This document is marked up using the future International Standard ISO 15445 ISO-HTML (work in progress). It may be validated by the reader's browser as syntactically correct, thus ensuring the reader that the author's work is faithfully reproduced. Questions?

No more Me Too - Different approaches to logic documents

Roger Price
April 13th 1997. Major revision May 22nd, 1997

Department of Computer Science,
University of Massachusetts Lowell,
One University Avenue,
Lowell MA 01854 USA.
Email: `rprice@cs.uml.edu`

Abstract

The aim of this paper is to consider deeper relationships between LP and the underlying SGML on which web documents are based. The paper identifies four approaches and discusses three of them. The first has been investigated by Loke and Davison and considers the document as a fact. The second considers LP rules and facts to be a part of the document, and requires SGML engineering provided by a standardisation of the HTML language: ISO-HTML. The third approach considers that the elements and attributes that make up the page are facts. The fourth approach considers that the meta language is a logic program and calls for a new document description language based on LP to replace the current SGML.

Keywords: Logic programming, Web, SGML, Prolog, HTML, documents.

Introduction

Foreword

Contributions by logic programming (LP) to the World Wide Web are often considered in terms of the existing document structure and the existing agent architectures. This leads to a me-too approach which places logic programmers at a disadvantage. The paper argues against this approach. A more fruitful avenue would be to consider the relationship between logic programming and the underlying Standard Generalised Markup Language (SGML) on which web documents expressed in the HyperText Markup Language (HTML) are based.

This introduction takes Loke and Davison's LogicWeb as the first approach, and then introduces three further approaches, each time increasing the role of LP in documents, and ending with a proposal for an LP replacement for SGML.

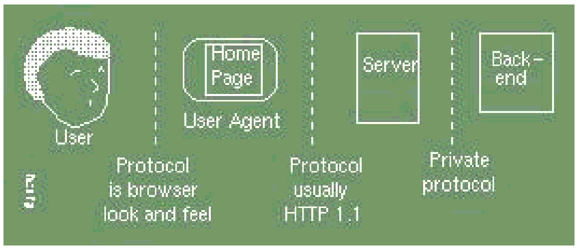
Throughout this paper the term SGML document means a marked up document based on some grammar or document type definition (DTD) specified using the SGML language. The grammar is not specified. Where we refer to an instance of a specific grammar, we shall say for example HTML document.

Web documents

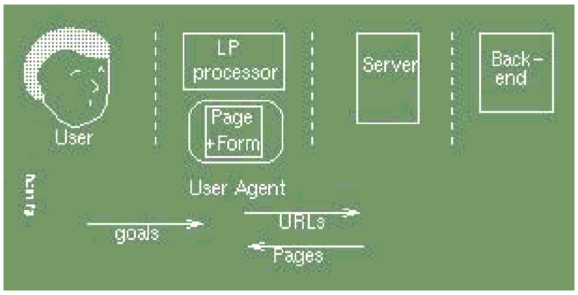
The documents that are interchanged on the World Wide Web, and whose links define the web are expressed using one or other of the dialects of the HTML language. The IETF have developed RFCs HTML 2.0, Form-based file upload, HTML Tables and HTML Internationalization, and the World Wide Web Consortium (W3C) have published a Recommendation for HTML 3.2 in an effort to document current practice. The W3C are continuing to develop the language with the Cougar Project.

The general level of conformity to these specifications is low. Documents structured using the HTML language have little intrinsic behaviour other than to be browsable. Commercial products such as ActiveX and Java have sought to add additional behaviour using mobile objects programmed in imperative languages. The main objective of these mobile objects is often to enhance the human computer interface, but these techniques also allow the intrusion of hostile objects.

In this paper we will be examining other approaches. To help the reader compare the different approaches being considered, we use this simple reference diagram:



