ELECTRONIC PUBLISHING, VOL . 2(2), 65–90 (DECEMBER 1989)

The implementation of the Amsterdam SGML Parser

JOS WARMER AND SYLVIA VAN EGMOND
Faculteit Wiskunde en Informatica
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
The Netherlands

SUMMARY

The Standard Generalized Markup Language (SGML), is an ISO Standard that specifies a language for document representation. This paper gives a short introduction to SGML and describes the Amsterdam SGML Parser and the problems we encountered in implementing the Standard. These problems include interpretation of the Standard in the places where it is ambiguous and the technical problems in parsing SGML documents.

KEY WORDS SGML Structured documents Document preparation Parser generators

INTRODUCTION

When a complex document is to be printed on a typesetter or laserprinter, the usual method is to use a formatter to process the document. The input to such a formatter consists of the text of the document interspersed with formatting commands. These formatting commands are called *mark-up*. In powerful formatters such as TEX [1] or *troff* [2] the user usually specifies the formatting commands. Using *troff*, for example, each line starting with a dot ('.') is interpreted as a formatting command. The start of this paper might be marked up as shown in Figure 1. Notice that the formatting commands are often cryptic and of a very low-level nature.

The mark-up provided by TEX or *troff* is called *procedural mark-up* which involves specifying the processing instructions within the text. The user is mainly concerned with the application that is going to process the document (in this case the formatter) and thinks in terms of 'insert a blank line' and 'start a new page' rather than 'start a new paragraph' and 'start a new section'. Changing to another formatter, or using the document in some other application (e.g. in a database), may make it necessary to change all the mark-up.

To overcome this problem many formatters provide a macro facility. A macro package identifies the different structural elements of a document and for each element a macro is defined which contains the low level formatting commands. A well known macro package for *troff* is *ms* [3], and, using the *ms* macros TL, AU, AI, AB and AE, the previous mark-up could be simplified to something along the lines shown in Figure 2.

The use of high-level macros simplifies the mark-up process, but several problems still remain. For instance, there is no way to determine whether the macros are used

0894–3982/89/020065–26$13.00

Received 15 July 1988
Revised 3 April 1989

```
.bp                  \" start a new page
.ps 12               \" pointsize is set to 12
.ft B                \" fontstyle is bold
.ce                  \" centre one line
The Implementation of the Amsterdam SGML Parser
.sp 1                \" insert a blank line
.ps 10
.ft I                  \" fontstyle is italic
.ce
JOS WARMER AND SYLVIA VAN EGMOND
.ft R                  \" fontstyle is roman
.sp 0.5
.ce 4
Faculteit Wiskunde en Informatica
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
The Netherlands
e-mail: jos@cs.vu.nl
.sp 3
.ft I
.ce
SUMMARY
.sp 2
.ft R
.ti 0.5              \" temporary indent for next text line
The Standard Generalized Markup Language (SGML), is an ISO
standard that specifies a language for document
representation. This paper gives a ....
```

Figure 1. Document marked up with troff code

```
.TL                                  \" title
The Implementation of the Amsterdam SGML Parser
.AU                                  \" authors
JOS WARMER AND SYLVIA VAN EGMOND
.AI                                  \" affiliation
Faculteit Wiskunde en Informatica
Department of Mathematics and Computer Science
Vrije Universiteit
Amsterdam
The Netherlands
e-mail: jos@cs.vu.nl
.AB                                  \" start abstract
The Standard Generalized Markup Language (SGML), is an
ISO standard that specifies a language for document
representation. This paper gives a ....
.AE                                  \" end abstract
```

Figure 2. Document marked up with ms macros

consistently; the TL macro can be used anywhere in the document, and not just for the title page; the AE macro, which denotes the end of the abstract, can be forgotten or can occur when no abstract has been started. The user can still use low-level formatting commands in the text, and these can interfere with the correct operation of the macro package.

To overcome these problems, a new, more rigorous, approach is needed which will enforce the consistency of mark-up. Such an approach is taken by SGML. SGML (**S**tandardized **G**eneralized **M**arkup **L**anguage) is an ISO Standard, issued in October 1986 [4, 5] and, as its name suggests, it specifies how mark-up can be incorporated into a document. An introductory survey of SGML is given in [6] and a more elaborate description in [7].

This paper discusses some of the problems we encountered while implementing the Amsterdam SGML Parser. The first section, below, gives a short introduction to SGML and describes the task of an SGML parser in processing a document marked up using SGML. The second section discusses some special features of SGML, the errors or deficiencies in the Standard concerning these features and the design decisions we had to make. This section also describes the 'back end' of the Amsterdam SGML Parser, which can be used to convert SGML documents to *troff* or TEX documents. The third section is more technical and describes the most important implementation problems and how we solved them. In the final section some conclusions are drawn, concerning SGML, the SGML Standard, and our implementation.

Introduction to SGML

SGML is a language which is used on two levels. At the first level, it is used to write a document type definition (DTD) which defines the structure of a document. An example of a DTD is given in Figure 3.

```
<!doctype memo [
<!element memo           - O (sender, receiver, contents)>
<!element sender         O O (person) >
<!element receiver       O O (person)+>
<!element person         O O (nickname | (forename?, surname))>
<!element (forename,
           nickname,
           surname)      O O (#PCDATA)>
<!element contents       O O (#PCDATA)>
]>
```

Figure 3. A simple document type definition

The first line specifies that this DTD describes a document type called `memo`. The second line says that a `memo` consists of a sequence-group of three consecutive elements: `sender`, `receiver` and `contents`. The `sender` consists of one `person`, and a `receiver` of one or more persons. A `person` is an or-grouping of a `nickname` and an optional `forename` followed by a `surname`. All other elements are defined as PCDATA, which means they consist of data characters. A line starting with `<!element` is called an element declaration. Note that a DTD describes the logical

structure of a document, using a formal grammar, and that it serves to name the structural elements and their relationships. It cannot say anything about the layout of the document or the meaning of the elements.

At a second level SGML can be used to describe a document because, as we shall see, the document's structure is indicated by suitable start-tags and end-tags. This actual structure must satisfy the formal structure given in the DTD; as an example of this, Figure 4 shows a document which conforms to the DTD shown in Figure 3.

```
<memo>
  <sender>
    <person>
      <forename>Jos</forename>
      <surname>Warmer</surname>
    </person>
  </sender>
  <receiver>
    <person>
      <forename>Sylvia</forename>
      <surname>van Egmond</surname>
    </person>
  </receiver>
  <contents>
    Tomorrow's meeting will be postponed.
  </contents>
</memo>
```

Figure 4. A document marked up with SGML

The start and end of an element are denoted as follows: any text between '<' and '>' is called the *start-tag*, and text between '</' and '>' is an *end-tag*. The indentation in Figure 4 is not mandatory, but is used to make the structure clear.

In the SGML scheme a document type definition describes a **class** of documents, and any particular document conforming to that DTD is an **instance** of such a class.

