), ID="Sn.p.q" for level 3 (SEC3), etc.

Option 1: assume that collisions never occur between newly added and already defined IDs.

Option 2: prevent collisions by also modifying existing Ids together with all IDREFs refering to them.

CoST solution

ID type checking

Strategy Tree-mode, internal two-pass translation.

General approach It isn't really necessary to handle this problem in two passes. For large documents it would probably be more reasonable to use a one-pass approach which simply collects the GI of the elements with an `ID` attribute and the GI of every element that has a `REFID` attribute, and then check at the end if the types of the cross references are what they should be. Nevertheless, the convenient approach which uses `Treemode` and two passes was chosen for this problem so the cross references can be checked as they are encountered in pass two.

Notes

- None.

```
itcl_class SHARED {
  Set fnrefs
  Set figrefs
  Set secrefs
  Set ids
  common idgis
  common errors 0
}

element DOC {
  inherit SHARED
  Treemode
  pass1Start {}
  end { puts "$errors errors found." }
}

element {SEC1 SEC2 SEC3 SEC4 FIG FN} {
  inherit SHARED
  pass1Start {
    if {![isImplicit ID]} {
      ids insert [attrValue ID]
      case [GI] in {
        {FN FIG} { set idgis([attrValue ID]) [GI] }
        default  { set idgis([attrValue ID]) SEC  }
      }
    }
  }
}
```

```
element {FNREF FIGREF SECREF} {
  inherit SHARED
  start {
    if {[ids isMember [attrValue REFID]]} {
      case [GI] in {
        FNREF   { set corrval "FN"  }
        FIGREF  { set corrval "FIG" }
        default { set corrval "SEC" }
      }
      if {$idgis([attrValue REFID])!=$corrval} {
        puts stdout "[GI] REFID=[attrValue REFID] xref " nonewline
        puts stdout "to GI=$idgis([attrValue REFID])."
        incr errors
      }
    } else {
      # SGML error that the parser will report
    }
  }
}

TEXT      { start {} }
RECORDEND { start {} }
```

Notable program features

- None.

ID computation/generation

Strategy Tree-mode, internal two-pass translation.

General approach The basic problem here is to make sure that the new `ID` attribute value is accessible when the start tag is generated for reference elements (`SECREF`, `FIGREF`, `FNREF`). This is done by computting all the new ID attribute values for section elements in pass one, and generating the output in pass two. The new attribute value is stored in a global array (`idtable`) which is accessed when the start tag is generated.

Notes

- Only elements with explicit `ID` attribute values are candidates as targets of cross-references, so only they are stored in the table.

```
element DOC {
  Treemode
  start { puts "<DOC>" }
  end   { puts "\n</DOC>" }
}
```

```
element {SEC1 SEC2 SEC3 SEC4} {
  protected  newid
  common     {SEC1 1}
  common     {SEC2 1}
  common     {SEC3 1}
  common     {SEC4 1}
  pass1Start {
    global idtable
    set gi [GI]
    if {$gi=="SEC1"} { set SEC2 1 }
    if {$gi=="SEC1" || $gi=="SEC2"} { set SEC3 1 }
    if {$gi=="SEC1" || $gi=="SEC2" || $gi=="SEC3"} {
      set SEC4 1
    }
    case $gi in {
      SEC1 { set newid "S$SEC1" }
      SEC2 { set newid "S$SEC1.$SEC2" }
      SEC3 { set newid "S$SEC1.$SEC2.$SEC3" }
      SEC4 { set newid "S$SEC1.$SEC2.$SEC3.$SEC4" }
    }
    if {![isImplicit ID]} {
      set idtable([attrValue ID]) $newid
    }
    incr [GI]
  }
  start { puts stdout "<[GI] ID=\"$newid\">" nonewline }
  end { puts stdout "</[GI]>" }
}

element SECREF {
  start {
    global idtable
    puts "<SECREF REFID=\"$idtable([attrValue REFID])\">"
  }
}

element {FIGREF FNREF} {
  start { puts stdout "<[GI] REFID=\"[attrValue REFID]\">" nonewline }
}

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

Notable program features

- None.

4.4.5 Floating elements consistency check (AIS #5)

Problem description (by Christophe Lecluse, AIS)

Floating (empty) elements are often used in DTDs to mark up revisions or internationalization. These empty elements occur in pairs with a "starting" and an "ending" element.

When such elements are declared floating, an SGML parser cannot check much about the way they are used. This problem examines how to handle that task with an application language.

Precise statement

We consider the same DTD as in Problem 4, but with two floating elements BEGREV and ENDREV.