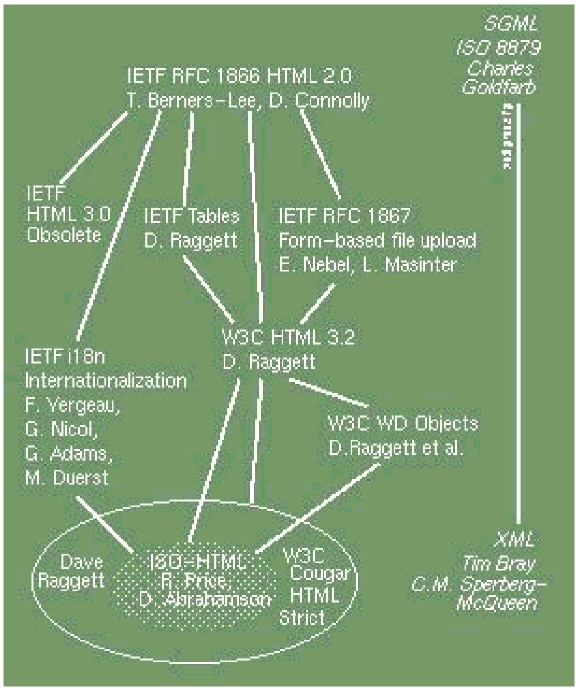
Logic programming techniques have been used to enhance the basic browser/server model, for example an attractive extension using LP has been introduced by Loke and Davison with their LogicWeb development of the user agent:



The LogicWeb approach views the HTML document instance as a single object with state. The logic program treats the URL of a page as a fact, and encapsulates the entire HTML document instance as another fact. It adds a forms interface to the page and allows the user to enter goals in a language close to Prolog. Further LP goals may be added as *live clauses* and complex relationships may be created requiring further HTML document instances fetched from the servers.

LogicWeb has the advantage that by encapsulating the HTML document instance as a fact it is isolated from any problems associated with the HTML. We classify this as the *first* approach, and we shall seek a closer integration since we believe that logic programmers should attempt to introduce a logical structure at a more fundamental level. This means that we are not immune to HTML language problems: we have to face up to the current instability of the language and the generally poor conformance.

As a first step to furthering a *second* approach, and in an effort to stabilize the HTML language, the author has proposed a new work item in the ISO subcommittee responsible for the SGML language family: ISO-HTML. The proposal is currently being balloted as a Final Committee Draft. *The W3C and the ISO are currently planning to harmonize their work so that ISO text will form a hard core of the W3C specification.* The language, besides many design improvements, introduces the extension technique required for integrating logic programming. The pedigree of the language is shown in the following figure:



We introduce the formal basis for the second approach to the integration of logic programs in documents. Further information is available via Robin Cover’s bibliography.

Once the SGML techniques are available, we use them to describe the interfaces which are needed in the second approach. The very trivial agent described in this section is not intended as an example of a real agent --- it serves only to illustrate the techniques used to include it in documents.

The *third* approach to the introduction of LP techniques into the web will be to look at the general problem of document description from an LP point of view. This goes deeper than the HTML language to the underlying SGML language and introduces a *logic document* without attempting to position it with respect to the considerable existing work on data bases and knowledge bases. Whether a logic document is data or knowledge is left unanswered -- it will probably be more fruitful to explore the concept without the preconceptions that are introduced by the use of the terms data and knowledge.

We introduce the third approach and describe the integration of documents and logic programs. We provide a simple example of this integration.

The *fourth* approach is for the long term: The replacement of the underlying SGML language with a simpler declarative language for specifying document architectures and the associated semantics.

We describe the fourth approach as part of the discussion of possible future directions.

The four approaches may be summarized as follows:

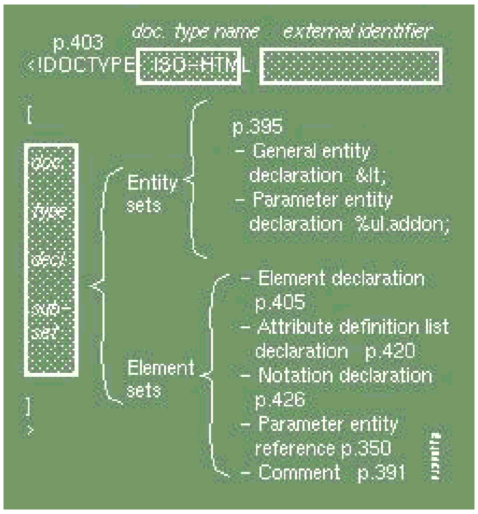
1. The page is a fact.
2. The rules and facts are part of the page.
3. The elements and attributes are facts.
4. The meta language is a logic program.

The SGML basis for HTML

We first review the features of the Standard Generalised Markup Language SGML, ISO 8879:1986, which form the basis for HTML. SGML is complex and the International Standard is generally accepted to be unreadable: the preferred alternative is the SGML Handbook by Charles Goldfarb and in this section we indicate the relevant pages.