Typing a document with all these start- and end-tags can be very tedious and SGML overcomes this problem by defining several rules for omitting tags. The main idea behind these rules is that if only one type of tag is possible in a given context, then that tag may be omitted. However, the fact that a tag might be omitted has to be stated when declaring an element; for example, in the declaration of `memo` (Figure 3), the '`-`' denotes that the start-tag may not be omitted, while the '`O`' denotes that the end-tag may be omitted. In SGML this is called the *OMITTAG* feature (see next section). For all other elements the start-tag as well as the end-tag may be omitted. Figure 5 shows the same memo as Figure 4, but now with more of the tags omitted. This is a far more readable version and is much easier for humans to type.

There are a number of other features in SGML which can also be used to ease the typing of a document, or to add additional information to an element. These include ways of defining abbreviations of tags (called the *SHORTTAG* feature) and plain text. The more important of these features are described in the next section. Despite the undoubted usefulness of these abbreviations the long form of mark-up is the easier form to use whenever a document is to be processed by a computer program because the

```
<memo>
      <forename>Jos
      <surname>Warmer
  <receiver>
      <forename>Sylvia
      <surname>van Egmond
  <contents>
    Tomorrow's meeting will be postponed.
</memo>
```

Figure 5. The same document as in Figure 4 but with tag omission

start and end of each element are then clearly marked. This greatly reduces the complexity of processing the document.

Because a DTD is a formal description, the process of verifying the structure of a document can be done automatically. A program that checks whether a document conforms to a DTD is called an SGML parser and such a parser is a necessary tool if the full advantage is to be gained from SGML's rigorous approach.

Structure of the parser

This section describes the rôle of an SGML parser in processing documents that are marked up using SGML. An SGML parser takes a DTD and a document as its inputs and delivers a 'complete' document as output. The rôle of the parser in this is twofold. Firstly the parser must check whether the DTD is correct: that is, the DTD must obey all rules and restrictions that are specified in the Standard. Secondly, the parser checks whether the document conforms to the DTD and, if it does, the parser then provides all the tags in the document that the user omitted. For example, the document of Figure 5 would be transformed to the 'complete' document of Figure 4 by this process.

The general structure of an SGML parser is shown in Figure 6.

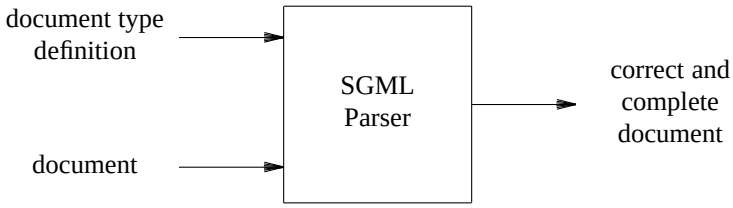

Figure 6. Structure of an SGML parser

Alternatively, because checking the DTD and checking the document are two separate processes, an SGML parser can be divided in two parts. The first part, called the *dtd-parser*, checks the DTD, and generates a parser for the corresponding class of documents. This generated parser, called the *document parser*, checks a document according to the document class, and delivers the complete document. This second approach is shown in Figure 7.

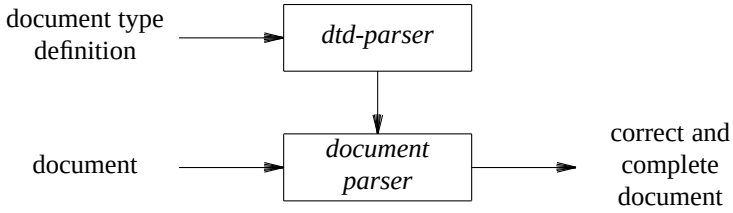

Figure 7. An SGML parser divided in two parts.

Both of these alternatives have their advantages and disadvantages. The first approach is more appropriate in an environment where DTD's change rapidly or where each user writes his own DTD. The disadvantage is that processing a document takes a relatively long time, because processing of each document will include processing of the DTD.

The second approach is more appropriate in an environment where a number of standard DTD's are used. If an executable binary version of each document parser is the only publicly available software then the users will not be able to use a different DTD, which will guarantee the actual use of standard document classes. Another advantage is that the document parser can be optimized for the particular class of documents, which can make it much faster. If many documents are to be processed this advantage becomes greater.

Implementation of the parser

Another advantage of the second approach is the possibility of making more use of existing parser-generators. In the first approach, the DTD is scanned and stored in some internal tables or other data structures. While parsing the document these tables have to be examined to see whether or not an element is allowed at some particular place in the document.

In the second approach the dtd-parser can generate input for an existing parser-generator which will then produce the actual document parser. As will be shown later, the DTD-grammar in SGML must conform to a notion of unambiguity which closely resembles the LL(1) conditions [8, 9]. This led us to the conclusion that it should be possible to use an existing LL(1) parser-generator and this is the main reason why we have chosen the approach of Figure 7. We believe that this approach saves a lot of work and creates efficient document parsers.

The parser-generator we use is *LLgen*[10], an extended LL(1) recursive descent parser-generator, developed at the Vrije Universiteit, which is known to produce efficient parsers. LLgen takes as input an extended LL(1) grammar, with semantic rules in the C language [11], and produces a C program as output. LLgen is part of ACK — the Amsterdam Compiler Kit [12] — and parsers for Pascal, C, Modula2 and Occam have been written in LLgen.

The element declarations in a DTD are written in a form which closely resembles the form of an LLgen rule. On the face of things it only requires a simple syntax change to convert the DTD to LLgen. For example, the DTD from Figure 3, when rephrased in LLgen, becomes:

```
%start memo, memo;

memo        : sender receiver contents ;
sender      : person  ;
receiver    : person+ ;
person      : [nickname | [forename? surname]] ;
forename    : PCDATA ;
nickname    : PCDATA ;
surname     : PCDATA ;
contents    : PCDATA ;
PCDATA      : data_character* ;
```

However, we shall show that it generally takes rather more than just a syntax change to rewrite the element declarations into LLgen rules.

The complete structure of the Amsterdam SGML Parser, including the intermediate steps, is shown in Figure 8. As one can see from the figure, LLgen is not only used in the generation of the document parser but also in the implementation of the dtd-parser. The rules in the Standard which describe what a document type definition looks like are in the same form as the LLgen rules. It took very little effort to make these rules LL(1), and implementing them in LLgen was easy. In this way the dtd-parser itself is generated from the SGML-rules defined in the Standard.