```
<!DOCTYPE DOC [
<!ELEMENT DOC       - O     (P+)    +(BEGREV|ENDREV)>
<!ELEMENT P         - O     (#PCDATA|L)+      -- Paragraphs -->
<!ELEMENT L         - O     (I+)              -- Lists      -->
<!ELEMENT I         - O     (P+)              -- List items -->
<!ELEMENT BEGREV    - O     EMPTY             -- Revision mark - start -->
<!ELEMENT ENDREV    - O     EMPTY             -- Revision mark - end  -->
]>
```

The instance will be considered correct iff BEGREV/ENDREV marking has the following properties:

- BEGREV and ENDREV can be paired with no embedding of BEGREV/ENDREV
- The ENDREV corresponding to a BEGREV is located in the same element as the BEGREV.

Below are examples of a correct and incorrect instance.

A correct instance :

```
<DOC>
<P>Introductory <BEGREV>paragraph<ENDREV></P>
<P>Second paragraph :
<L><I><P>First case</P><P>Second paragraph in item</P></I>
<I><P>Second case :
<L><I><P>First sub-case</P></I>
<BEGREV><I><P>Second sub-case</P></I><ENDREV>
</P>
</I>
</P>
</DOC>
```

An incorrect instance:

```
<DOC>
```

```
<P>Introductory <ENDREV> <BEGREV>paragraph<ENDREV></P>
<P>Second paragraph :
<L><I><P>First case</P><P>Second paragraph in item</P></I>
<I><P>Second case :
<L><I><P>First sub-case</P></I>
<BEGREV><I><P>Second sub-case<ENDREV></P></I>
</P>
</I>
</P>
</DOC>
```

CoST solution

Strategy One-pass translation.

General approach This exercise consists of two sub-problems, (1) to make sure that the BEGREV and ENDREV elements alternate correctly (beginning with a BEGREV); and (2) to check if each BEGREV/ENDREV pair has the same parent. The first problem is handled by maintaining information about the last BEGREV or ENDREV seen (in the variable prev) and about the parent of the last BEGREV seen (in the variable Bparent).

Notes

- The problem description is a little unclear. That BEGREV and ENDREV are required to be located in the same element is assumed to mean that they are required to have the same parent.

```
itcl_class SHARED {
  common prev ENDREV
  common Bparent ""
}

element {BEGREV ENDREV} {
  inherit SHARED
  start {
    if {[GI]=="BEGREV"} {
      if {$prev!="ENDREV"} {
        puts stderr "BEGREV followed by BEGREV."
      }
      set Bparent [parentID]
    } else {
      if {$prev!="BEGREV"} {
        puts stderr "ENDREV out of sequence."
      }
      if {[parentID]!=$Bparent && $Bparent!=""} {
        puts stderr "BEGREV-ENDREV pair not in same element."
      }
    }
    set prev [GI]
  }
}

RECORDEND { start {} }
TEXT      { start {} }
```

Notable program features

- The declarations for `TEXT` and `RECORDEND` override the default declarations, which produce output. For this problem we don't want any output except error messages if the floating elements aren't placed in accordance with the specification.

4.4.6 CALS to AAP Table conversion (AIS #6)

Problem description (by Christophe Lecluse, AIS)

The general problem consists in transforming SGML structured tables from the CALS table DTD to the AAP table DTD.

The transformation program should handle

- the general structure of the table
- spanning
- borders

The program should be DTD independent. That is, given a DTD including CALS table description, the program can take any instance of this DTD and output the given instance where all CALS tables have been replaced by AAP tables.

Precise statement

The precise DTDs that we take are the CALS table DTD and the SoftQuad A/E table DTD which is a variant of the AAP DTD. The precise semantics of all CALS & AAP elements and attributes cannot be detailed here but can be found in the litterature.

(The AAP and CALS table DTDs and example instances were omitted. -KH)

CoST solution

Knowing next to nothing about these rather large table DTDs, I have decided not to attempt to write conversions between them.

A The CoST program used to create this document

As an illustration of a practical use of CoST, the CoST program used to format this document is shown.

The document was created using the P2 DTD of the Text Encoding Initiative. The CoST program generates a LaTeX file which is subsequently formatted and printed.