An SGML document entity has three parts [p.295]

1. The SGML declaration which says for example which character sets and language features are to be used.
2. The prolog which provides [p.303]:
    1. Application specific rules for the parser.
    2. The *document type declaration* [p.403]. This plays a key role in the proposed second approach to web logic programming. The declaration identifies the document grammar to be used and provides in the document type declaration subset, an opportunity to extend this grammar. The extension techniques we require will be described later. The document type declaration is summarised in the following figure:



3. The document instance set which in the case of ISO-HTML provides the familiar markup:

```
<iso-html>
<head>...</head>
<body>...
  <h1>...
</body>
</iso-html>
```

Declaring elements and attributes

Before entering into the details of the extension interfaces, we first review the manner in which elements and their attributes are defined by SGML. An element declaration typically takes the form:

```
<!ELEMENT ISO-HTML - - (HEAD,BODY) >
```

This should be read as The element ISO-HTML is defined as containing a `<HEAD>` element followed by a `<BODY>` element. The `- -` says that it is an error to omit either the start tag `<ISO-HTML>` or the end tag `</ISO-HTML>`.

Another example is

```
<!ELEMENT P - O (#PCDATA|A|BR|IMG|MAP) >
```

This says The element P may contain character data which is to be parsed, or the elements A, BR, IMG or MAP. The `- O` says that the end-tag `</P>` may be omitted.

An attribute definition list declaration typically takes the form

```
<!ATTLIST P
          DIR  (ltr|rtl) #IMPLIED
          ID   ID        #IMPLIED
          LANG NAME      #IMPLIED >
```

This should be read as The element P has three attributes. The first is named `DIR` and takes one of the values `ltr` or `rtl`. The second is named `ID` and has type ID, i.e. its value is a unique identifier. The third is named `LANG` and is a name which need not be unique. The default behaviour of all three if no value is provided by the document author is *implied* by the application. These attributes specify the writing direction of the text, provide a name for the paragraph and specify the language used.

The definitions given above are sufficient to allow an author to write

```
<P ID=debut LANG=fr>
La nuit fut sombre et l'orage...
```

SGML also provides a macro mechanism. The macros are known as *entities*. Here is an example of an entity designed to save a lot of typing at election time:

```
<!ENTITY promise
  'If elected I promise you that I will
   reduce taxes.  I also promise you' >
```

Such an entity is called as follows

```
&promise; complete social security.
```

If the entity is used in the definition of a language element, the `&` becomes a `%`.

Document type declaration subset

In many SGML-based document architectures, and in all HTML where the formal basis is often completely absent, the document type declaration subset, i.e. the part of the DOCTYPE declaration between the square brackets, is absent. However its use is explicitly called for by ISO-HTML and this provides the interfaces needed for our second approach to web logic programming:

1. The **extend** interface provides a socket into which one may plug declarations of new elements.
2. The **addon** interface provides a socket into which one may plug additional attribute declarations.
3. The **notation** declaration interface is provided by the SGML language as a means for identifying the nature of data which is not parsable by an SGML parser.

For those systems that are unable to handle the subset, including many Web browsers, it is always possible to repackage the document type declaration in such a way that no subset need appear in a document instance. This approach is preferable in extensions that become the subject of further industry, regional or national standards.

The extend interface

ISO-HTML defines content models for the major elements `<HEAD>` and `<BODY>` in such a way that the models may be extended by an application architect. In the case of the `<BODY>` element, the base definition is:

```
<!ENTITY % extend 'NOP' >
<!ELEMENT BODY - -
   ((#PCDATA | A | BR | IMG | MAP | DL | FORM
          | HR | PRE | TABLE | %extend;)* ) >
```

These may be read as saying that the body element may contain parsed character data and any of the elements `<A>` through `<TABLE>`. In addition any elements defined by the `extend` entity may appear. The default value of `extend` is the `<NOP>` element which has no semantics. If a new definition for the `extend` appears in the subset, it will override the NOP default definition.

The addon interface

All the elements of the ISO-HTML language have an attribute definition list which terminates with a parameter entity reference of the form `X.addon` where `<X>` is the element. The full definition of the `<P>` element of ISO-HTML is thus:

```
<!ENTITY % P.addon "" >
<!ATTLIST P
          DIR  (ltr|rtl) #IMPLIED
          ID   ID        #IMPLIED
          LANG NAME      #IMPLIED
          %P.addon; >
```

The addition reads Element P also has the attributes defined by the parameter `P.addon`. The default value of the addon is an empty string. If an author specifies a new attribute value in the entity `P.addon` it becomes an attribute of the `<P>` element. It is not an error in SGML to have multiple definitions for an entity (macro). The first definition the parser meets applies.