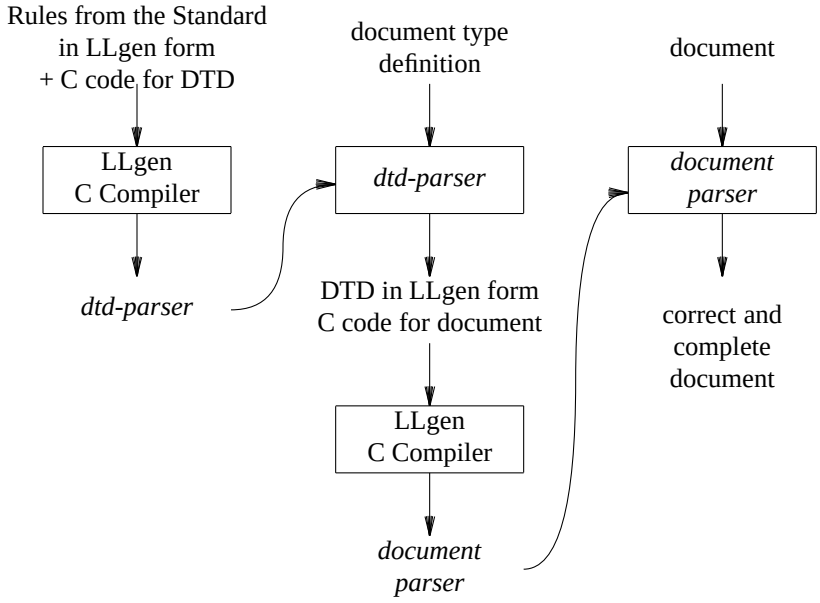

Figure 8. Structure of the Amsterdam SGML Parser

SPECIFICATION OF THE AMSTERDAM SGML PARSER

The functionality of an SGML parser is defined in the *SGML declaration*. This declaration may precede the DTD. In an SGML declaration all possible features are

given by their name followed by the keyword, **YES** or **NO**, to indicate whether the feature is supported or not. Examples of features are tag omission (see section 1) and tag minimization denoted by **OMITTAG** and **SHORTTAG** respectively.

The SGML declaration also specifies numerical limits, such as the maximum number of declared elements and the character sets used. If an SGML declaration uses the *reference capacity set* for the numerical limits, the *reference concrete syntax* for the used character sets, and supports only the **OMITTAG** and **SHORTTAG** features, it is said to support *basic SGML*. The Amsterdam SGML Parser is a conforming parser for basic SGML.

In our opinion basic SGML contains the most useful features. The other features in SGML are exotic and seldom used. For instance, the feature **SUBDOC** specifies that within a document a subdocument may occur. This subdocument is marked up according to its own DTD. The DTD of the subdocument and the DTD of the main document can differ. This feature may be of use in special cases, but in general it will not be used. The Association of American Publishers has developed three standard DTD's using basic SGML [13, 14]. These three DTD's define a book, an article and a serial. This shows that basic SGML is powerful enough for standard applications.

In later sections we describe several characteristics of basic SGML, especially those characteristics that are ambiguously or incompletely described in the Standard, and the subsequent design decisions made by us. We then go on to describe the kind of error checking performed in the Amsterdam SGML Parser, and the difficulties encountered in the Standard when implementing basic SGML. Finally, the output of a document parser is described.

Validation services

The DTD and the document are both checked for errors. The DTD is also checked for inconsistencies, such as doubly-declared elements. The Standard, in Section 15.6.2, specifies so-called *validation services* which formally define the checks performed by an SGML parser. The following is a list of validation services, each with a short explanation. **YES** indicates that the validation service is provided by the Amsterdam SGML Parser, **NO** that it is not.

GENERAL (**YES**):

> All syntax errors in the DTD and the document are reported. However, as a result of one error several other errors may occur. The parser tries to resume parsing after an error has occurred by adding or deleting input until a token is found which fits the grammatical constraints [15]. All semantic checks described throughout the Standard are performed and an error message is generated when an error is encountered. Examples of semantic errors might be a reference to an undeclared element or to a doubly-declared element. Exceeding a numerical restriction on some aspect of the mark-up language, such as the length of a name, results in a warning message.

MODEL (**YES**):

> The element declarations define the structure of a document in a so-called *content model*. This structure may not be ambiguous (an explanation of the ambiguities which are recognized is given in a later section). The Amsterdam SGML Parser checks the structure for ambiguities and reports them.

EXCLUDE (**NO**):
: Exclusions that change the structure of a document are not reported. Owing to a limited amount of time and the fact that exclusions are rare, we have chosen not to implement these checks, yet. Note, however that the parser does handle exclusions and inclusions.

CAPACITY (**YES**):
: There are some numerical limits (capacities) on the complexity of the DTD and the document. These capacities are expressed as a number of *capacity points*. The Amsterdam SGML Parser collects these points and gives an error message when some value is exceeded. For instance, each declared element generates a constant number of points (8 for basic SGML) which are counted and the total number of points may not exceed *ELEMCAP* (35 000 in basic SGML). Each capacity derived from a specific DTD may not exceed its maximum value. Also the grand total of all the capacities may not exceed *TOTALCAP* (35 000 in basic SGML).

SGML (**YES**):
: Every DTD may start with an SGML declaration. If none is present, the basic SGML declaration is taken. The SGML declaration is checked for syntax errors and it is checked whether the SGML declaration conforms to basic SGML.

NONSGML (**YES**):
: The occurrence of **NONSGML** characters will be reported. **NONSGML** characters are those that are marked *UNUSED* in the SGML character set description.

FORMAL (**NO**):
: A public identifier denotes where public information, such as public character sets, can be found. If a public identifier conforms to the syntactic requirements as specified in the Standard (section 10.2), it is said to be *FORMAL*. All public identifiers must be formal public identifiers when the feature **FORMAL** is provided in the SGML declaration. Since this feature is not provided in basic SGML, a public identifier need not be *FORMAL*. This implies that the syntax of a public identifier is not restricted and no checking is done.

Entities

Entities can be regarded as a substitution mechanism and an entity declaration associates a name with a text. The text is substituted on encountering the entity name in the document. Take for example the following entity declaration:

```
<!entity asp    "the Amsterdam SGML Parser" >
```

The word ‘asp’ can still be used in the document in the usual way. It is only substituted when the word is preceded by ‘&’ and followed by ‘;’. For example the sentence: ‘This is &asp;.’ resolves to ‘This is the Amsterdam SGML Parser.’ The string `&asp;` is called an entity reference.

It is possible to define an entity whose substituting text is not specified in the entity declaration itself: such entities are called external entities. External entities have two associated identifiers, ‘SYSTEM’ and ‘PUBLIC’, which indicate where the text can be found in the system. SGML does not specify the correspondence between the identifiers and the text they represent. In the Amsterdam SGML Parser we have decided that one of

the identifiers must represent a file name and the contents of the named file are included in the document when the entity reference is encountered.

Short references

A short reference delimiter is a shorthand notation for an entity reference. If the above defined entity reference were associated with the short reference delimiter #, it would be sufficient to type # instead of the string `&asp;`.

There are 32 short reference delimiters defined in basic SGML. Every short reference delimiter can be associated with an entity. Short reference delimiters are for instance '#' and 'TAB'. For every element of a DTD a *short reference map* may be associated. This map defines which short reference delimiters are applicable within an element and, for every applicable delimiter, it specifies the associated entity. At the start of an element, the associated map becomes active and the short references defined in this map become applicable.