```
element TEI.2 {
  start {
    global putout
    set putout 0
  }
}

element BODY {
  start {
    global putout
    set putout 1
    puts stdout "\n\\begin\{document\}"
  }
  end { puts stdout "\\end\{document\}" }
}

element DIV0 {
  start { puts stdout "\n\\newpage\n\\section" nonewline }
  end   { puts stdout "" }
}

element DIV1 {
  start { puts stdout "\n\\subsection" nonewline }
  end   { puts stdout "" }
}

element DIV2 {
  start { puts stdout "\n\\subsubsection" nonewline }
  end   { puts stdout "" }
}

element DIV3 {
  start { puts stdout "\n\\par" nonewline }
  end   { puts stdout "" }
}

element DIV4 {
  start { puts stdout "\n\n\\noindent" nonewline }
  end   { puts stdout "" }
}

element HEAD {
  start {
    when {
      {in LIST} {
        puts stdout "\n\\item\[ \] \{\\Large\\bf " nonewline
      }
      {in DIV3} { puts stdout "\{\\bf " nonewline }
      {in DIV4} { puts stdout "\{\\it " nonewline }
      default   { puts stdout "\{" nonewline }
    }
```

```
    }
    end {
      when {
        {in LIST} { puts stdout "\}" }
        default   { puts stdout "\}" nonewline }
      }
    }
}

element P { start { puts "\n" } }

element QUOTE {
  start {
    puts stdout "\\begin\{quotation\}"
    puts stdout "``" nonewline
  }
  end {
    puts stdout "''\\end\{quotation\}"
  }
}

element CODESAMPLE {
  start {
    if {[attrValue NTN]=="cost"} {
      set file [attrValue FILE]
      puts "\\rule\{\\textwidth\}\{.3mm\}"
      puts "\\input\{$file-tex\}"
      puts "\\rule\{\\textwidth\}\{.3mm\}"
    }
  }
}

element PROGRAM {
  start { puts "\{\\footnotesize\\begin\{verbatim\}" }
  end   { puts "\\end\{verbatim\}\}" }
}

element FIG {
  start {
    puts stdout "\\begin\{figure\}"
    puts stdout "\\centerline\{\\hbox\{" nonewline
    puts stdout "\\epsfxsize[attrValue SIZE]truecm" nonewline
    puts stdout "\\epsffile\{" nonewline
    puts stdout "[attrValue FIGFILE]-ps" nonewline
    puts stdout "\}\}\}"
    if {![isImplicit CAPTION]} {
      puts stdout "\\caption\{[attrValue CAPTION]\}"
    }
    puts stdout "\\end\{figure\}"
  }
}

element SOCALLED {
  start { puts stdout "``"          nonewline }
  end   { puts stdout "''"          nonewline }
}

element NOTE {
  start { puts stdout "\\footnote\{" nonewline }
  end   { puts stdout "\}"          nonewline }
}

element EMPH {
  start { puts stdout "\{\\it "     nonewline }
```

```
  end   { puts stdout "\\/\}"        nonewline }
}

element HI {
  start { puts stdout "\{\\large\\tt "      nonewline }
  end   { puts stdout "\}"           nonewline }
}

element LIST {
  protected listtype
  start {
    when { {in ITEM in LIST} { puts "\\rule\{\\textwidth\}\{0pt\}" } }
    puts stdout "\\begin\{" nonewline
    if {[attrIsImplied TYPE]} {
      set listtype itemize
    } else {
      case [attrValue TYPE] in {
        ordered   { set listtype enumerate }
        unordered { set listtype itemize }
        gloss     {
          set listtype description }
        default   {
          set listtype itemize }
      }
    }
    puts stdout "$listtype\}" nonewline
  }
  end {
    puts stdout "\n\\end\{" nonewline
    puts stdout "$listtype\}"
  }
}

element ITEM {
  start {
    if {$__predType!="LABEL"} {
      puts stdout "\n\\item " nonewline
    }
  }
}

element LABEL {
  start { puts stdout "\n\\item\[" nonewline  }
  end   { puts stdout "\] " nonewline }
}


TEXT {
  start {
    global putout
    if {$putout!=0} {
      puts stdout "$data" nonewline
    }
  }
}

SDATA {
  start {
    case $entname  {
      {parsep}   { puts stdout "\n\n"      nonewline }
      {nl}       { puts stdout "\\\\"      nonewline }
      {hashmark} { puts stdout "\\#"      nonewline }
      {dollar}   { puts stdout "\\$"      nonewline }
      {uline}    { puts stdout "\\_"      nonewline }
```

```
        {lt}       { puts stdout "<"        nonewline }
        {amp}      { puts stdout "\\&"      nonewline }
        {percnt}   { puts stdout "\\%"      nonewline }
        {lsqb}     { puts stdout "\["        nonewline }
        {rsqb}     { puts stdout "\]"        nonewline }
        {lcub}     { puts stdout "\\\{"     nonewline }
        {rcub}     { puts stdout "\\\}"     nonewline }
        {eacute}   { puts stdout "\{\\'e\}" nonewline }
        {appendix} { puts stdout "\\appendix" nonewline }
        default    { puts stdout "?" nonewline }
      }
    }
  }
```