The notation declaration interface

An SGML *notation* is any data which cannot be understood by the SGML parser. (The name refers to the `<!NOTATION...>` declaration which provides a type for the data.) The data is significant to applications or humans. e.g. TeX, MIDI, JPEG or G.721.

If the notation uses the character set of the document, it may be placed in the text flow of the document as shown in the following example:

```
<!DOCTYPE...
[<!NOTATION Prolog PUBLIC
    "ISO 13211-1:1995//NOTATION Prolog//EN">
]>
```

This says that we are extending the definition of the type of the document to include a type called Prolog which is formally defined by ISO 13211. We may now refer to Prolog as a data type. Let us define an element to hold instances of this data type:

```
<!ELEMENT RULES - - CDATA>
<!ATTLIST RULES
          TYPE  NOTATION (Prolog) #REQUIRED>
```

This reads: The `<RULES>` element contains a sequence of characters terminated by `</RULES>`. The element has the attribute `TYPE` which specifies a notation. It is required and must be specified as Prolog. This tells us that the sequence of characters can be understood if we read them as Prolog.

We are now able to write the (trivial) example

```
<RULES NOTATION=Prolog>
comfortable(X):-
   not_too_low(X), not_too_high(X) .
```

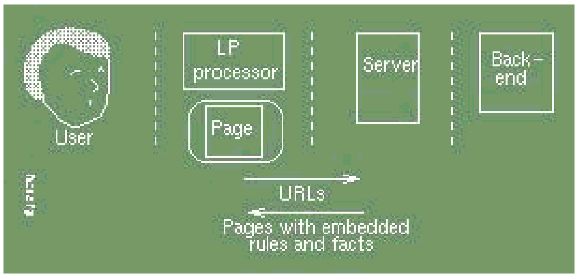
```
</RULES>
```

If the notation does not use the same character set as the text flow, it may be placed in an external file. The details will not be addressed in this paper.

These SGML techniques are hardly ever used in the Web, but they are used in more sophisticated SGML applications. We will now use them to illustrate our proposed second approach to logic programming on the Web.

Second approach: Documents contain rules

We shall use a trivial agent to assure us that the temperature selected by the user is acceptably comfortable. The agent is placed in the page requested from the server. (We have deliberately chosen a trivial agent: the aim of this paper is not to investigate any particular class of agents, but to show the binding mechanisms required.)



The approach requires the use of the future international standard 15445 ISO-HTML since it builds on the general SGML techniques which are provided by that specification.

First we have to define the agent:

```
<!ELEMENT AGENT - - (BIND*, RULES) >
<!ATTLIST AGENT
          ALT        CDATA    #IMPLIED
          NAME       CDATA    #IMPLIED >
```

The `<AGENT>` element contains zero or more instances of `<BIND>` followed by exactly one instance of `<RULES>`. The attributes provide:

- `ALT` Text to be displayed if the agent cannot be activated.
- `NAME` Used for name/value pairs in a form data set.

We now define the binding mechanism:

```
<!ELEMENT BIND         - O  EMPTY >
<!ATTLIST BIND
          SGML       IDREF    #REQUIRED
          ATTRIBUTE  NAME     #IMPLIED
          CONTENT    (content) #IMPLIED
          RULE       CDATA    #REQUIRED
          VAR        CDATA    #IMPLIED >
```

The element `<BIND>` has no content and its end tag is always omitted. Its attributes provide the binding between an attribute of an element in the source document and a variable in the LP rules:

- **SGML** A reference to the unique identifier of the element having the attribute we wish to bind. ISO-HTML allows for `ID` attributes to be placed on any element.
- **ATTRIBUTE** The name of the attribute to which we wish to bind.
- **CONTENT** A flag saying that we wish to bind to the content of the element specified by the `SGML` attribute.
- **RULE** The functor of the head of the clause to which we wish to bind.
- **VAR** The variable in the clause identified by the `RULE` attribute to which we wish to bind.