When a short reference delimiter is encountered, the text of the associated entity is substituted. According to the Standard, the longest replacement takes precedence in the case where there is a conflict (i.e., two or more short references are possible) but it also states that the most specific delimiter is to take precedence. The Standard does not describe what to do in the case of a further conflict between these two rules. For example:

A blank is a tab or a space character, so a blank sequence consists of spaces and tabs. *TAB*, *SPACE* and *blank sequence* are valid short reference delimiters in basic SGML. The sequence 'tab space' will be recognized as the *blank sequence* short reference delimiter, if the longest replacement takes precedence. But the same sequence will be recognized as a *TAB* followed by a *SPACE* delimiter if the most specific replacement is to be preferred. To solve this problem we have chosen that the longest replacement will always take precedence.

Capacity points

The capacities restrict the complexity of the DTD and the document. These capacities must be calculated and checked to ensure that they do not exceed the maximum value. However, the Standard gives no explanation about how the capacities should be calculated.

For instance, the capacity *ELEMCAP* is used to count the number of defined elements, and the capacity *ATTCAP* counts the number of the defined attributes. When the DTD is read, *ELEMCAP* and *ATTCAP* can be calculated. There is however one problem.

```
<!element (a, b, c)     - - (#PCDATA) >
                                          number of elements equals 3
<!attlist (a, b, c)     name    NAME      "jos" >
                                          number of attributes equals 1
```

Figure 9. Counting capacity points for a DTD

Does Figure 9 define three elements and three attributes, or just one element and one attribute? The Standard gives no explanation, but the example in Annex E does (Annex

E, however, is not an integral part of the Standard). Annex E makes it clear that in Figure 9 **three** elements are defined but only **one** attribute. This is not very logical because if, in Figure 9, all the attributes and elements are declared separately (Figure 10), the number of attributes is then 3 instead of 1, while the number of elements will remain the same.

```
<!element a     - - (#PCDATA) >          number of elements equals 3
<!element b     - - (#PCDATA) >
<!element c     - - (#PCDATA) >

<!attlist a     name    NAME    "jos" > number of attributes equals 3
<!attlist b     name    NAME    "jos" >
<!attlist c     name    NAME    "jos" >
```

Figure 10. Counting capacity points for an extended DTD

However, when counting the capacity points for all the attributes as though they where declared separately, the maximum is very quickly exceeded. We therefore have chosen to do the same as in Annex E and count the attributes and elements differently when several elements and attributes are defined in one declaration.

Data characters

In SGML there are three different types of data characters: **CDATA** (character data), **RCDATA** (replaceable character data) and **PCDATA** (parsable character data). When an element is declared as **CDATA** all the characters after the start-tag of the element are data characters. No mark-up is recognized, except for end-tags. When an element is declared as **RCDATA**, entity references and character references (decimal character codes) are also recognized. For example see Figure 11.

DTD:

```
      <!element a       - - CDATA>
      <!element b       - - RCDATA>
      <!entity asp      "the Amsterdam SGML Parser">
```

Input text:

```
<a>
This is &asp;
</a>

<b>
This is &asp;
</b>
```

text after parsing

```
<a>
This is &asp;
</a>

<b>
This is the Amsterdam SGML
                        Parser
</b>
```

Figure 11. The difference between CDATA and RCDATA

The third type of character data is **PCDATA**. **CDATA** and **RCDATA** are terminated by an end-tag but the Standard does not specify when **PCDATA** terminates. The Standard only makes clear that all characters which are not recognized as mark-up

delimiters are part of **PCDATA**. But characters that are recognized as mark-up delimiters present a problem. The following are the three different ways to interpret the Standard in the case where **PCDATA** contains a mark-up character.

(1) A mark-up character terminates **PCDATA** because, according to the definition, **PCDATA** contains zero or more characters that are not recognized as mark-up.
(2) A mark-up character that resolves to data characters (for instance entity references and short reference delimiters) is part of **PCDATA**. Other mark-up characters that do not resolve to data characters terminate **PCDATA**.
(3) All characters between two successive tags are part of **PCDATA**.

In Figure 12 a processing instruction is used to make the difference clear. A processing instruction contains system-specific information about how the document is to be processed.

DTD:

```
<!element a       - - (#PCDATA, q) >
<!element q       - - (#PCDATA) >
<!entity  ent     "entity-text">
```

document:

```
<a>
This is &ent; <?processing instruction>followed by
                                                text.
```

Figure 12. Interpretation of PCDATA

Figure 12 contains the element declaration of `a`, whose contents consists of parsable data characters followed by element `q`. The following are the three different interpretations of input handling.

(1) Only the text: `This is` is part of **PCDATA** because the entity reference `&ent;` terminates **PCDATA**. The entity reference resolves to the text `entity-text` which becomes part of element `q`. If the start-tag of `q` may not be omitted, or if data characters are not a valid content of `q`, then an error occurs.
(2) Only the text `This is entity-text` is part of **PCDATA**. The processing instruction terminates the **PCDATA**. The processing instruction is not part of the data but is situated between **PCDATA** and element `q` which means that `followed by text` forms the start of element `q`. If the start-tag of `q` may not be omitted, or if data characters are not valid as the content of `q`, then an error occurs.
(3) All the text after the start-tag of element `a`, including the processing instruction, is part of **PCDATA**. Element `q` is not started yet because **PCDATA** is not terminated. Only the start of element `q` terminates **PCDATA**.

The first interpretation is not easy to use, because then it is impossible to use entity references within data text. The second interpretation also has a disadvantage. When the user uses a processing instruction, **PCDATA** terminates and following data characters are not part of **PCDATA**. If this behaviour is not desired, the above element declaration of `a` has to be changed into Figure 13.

```
DTD:    <!element a     - - ( (#PCDATA)*, q) >
```

Figure 13. Solving the disadvantage of the second interpretation

A disadvantage of the element declaration in Figure 13 is that text which logically forms one piece is divided into two pieces, separated by a processing instruction. (In the above example, there are two blocks of text: `This is entity-text` and `followed by text.`). We have therefore chosen the third interpretation, in which all characters between two successive tags are part of **PCDATA**.

Complete document

Figure 7 shows that the output of the *document parser* is a complete document. The complete document differs from the input document in the following way:

- all short reference delimiters are expanded to the associated entity references;
- all entity references are expanded, including those introduced by a short reference delimiter;
- all use-map declarations in the document are parsed and are not part of the complete document. A use-map declaration in the DTD associates a short reference map with an element. In the document, a use-map declaration activates the named map, i.e., all short reference delimiters defined in this map become applicable. The previous map, if there was one, cannot be used any more in the element containing the use-map declaration;
- all entity declarations at the beginning of the document are parsed and are not part of the complete document. All entities declared in the DTD are applicable in the document. Because the DTD is the same for many documents, it is sometimes convenient to define additional entities in the document, for instance often used long phrases, to reduce typing and typing errors. These additional entities will differ from one document to another and can be defined at the beginning of the document and can be used throughout the document;
- all omitted end-tags are added;
- all omitted start-tags are added;
- all abbreviated tags are expanded;
- all spaces, tabs, new lines and comments between consecutive tags are removed. Comments, new lines etc. may be placed between consecutive tags. These comments, etc. are not part of the content of any of the elements. It only makes typing and reading the document easier. For example:

  DTD: `<!element a - - (b) >`
  document: `<a><!-- this is a comment --><b>`
  resolves to: `<a><b>`

- all *marked sections* are expanded. A marked section is a part of the document that has been marked for a special purpose, such as ignoring mark-up within it;
- all *attributes* with their values are set in the corresponding start-tag. Every element can have an attribute list attached. The attributes in this list give additional information about the element. **IMPLIED** attributes, (i.e., attributes whose value

the system will insert) will not be generated unless the user specifies an alternative value for them;
- all *character references* are expanded. A character reference consists of a number which denotes the decimal character code in the character set used. For example `&#97;` denotes the letter 'a' in the ASCII character set;
- all processing instructions, i.e., system-specific processing information, are kept. If necessary, each processing instruction is truncated to *PILEN* (240) characters.