The rules require a `<RULES>` element as follows:

```
<!ELEMENT RULES  - O  CDATA >
<!ATTLIST RULES
          TYPE NOTATION(Prolog) #IMPLIED
          GOAL CDATA            #REQUIRED >
```

- **NOTATION** The data type of the rules as defined by the notation declaration. In our case this will be Prolog.
- **GOAL** The goal which is to be satisfied by the rules.

To complete these definitions, we need to define the notation and introduce the `<AGENT>` element into the ISO-HTML language:

```
<!DOCTYPE...
[<!NOTATION Prolog PUBLIC
    "ISO 13211-1:1995//NOTATION Prolog//EN">
]>
<!ENTITY % extend "AGENT" >
```

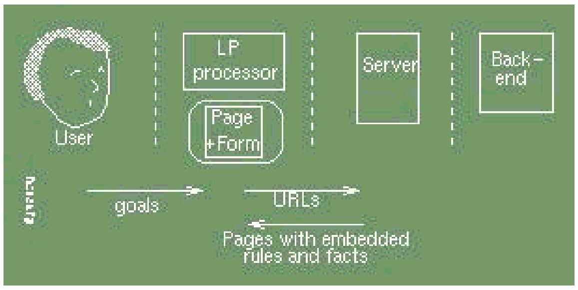
We are now in a position to markup an instance of an document with an agent as follows:

```
<h2>Form with agent to check temperature</h2>
<FORM ACTION="protocol://host.domain/server">
   Please type temperature of your choice:
   <INPUT TYPE=text ID=v1 NAME=choice
                       VALUE="21">
   When you are ready to submit your choice,
   <INPUT TYPE=submit VALUE="select here">.
   <INPUT TYPE=reset
          VALUE="Select here to start over.">
   <AGENT ALT="Agent not available."
          NAME=temp>
     <BIND SGML="v1" ATTRIBUTE=value
           RULE="comfortable" VAR="X">
     <RULES Prolog GOAL="comfortable(X)">
       comfortable(X) :-
          X>17, X<29,
          write('Temperature ok.'), nl .
       comfortable(X) :-
          write('Error: '), write(X),
          write(' is not in range 18-28.'),
          nl, fail .
   </AGENT>
</FORM>
```

The behaviour of this form is:

1. The value 21 of the `VALUE` attribute of the `<INPUT>` element with `ID` value `v1` is bound to the variable `X` of goal `comfortable(X)`.
2. The form appears to the user in the look and feel of user agent.
3. When the user selects the button or other device which represents the `<INPUT>` element with `TYPE=submit`, the user agent attempts to satisfy the goal `comfortable(X)` specified by the `GOAL` attribute of the `<RULES>` element. If the goal succeeds, the user agent returns the name/value pair `temp/X` as part of the form data set sent to the server. Otherwise an error message is displayed and the user invited to make another choice.

This second approach has the advantage that it allows a finer grained integration of the logic programming with the document instance, but the engineering is complex. It is orthogonal to the first approach and the two could be used together:

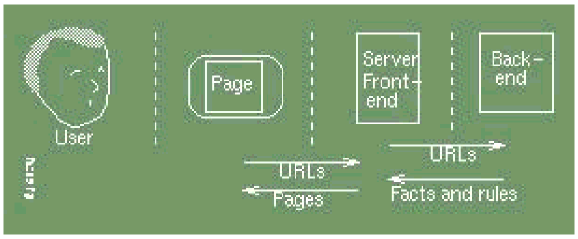


It is not reasonable to expect a typical web author to work with anything as complex as this, and any deployment would be based on authoring systems. Such an authoring system could be simplified if it were not required to manipulate two languages. We investigate this further in the third approach.

Third approach: The elements and attributes are facts

The traditional approach to extending the features of an HTML document is either to add proprietary elements and attributes, or to add executables to the stream carried by the HTTP protocol and have them executed or interpreted by an engine added to the browser.

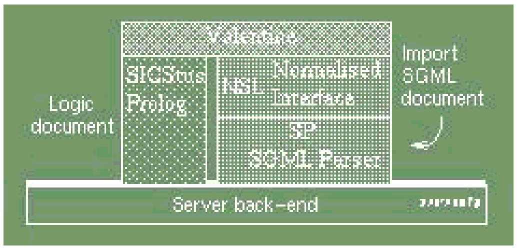
Our third approach proposes that rather than stick on executables, the entire document together with any extensions should be considered as a single integrated logic program.



The server is conceptually split into two parts: the front end continues to offer HTML pages in response to URL requests, but obtains them from a parsed form represented by a logic program rather than by searching in a database or executing a programmed sequence of write commands to generate the page.

To investigate this third approach, a simple demonstration of a server back-end Valentine is being prepared.

Valentine assumes that the documents are created as SGML markup and consists of a bridge between the SP SGML parser augmented with the Normalised SGML Library (NSL), on one side and the SICStus Prolog system on the other. The prototype runs on the Linux operating system. An overview of the architecture is shown below:



Valentine's process is as follows:

1. Import and parse the source document using the excellent SP parser. This produces a stream of events as the document is parsed.
2. These events are normalised using the Normalised SGML Library developed by the Language Technology Group of the University of Edinburgh to form a sequence of bits of the document instance.
3. The bits are expressed as fully ground unit clauses and asserted in the Prolog data base.
4. The document and its embedded logic are considered to be a single logic program.

This process works for any document type, but we will limit ourselves to ISO-HTML. Given the following tiny ISO-HTML document:

```
<iso-html>
<head><title>Andre' Gide</title></head>
<body>
<H1>Dogs bark</H1>
<p>but the caravan rolls on.
</body>
</iso-html>
```

we obtain the corresponding *logic document* (the attribute facts have been omitted):

```
tuple(0, start,
         'ISO-HTML', point(-1,[1,2,3,4,5,6,17])).
tuple(6, start,
         'HEAD', point(0,[7,8,9,10,11])).
tuple(11, start,
         'TITLE', point(6,[12,13,14,15,16])).
tuple(16, text,
         'Andre\' Gide', 11).
tuple(17, start,
         'BODY', point(0,[18,19,20,21,22,29])).
tuple(22, start,
         'H1', point(17,[23,24,25,26,27,28])).
tuple(28, text,
```

```
         'Dogs bark', 22).
tuple(29, start,
         'B1', point(17,[30])).
tuple(30, start,
         'P', point(29,[31,32,33,34,35,36])).
tuple(36, text,
         'but the caravan rolls on.', 30).
```

Logic documents may be exported as SGML document instances. Note that Goldfarb's First Law of Text Processing applies to the export of SGML. The export process does not decorate the markup declarations it issues with RSs (Record Starts) or REs (Record Ends). However a conforming SGML parser such as SP will remove REs which immediately precede markup since the RE is considered to be due to the markup, not the source text. As a result, importing SGML file *F* and then exporting it to file *G* will not necessarily produce two identical files. Insignificant REs in *F* will not appear in *G*. There are other reasons for differences, but this is sure to be one of them.

Logic documents in raw form

The *logic document* view abstracts the internal details of the parsing and presents a complete declarative image of the document. Let's be modest: only the document instance is represented, without the document type declaration and the important document type declaration subset. A logic document with no additional rules is very close to the parse tree.

In the demonstration, a document is represented in the Prolog database as a set of tuple/4 facts asserted in the user's space (module user). These have the typical form

```
tuple(Serial,Type,Value,Point) .
```

The arguments are as follows:

1. `Serial`: A non-negative integer which distinguishes the components of a given document. Although called serial numbers, the order is not significant. The integer value assigned to each fact in a document is unique.
2. `Type`: one of {`START`, `ATTRIB`, `EMPTY`, `TEXT`, `PI`}.
   - The value `START` indicates a fact describing the start of an element. A `START` fact points back to its parent `START` fact and to each child fact within its scope.
   - The value `ATTRIB` indicates a fact describing an attribute. The fact points back to the corresponding `START` or `EMPTY` fact and provides the attributes name, declared value or a flag if no value was provided, the declared value type and the default behaviour.
   - The value `EMPTY` indicates a fact describing an element with no content. The `EMPTY` fact points back to the parent `START` fact and to each of its `ATTRIB` facts.
   - The value `TEXT` indicates a fact which contains text. It provides a pointer back to the parent `START` fact.
   - The value `PI` indicates a fact which contains a processing instruction. It also provides a pointer back to the parent `START` fact.
3. `Value`: The possible values are given in the following table. All the values are expressed as atoms. (This is a weakness - a Prolog processor cannot be assumed to be able to handle atoms of arbitrary size efficiently.)

Type	Value
`START`	Element name, e.g. 'H1'
`ATTRIB`	Attribute description, see below.
`EMPTY`	Element name, e.g. 'IMG'
`TEXT`	E.g. 'It was a dark and stormy night'
`PI`	The processing instruction.