Changing SGML constructs to other formats

As described in previous sections the output of an SGML parser is a 'complete' document. However, most people using SGML would not consider this to be the final stage and would normally print the document, or store it in a database, so that it could be processed further. With this in mind, we decided that the parser would not be complete without a back-end whereby the SGML document could be converted into some other coding scheme.

The back-end of the Amsterdam SGML Parser is simple, yet powerful enough to create typeset documents from SGML documents. The user can specify a mapping from each start-tag with its attributes to a replacement text, and a mapping from each end-tag to a replacement text. For example the mapping:

```
<title>    ".TL"
<head>     ".NH [level]"
```

denotes that the start-tag of the element `title` will be replaced by the string `.TL`, which is the *ms*-macro for a title. The start-tag of `head` will be replaced by the string `.NH` followed by the value of the attribute `level`. Of course `level` must be a valid attribute of the element `head`, otherwise an error message is given. The replacement text stands between double quotes and an attribute value is referred to by placing the attribute name between brackets (i.e., `[` and `]`). The *document parser* can be called with a user specified *replacement file*, which contains the mapping for the tags in the DTD. If a replacement file is specified, the tags in the output are replaced according to the mapping in the replacement file. Otherwise the 'complete' document will be output.

It is possible to specify that the replacement text must appear on a separate line by enclosing it between two `+` characters. This is needed, for instance, by *troff*, since each *troff* command must start with a `.` at the beginning of a line. Provisions are made to make it possible to put any character (including non-printable ones) into the replacement text. This is done by an escape mechanism similar to that of the C programming language. If, for example, the replacement file looks like:

```
<title>     +   ".TL"   +
<authors>   +   ".AU"   +
<affil>     +   ".AI"   +
<abstract>  +   ".AB"   +
</abstract> +   ".AE"   +
```

the document in Figure 14 will be converted to the *troff* document in Figure 2. Tags that are not mentioned in the replacement file are mapped to the empty string and they do not appear in the output.

```
<front><title>
The Implementation of the Amsterdam SGML Parser
<authors>Jos Warmer, Sylvia van Egmond
<affil> Faculteit Wiskunde en Informatica
Department of Mathematics and Computer Science
Vrije Universiteit Amsterdam
e-mail: jos@cs.vu.nl
<abstract>
The Standard Generalized Markup Language (SGML), is an
ISO Standard that specifies a language for document
representation. This paper gives a ...
</abstract></front>
```

Figure 14. SGML input document

Our experience has been that it is easy to convert an SGML document to *troff* or TEX, or some other similar type of formatter, to produce a typeset document on paper.

IMPLEMENTATION OF THE AMSTERDAM SGML PARSER

This section describes several implementation details, and is divided into subsections which discuss respectively, 'the lexical analyser', ambiguities, tag omission, the exception mechanism and the generation of LLgen code.

Recognition modes in the lexical analysis

In SGML there is a set of predefined delimiters that identify mark-up. Recognition of delimiters is done in several modes. These modes are called *recognition modes*. In each mode a specified subset of all delimiters is recognized. For example, inside a processing instruction only the delimiter **pic**, which denotes the end of the processing instruction, is recognized.

The Standard defines nine different modes and specifies which subset of delimiters is recognized for each mode.

This concept of modes is common in programming languages. There is always a comment-mode, in which only the end of the comment is recognized, and, often, a string-mode, in which only the end of the string is recognized. The main difference between modes in programming languages, and those in SGML, is that SGML makes the concept very explicit, providing many more modes than a programming language would do, and, furthermore, that modes in SGML can be nested.

However, there is one big problem with modes, namely that their definition in the Standard is incomplete. For example, there is no separate mode for a *comment*, so delimiters other than end-comment could potentially be recognized inside a *comment*. This problem is circumvented by a special note that goes with the definition of *comment*. It says:

> ''No mark-up is recognized in a *comment*, other than the **com** delimiter that terminates it''.

This note, of course, implicitly defines a comment-mode. The Standard suggests, while talking about the different recognition modes, that these modes will solve the problem of delimiter recognition, but in fact they do not solve this problem at all. Within the Standard there are several places where notes like the quotation above are used to identify special cases. If all these notes were incorporated in the general discussion about the recognition modes, the Standard would be far more readable and consistent.

Altogether, we need nineteen modes to incorporate all the notes about special cases, which is more than twice the number of modes that the Standard defines (nine).

Ambiguities in SGML

At the end of the first section it was mentioned that a DTD in SGML must be unambiguous. In section 11.2.4.3 of the Standard it says:

> ''A *content model* cannot be ambiguous; that is, an element or character string that occurs in the document instance must be able to satisfy only one primitive content token.''

There is also a reference to Annex H of the Standard and there it says:

> ''... by prohibiting models that are ambiguous or require 'look-ahead'; that is, a model group is restricted so that, in any given context, an element (or character string) in the document can correspond to one and only one primitive content token (see 11.2.4.3). In effect, the allowable set of regular expressions is reduced to those whose corresponding NFA can be traversed deterministically, ...''

These definitions can be compared to the definition of an LL(1) grammar in [9] on page 61:

> ''If the analyser can choose its target by simply looking at one character, the grammar is said to be LL(1).''

These definitions are very similar but in the next two subsections we give a little more detail about the differences between the LL(1) grammar conditions and the unambiguity requirement for SGML grammars.

Note that the LL(1) property is defined for Context Free (CF) grammars and that the SGML grammar (see later) is written in a slightly non-standard notation. However, this does not affect the checks that need to be performed on it.

It is beyond the scope of this paper to explain the LL(1) conditions in depth, and we shall assume that the reader is familiar with the idea of first- and follow-sets, as defined in the literature, to check whether a grammar is LL(1). Readers not familiar with these concepts are advised to read pages 209 – 215 of [8] to obtain the requisite background knowledge.

LL(1) grammars

We define three functions called MayBeEmpty, First and Follow for each non-terminal. The function MayBeEmpty is often called Empty or Nullable. Intuitively the meanings of the functions are as follows.

- MayBeEmpty(n) for any non-terminal is **true** if and only if n can produce empty.
- First(n) for any non-terminal n is the set of all tokens that can start n.
- Follow(n) for any non-terminal n is the set of all tokens that can follow n directly.

With these functions we can find out whether a grammar is LL(1). A grammar is LL(1) if the following two conditions hold for each non-terminal in the grammar:

(A1) For all productions in the grammar of the form $n_1|n_2|\cdots|n_k$: First(n_1) $\cap$ First(n_2) $\cap$ ... $\cap$ First(n_k) = $\varnothing$. That is, each alternative of a production rule starts with a unique set of tokens.
For example, the next rule does not conform to A1, because both alternatives start with token a.

```
<!element  a1   - - ( (a, b) | (a, c) )>

a ∈ First( (a, b) )
a ∈ First( (a, c) )

a ∈ First( (a, b) ) ∩ First( (a, c) ) ≠ ∅
```

When an a is encountered, it is impossible to decide which alternative should be chosen. Only when we look ahead to the next token, can we decide which alternative must be chosen, i.e. if the next token is a b then the first alternative is taken but if it is a c then we take the second. However, this *look-ahead* to the next input token is forbidden in LL(1) grammars.

(A2) For all non-terminals with MayBeEmpty(n) = **true** : First(n) $\cap$ Follow(n) = $\varnothing$. That is, any token which can start a non-terminal that could, potentially, be empty may not also be the start of anything directly following that token.
The a? in the following example may be empty.

```
<!element  a2   - - (a?, a, b) >

a ∈ First ( a? )
a ∈ Follow( a? )

a ∈ First ( a? ) ∩ Follow( a? ) ≠ ∅
```

When an a is encountered in the input, it is impossible to decide whether the first alternative, of a?, or the second, of a, should be chosen. Only when we look ahead to the next input token, we can decide which one to choose, i.e. if the next token is an a, then the a? alternative should be chosen, but if the next token is a b, then the a? should be taken as empty and the a option should be chosen. Once again, however, this look-ahead to the next input token (forbidden in LL(1) grammars) has had to be used in making our choice.

SGML grammars

To discuss the difference between LL(1) and unambiguity in SGML we need to understand the notions of *element token*, *content token* and *content model*. A *content*

model or *model group* is the right hand side of an element declaration (e.g. `(a?, a, b)` in the previous example). The occurrence of an element name in a *content model* is called a *primitive content token* or an *element token*. A *content token* is either an *element token* or a *model group*. A *model group* can be a sequence-group, an or-group or an and-group, depending on the type of connector in it (',', '|' and '&' respectively). This situation is shown in the rules below (which are slightly simplified versions of the actual SGML rules).

```
content_model            :  model_group
model_group              :   "(" content_token
                             [connector content_token]* ")"
                              occurrence_indicator?
content_token            :  primitive_content_token
                              | model_group
primitive_content_token  :  element_token
element_token            :  name occurrence_indicator?
connector                :   ","
                              | "|"
                              | "&"
occurrence_indicator     :   "?"
                              | "+"
                              | "*"
```

Differences between LL(1) and SGML grammars

The difference between the LL(1) property and the unambiguity requirement for SGML is that there is one construct which is ambiguous for LL(1), but unambiguous for SGML. Consider the element declaration:

```
<!element a   - - (b)* >
```

where `b` can produce empty.

This rule is ambiguous according to LL(1): if the input token is `b`, it is impossible to determine whether `b`, or the empty production for `b`, should be chosen first, which leads to more than one possible choice. In general a non-terminal n that can be empty (i.e MayBeEmpty(n) = **true**) may never have a '*' or '+' occurrence indicator. When such an occurrence indicator is present, the non-terminal can always be followed by itself, so First(n) $\subset$ Follow(n) and condition A2 is never satisfied.

However, in SGML, the above mentioned element declaration is not ambiguous. There is a disambiguating rule which specifies that empty productions don't matter as long as there is only one *element token* in the declaration that can match the input. In the above example, it does not matter whether the empty production for `b` is chosen first, because the matched `b` is always the same *element token*.

We have to look very carefully at the LL(1)-type ambiguities that may arise because of a '*' or '+' occurrence indicator, since some of them lead to ambiguities in SGML too. Take for example the element declaration:

```
<!element a   - - (b, c, b?)*>
```

This expression is not LL(1): an ambiguity is encountered when the input matches the `b?` after the `c`. If `b?` is chosen to be the empty rule, then the first `b` matches the input. These are different *element tokens* in the declaration so the disambiguating rule for SGML does not apply and the element declaration is ambiguous in SGML also.

Minimization of start-tags and end-tags

In the following two sections we discuss the minimization of start-tags and end-tags by discussing the circumstances under which they may be omitted.

Omission of start-tags

Intuitively a start-tag may be omitted if omission does not create an ambiguity when trying to map the document onto the DTD and if the start-tag is marked for omission (with an `O`) in the element's declaration. However, the precise rule is somewhat more complicated.

A content token is called *inherently optional* if it is followed by a '?' or a '*'. If a content token is followed by a '+', it is transformed to a sequence-group, (i.e., a sequence of content tokens separated by commas), consisting of the content token followed by the content token with a following '*'. For example, 'a+' is transformed into 'a, a*'. This transformation does not change the semantics and is convenient in implementing start-tag omission.

To consider omission of an element token's start-tag, the following conditions must hold:

- the start-tag must be marked `O` in the element's declaration;
- the element token must not be inherently optional;
- the element token must be part of a sequence-group. If the sequence-group itself is inherently optional then in order to omit the element token's start-tag, one of the content tokens preceding the element token in this sequence-group, must have occurred;
- the element token must not have a declared content of **CDATA**, **RCDATA** or **EMPTY**; and
- the omission of the element token's start-tag must not create an ambiguity.

These rules are illustrated in Figure 15.

```
<!element a               O -  (b, (c, d)?, e+, f*) >
<!element (b,c,d,e,f)     O -  (g) >
<!element g               O -  CDATA >
```

Figure 15. A DTD with start-tag omission turned on

In Figure 15 the start-tag of element tokens `b` and `d` and the start-tag of the first occurrence of element `e` may be omitted. The start-tag of element token `c` may not be omitted because the sequence-group is inherently optional and there is no content token

of this group preceding element c. The start-tag of element token g may not be omitted because element g is declared as **CDATA**.

DTD:

```
<!element a    O - (b?, c) >
<!element b    - - CDATA >
<!element c    O - (d) >
<!element d    O - (b?) >
```

doc: `<a><b>element b</b></a>` may denote either (1) or (2).

```
(1)       <a> <b>element b</b> <c><d></d></c> </a>
(2)      <a> <c><d><b>element b</b></d></c> </a>
```

Figure 16. Ambiguity with start-tag omission

In Figure 16, the start-tag of element token c and d may both be omitted according to all the conditions given above — except for the last of those conditions. To understand why an ambiguity would be created we note that, when the start-tag of element token b is encountered it is not clear whether this is the beginning of optional element token b, and that element token d is empty (case 1 in Figure 16) or that the optional element token b is skipped and b belongs to d (case 2 in Figure 16). The ambiguity is resolved if either of the start-tags for element tokens c or d is not omitted. The problem is to determine which start-tag may not be omitted.