The attribute values are described by the attrib/4 structure: (The presence of `Decl` and `Default` is undesirable. When the model is able to handle the constraints on the logic document correctly, these will be removed.)

```
attrib(Name,Value,Decl,Default)
```

where

- `Name` is the name of the attribute, e.g. ID
- `Value` is the declared value or `[]` if there was none.
- `Decl` is one of the following declared values: `CDATA`, `ENTITY`, `ENTITIES`, `ID`, `IDREF`, `IDREFS`, `NAME`, `NAMES`, `NMTOKEN`, `NMTOKENS`, `NOTATION`, `NUMBER`, `NUMBERS`, `NUTOKEN`, `NUTOKENS` or `nameTokenGroup`.
- `Default` is one of the following default values: `FIXED(Value)`, `REQUIRED`, `CURRENT`, `CONREF`, `IMPLIED` or `specified(Value)`.

4. `Point`:
   - **Case a:** If the type is `START` or `EMPTY` this argument provides pointers to the parent fact and to the content of the `START` fact. The argument has the form

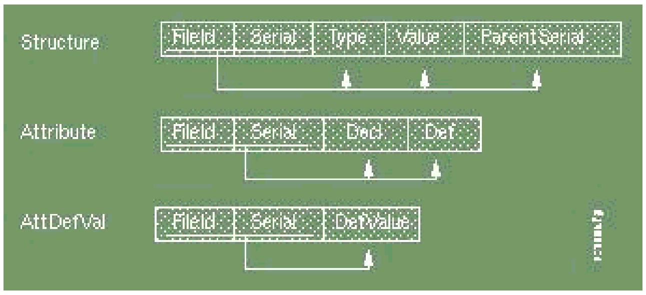
     ```
     point(Parent,ListOfSiblings)
     ```

     The `Parent` is the integer serial number of the parent fact, or -1 if the fact is the document fact. The `ListOfSiblings` is a possibly empty list of the facts contained in this fact. The `ListOfSiblings` also gives their order of occurance.
   - **Case b:** If the type is `ATTRIB`, `TEXT` or `PI` this argument provides the serial number of the parent fact.

Logic documents in Boyce-Codd Normal Form

The form of document we described is suitable for direct processing by Prolog programs, but is not in a suitable form for permanent storage as a logic document. We show a set of schema more suitable for use in general purpose relational database systems.

The attributes of the tuples `tuple(FileId,Serial,Type,Value,Point)` are not atomic. For example, when the tuples represent attribute values, `Value` is a term and not an atom. Similarly for the `START` and `EMPTY` tuples, the value of `Point` is a term containing a list. The result of this complexity is that the document is not even in first normal form, 1NF. However a restatement of the tuple structure as shown below satisfies the criteria for Boyce-Codd normal form (BCNF): A relational schema *R* is in BCNF, if whenever *X->A* holds in R, either *X->A* is trivial, or *X* is a superkey of *R*. Since there are no multivalued dependencies, the document is also in 4NF.



The primary keys are underscored and the functional dependencies indicated by arrows.

We have not forgotten the constraints imposed on the document by the original document type definition (DTD). Work is currently underway to investigate this. The mapping of the DTD to a set of LP constraints requires parsing of a significant part of the SGML meta-language.

An example

We now discuss techniques for programming agents in logic documents. Again, we insist that this trivial agent is intended only to show the logic document techniques; it is not intended as a real agent.

Documents may be created by logic programs or by importing SGML documents, which are read piece by piece through the SP parser and asserted as logic documents in the Prolog database.

The agent is programmed using Prolog, and acts directly on the logic document. This simplifies the binding compared to the second approach, but makes the agent sensitive to changes in the logic document representation.