We have implemented start-tag omission statically. Every element's content model is examined in the *dtd-parser*. Every start-tag may be omitted as long as the conditions mentioned before are satisfied, and as long as it does not create an ambiguity. The content model of element a in Figure 16 is examined before the content model of element c so the start-tag of the element token d is not considered for omission yet. Next the *dtd-parser* examines the content model of element c but now the omission of the start-tag of element token d is not allowed, because that would introduce an ambiguity. If the content model of element c were examined before the content model of element a, then the start-tag of element token d could have been omitted and not the start-tag of element token c.

Start-tag omission is very restrictive. In some cases start-tag omission is not allowed according to the Standard although it does not create a parsing conflict. Take for instance Figure 17.

```
<!element a      O - (b?, c) >
<!element b      O - (d) >
<!element (c,d)  O - CDATA >
```

Figure 17. Restrictive start-tag omission

The start-tag of element b may not be omitted, but omission does not create a parsing conflict. Either the parser encounters the start-tag of element d, in which case the start-tag of element b was omitted, or the parser encounters the start-tag of element c in which case element b was skipped, and no look-ahead is required.

End-tag omission

To consider the omission of an element's end-tag, the end-tag must be marked `O` in the element declaration. According to the Standard, a marked end-tag can be omitted only if the end-tag is followed by either:

(1) the end-tag of another open element or
(2) the start-tag of another element or by an SGML character, both of which are not allowed in the element's content model.

An element is an open element when the element's start-tag has occurred either explicitly, or implicitly through omission, and the element's end-tag has not occurred yet. The above definition makes clear that end-tag omission cannot be corrected during parsing of the DTD — it can only be corrected during parsing of the document. To retain information about the content model of the declared elements in the *document parser*, information is generated about the content models by the *dtd-parser*.

To implement end-tag omission, all omittable end-tags could be marked optional in the generated LLgen code, but doing so creates a lot of ambiguities within LLgen. To prevent these ambiguities, we have implemented end-tag omission by using the user-definable error routine of LLgen. We mark all the end-tags required in the LLgen code. Thereafter, any omission of an end-tag in the document causes an error and the error routine of LLgen is called.

To correct an error, LLgen first invokes the user-definable error routine. If this error routine succeeds in removing the error then LLgen resumes parsing, otherwise the standard LLgen error routine [15] takes over.

Our user-definable error routine first checks whether the token required by LLgen is an end-tag. If not, end-tag omission is not the cause of the error and the error routine of LLgen must correct the error. Otherwise, our error routine checks whether the wrong token (the one causing the error) is an end-tag of an open element so that (1) applies: if so the missing end-tag is inserted into the input and LLgen resumes parsing. If (1) is not applicable, our error routine checks whether the wrong token is an element's start-tag, or an SGML character that is not allowed in the content model of the most recent open element. In this case (2) applies, the missing end-tag is inserted and parsing is resumed, otherwise the user has omitted an end-tag in the document where that was not permitted.

Exclusions and inclusions

This section describes our way of handling exceptions. An exception is an exclusion or an inclusion. The exceptions are indicated in the DTD. The exception's scope is the element in which the exception is declared and all elements that are opened within this element.

Exclusions

An exclusion specifies that the excluded elements are not allowed in the content of the element in which they are declared, even though an excluded element may be part of the content model. For example:

DTD:

```
<!element a                - - (b, c) -(d) >
<!element b                - - (c | d)>
<!element (c,d)    - - CDATA >
```

documents:

```
(1)      <a> <b><d>content</d></b> <c>content</c> </a>
(2)      <a> <b><c>content</c></b> <c>content</c> </a>
```

The content model of element a consists of two elements, and element d is excluded from a according to the DTD. In SGML-terms, element d is then recursively excluded from all elements occurring within a. So, the first document in this example is not correct according to the DTD, since element d is excluded from the occurrence of b in a. The second document, however, **is** correct.

Exclusions are not difficult to implement. An excluded element that is part of a content model must always be an inherently optional element or a member of an or-group. Therefore the element is always optional. By maintaining a stack of excluded elements the parser can easily check whether an occurring element is excluded or not.

Inclusions

An inclusion is an element that may appear everywhere in the content of the element in which the inclusion is declared. Figure 18 presents a DTD with element d as an inclusion. Following the DTD are three documents, which are all correct according to the DTD.

DTD:

```
<!element a                - - (b, c) +(d) >
<!element b                - - (c | d)>
<!element (c,d)    - - CDATA >
```

documents:

```
(1) <a> <b><d>cont.</d></b> <c>cont.</c> </a>
(2) <a> <b><c>cont.</c></b> <d>cont.</d> <c>cont.</c> </a>
(3) <a> <b><c>cont.</c> <d>cont.</d></b> <c>cont.</c>
                                          <d>cont.</d> </a>
```

Figure 18. DTD and documents with inclusions

In the Standard, inclusions are defined in the following way. Let ‘Q’ be a generic identifier or group in a content model and ‘x’ the occurrence indicator, i.e., ‘x’ is ‘*’, ‘?’, ‘+’ or empty. If ‘R_1’ through ‘R_n’ are applicable inclusions, then a token Qx is treated as though it were :

$$(R_1 \mid \cdots \mid R_n)^*, (\mathrm{Q}, (R_1 \mid \cdots \mid R_n)^*)\mathrm{x}$$

This definition is not practical for an implementation. If it were used within LLgen, it would produce a large overhead and introduce many ambiguities. The Amsterdam SGML Parser uses the user-definable error routine of LLgen to handle an inclusion.

When a start-tag is encountered that does not fit in the production-rule being parsed, our error-routine is automatically called by LLgen. The error routine checks whether the start-tag is the start-tag of an applicable inclusion and if so the inclusion is parsed by recursively calling LLgen. To obtain a parser for the inclusion, we have to indicate in the LLgen-grammar that the included element is a possible start symbol. After the inclusion is parsed, the *document parser* resumes, with the rest of the input, in the state the parser was before the inclusion occurred. There is no restriction on the number of times an inclusion may appear.

Generation of LLgen-code

LLgen is a parser-generator. Its input is a grammar that consists of tokens (terminal symbols) and production-rules (non-terminals). Almost every element declaration can be translated directly into an LLgen production-rule. There is, however, one exception. To translate an and-group, an SGML construct, into LLgen, extra production rules are necessary because this construct is not known in LLgen: it must be simulated.

The and-group `a & b` means that elements `a` and `b` may occur in any order, i.e., `a` followed by `b` or `b` followed by `a`. A possible LLgen-rule for the content model `a & b` might be: `and-group : [ a b ] | [ b a ] ;`. Such a rule does not create an ambiguity when there are only two members. However, as the number of members becomes three or more, an ambiguity does appear. To get round this problem we generate several rules.

When the parser encounters the start-tag of `a`, it finds that this tag is allowed as the start of `and1` and so this rule is taken. After that the rule for `and2_3` will be entered. Every alternative, except the last one in the or-group, is preceded by a so-called *conflict*

```
DTD:
<!and-group     - -  (a & b & c) >

ambiguous:
and-group :   [a b c] | [a c b] | [b a c] | [b c a] |
              [c a b] | [c b a];

unambiguous:
and-group : [%if (token is start of and1) [ and1 and2_3] |
              [%if (token is start of and2) [and2 and1_3] |
              [and3 and1_2]]
            ];
and1_2    : [%if (token is start of and1) [and1 and2] |
            [and2 and1]];
and1_3    : [%if (token is start of and1) [and1 and3] |
            [and3 and1]];
and2_3    : [%if (token is start of and2) [and2 and3] |
            [and3 and2]];
and1      : starttag_a a ;
and2      : starttag_b b ;
and3      : starttag_c c ;
```

Figure 19. Generated LLgen code for an and-group

resolver, which ensures that an alternative is chosen only if its start-tag is actually encountered. This is needed because several of the alternatives may produce empty.

SUMMARY AND CONCLUSIONS

In this paper we have given an introduction to SGML and described the design and implementation of the Amsterdam SGML Parser. This takes, as its input, a document type definition and generates a *document parser*. The *document parser* recognizes documents constructed according to the DTD and adds missing tags, expands short references and so on.

The parser is written in the programming language C and uses only standard C constructs [11]. It was developed under SUN/Unix 4.2 BSD and also runs under VAX/Unix 4.1 BSD and VAX/VMS system 4.5. The source code is 333 Kb with an extra 141 Kb source code for LLgen. The object code for the *dtd-parser* is 131 Kb on a SUN with a 2 Mb run-time overhead (for the AAP BK-1, which is a rather large DTD) mainly used for ambiguity checking and start-tag omission. The object code of the *document parser* for the AAP BK-1 [13] amounts to 352 Kb on a SUN, with no significant run-time overhead. At present the parser is used by the Dutch publishing company Elsevier Science Publishers and will shortly be used by another Dutch publisher, Wolters Kluwer.

To parse the DTD's and documents, we have chosen to use an existing parser generator instead of building one ourselves. The advantage of building one ourselves would be that the parser could be specialized for parsing SGML documents, and could therefore be optimized. The drawback is that adding good error-recovery is extremely difficult and a lot of work would have to be expended in this area. Therefore we decided to use LLgen, which has a very good error-recovery facility. Using LLgen for parsing DTDs was straightforward. It was, however, more work than we expected to convert the SGML content model to a grammar suitable for LLgen, but still the advantages of using LLgen outweighed the disadvantages.

While implementing the Standard, we found that it contained a lot of special cases, ambiguities and errors, some of which have been discussed in previous sections. The special cases, mostly expressed in notes, make the Standard difficult to understand. For almost every rule there is at least one exception, and owing to the lack of an index, the Standard is not easy to use as a reference manual. A separate index has been published [16], and there is a clear need for it. The annexes contain explanations of the Standard, but they were not very helpful. We found that only the simple and straightforward concepts are explained; the more difficult concepts remain vague. The annexes also contain information on topics (e.g. ID-attributes) which are not discussed in the Standard itself.

After finishing the parser we decided to use it ourselves to discover the pros and cons of SGML and our parser. We wrote the technical documentation for our parser, as well as the draft version of this paper, using SGML. For the documentation we wrote a simple DTD but for the draft paper we wrote a DTD that formalizes the *troff ms* package. As a result of our experiences we have reached the following conclusions about SGML.

- The complete checks on ambiguity of the content model are absolutely necessary when one is writing a document type definition. That is, the MODEL validation service of SGML is needed.

- Also necessary is the GENERAL validation service. This means that SGML errors and/or mark-up errors are reported. Without these one easily creates incorrect documents and/or DTD's.
- The other validation services are not needed very often and one can easily do without them. At least we did not miss them.
- Tag omission and short references are an absolute must, if you write a document in SGML. Otherwise specifying mark-up in the document becomes very tedious.
- Converting an SGML document into *troff* or TEX is very easy, using the back-end provided.
- Having a formal document type definition, against which your document is checked, is very helpful. The error reports from the parser make it easier to create correct output because mark-up errors are found and reported. Strange looking output, caused by errors in the mark-up, is less frequent than when using bare *troff* macros. In fact, using SGML can save a lot of time in this case.

ACKNOWLEDGEMENTS

We would like to thank Frans Heeman for volunteering to test the parser. Because this involved reading and understanding the Standard, it was not an easy job. He never stopped bothering us with questions such as 'is this a bug or a feature?'. This helped in our understanding of the Standard and made the parser into a stable product. We would also like to thank Hans van Vliet for his numerous remarks while we were writing this paper, Greg Sharp for correcting the English and Ceriel Jacobs for his help with LLgen.

REFERENCES

1. D. E. Knuth, *TEX and METAFONT: New Directions in Typesetting*, Digital Press, Bedford, 1979.
2. J. F. Ossanna, 'NROFF/TROFF User's Manual', *UNIX Programmers Manual*, **2A**, 203–263 (1979).
3. M. E. Lesk, *Typing Documents on UNIX and GCOS: The -ms Macros for Troff*, 1977.
4. ISO, *Information processing — Text and office systems — Standard Generalized Markup Language (SGML)*, First edition 1986-10-15, . Ref. No. ISO 8879-1986 (E)
5. ISO, *Information Processing — Text and office systems — SGML Amendment 1 (Final Text with Ballot Commends Resolved)*, Ref. No. ISO 8879-1986 (E) Amendment 1
6. D. W. Barron, 'Why use SGML?', *Electronic Publishing — Origination, Dissemination and Design*, **2** (1), 3–24 (1988).
7. Martin Bryan, *SGML: An Author's Guide to the Standard Generalized Markup Language*, Addison-Wesley, Wokingham, 1988.
8. J. Lewi, K. De Vlaminck, J. Huens, and M. Huybrechts, 'The ELL(1) Parser Generator and the Error Recovery Mechanism', *Acta Informatica* (10), 209–228 (1978).
9. M. Griffiths, 'LL(1) Grammars and Analysers', in *Compiler Construction, An Advanced Course*, ed. J. Eickel, Springer-Verlag, New York, (1974).
10. C. J. H. Jacobs and D. Grune, 'A Programmer — friendly LL(1) Parser Generator', *Software — Practice and Experience*, **18** (1), 29–38 (1988).
11. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, New Jersey, 1978.
12. A. S. Tanenbaum, H. van Staveren, E. G. Keizer, and J. W. Stevenson, 'A Practical Toolkit for making Portable Compilers', *Communications of the ACM*, **26** (9), 654–660 (1983).
13. Association of American Publishers, *Standard for Electronic Manuscript Preparation and Markup*, February 21, 1986.

14. Association of American Publishers, *Reference Manual on Electronic Manuscript Preparation and Markup*, May 1986.
15. J. Röhrich, 'Methods for the Automatic Construction of Error Correcting Parsers', *Acta Informatica*, **13**, 115–139 (1980).
16. R. Stutely and J. Smith, *SGML: The User's Guide to ISO 8879*, Ellis Horwood Ltd, Chichester, 1988.