We now show an example of this approach. Consider the following ISO-HTML page based on the reply to the SQL query Q1 given by Elmasri and Navathe in Fundamentals of Database Systems p.193.

```
<ISO-HTML>
<HEAD>
<TITLE>Research Department Staffing</TITLE>
</HEAD>
<BODY>
<P>
The following people work in the
research department: Franklin Wong,
John Smith, Joyce English and Ramesh
Narayan.
</BODY>
</ISO-HTML>
```

The corresponding raw logic document is:

```
tuple(0, start, 'ISO-HTML',
         point(-1,[1,2,3,4,5,6,17])).
tuple(6, start, 'HEAD',
         point(0,[7,8,9,10,11])).
tuple(11, start, 'TITLE',
         point(6,[12,13,14,15,16])).
tuple(16, text,
```

```
         'Research Department Staffing', 11).
tuple(17, start, 'BODY',
         point(0,[18,19,20,21,22])).
tuple(22, start, 'P',
         point(17,[23,24,25,26,27,28])).
tuple(28, text,
         'The following people work in the \n
          research department: Franklin Wong,\n
          John Smith, Joyce English and Ramesh \n
          Narayan.', 22).
```

This is fine as long as the personnel do not change, but in these days of research budget compression and corporate downsizing, it would be better if the document itself could check on the current staffing. Clearly this could be done with a periodic batch job or a query addressed by the reader, but we want a more integrated behaviour. The required SQL query is:

```
select fname,lname
 from 'Employee','Department'
 where dname='Research'
   and dnumber=dno
```

The Valentine prototype contains a simple SQL compiler which translates this SQL query to a Prolog goal. (The syntax of the subgoals of this goal is not relevant to this paper --- the subgoals are shown only to indicate that the goal will perform a join of two tables in the database.)

```
qep('U15', [C,B]) :-
   database('Department', ['Research',A,_,_]),
   database('Employee', [C,_,B,_,_,_,_,_,_,A]).
```

The compiler also generates a driver which includes the subgoal `(setof(A,user:qep('U15',A),B);B=[])` which calls out a list of solutions. We shall use this in the logic document.

The integration takes place in `tuple(22,...)` which represents the paragraph `<P>`. The result is as follows

```
tuple(0, start, 'ISO-HTML',
         point(-1,[1,2,3,4,5,6,17])).
tuple(6, start, 'HEAD',
         point(0,[7,8,9,10,11])).
tuple(11, start, 'TITLE',
         point(6,[12,13,14,15,16])).
tuple(16, text,
         'Research Department Staffing', 11).
tuple(17, start, 'BODY',
         point(0,[18,19,20,21,22])).
tuple(22, start, 'P',
         point(17,[23,24,25,26,27,28,29,30])).
tuple(28, text,
         'The following people work in the \n
          research department: ', 22).
tuple(29, text, X, 22) :-
          (setof(A, user:qep('U15',A), B)
         ;
         B=[]),
         style(en,B,X) .
tuple(183, 30, text, '.', 22).
```

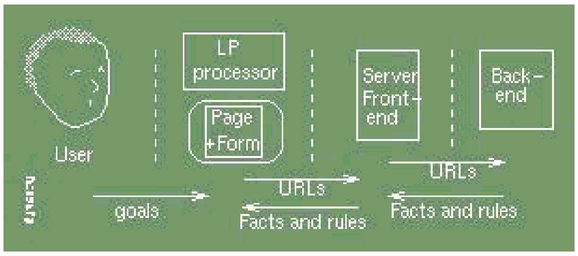
The list of children of `tuple(22, start, 'P',...)` is extended to introduce new tuples 29 and 30. The new `tuple(29, ...)` makes use of the `setof` generated by the SQL compiler to call out the current research staffers. The query returns a list of pairs of atoms. This list is converted by subgoal `style(en,B,X)` to a single atom in the style of the current language --- in our case `en` meaning English.

The resulting raw logic document can now be exported to an SGML form whenever a client requests it. Each time it is exported back to SGML it provides the latest news on the research staffing.

This simple example was hand-crafted. A more realistic example would require a rigorous definition of the what constitutes an allowable extension to the raw logic document, since the extension has broken the rule that the raw logic document is a set of fully ground unit clauses.

Discussion of the third approach

The third approach, like the second, is orthogonal to the first. It would be possible to combine the LogicWeb approach with our logic documents:



Logic documents compared with marked up text

We compare in more detail the logic documents we propose with the more traditional marked up text:

- **Access to structure**. The logic documents are represented in a parsed form: no further parsing is required to extract the structure. This facilitates queries on the structure, and manipulation of that structure. Such queries are not currently common due in part to the absence of SGML support in many tools, and the difficulty of performing the search operation on the source markup.
- **Uniformity of language**. The logic documents we propose are expressed in Prolog --- a language which also offers rapid development of agents to provide richer behaviour. The unity of language eliminates much of the engineering complexity of current web services. It would be of interest to investigate the specification of style sheets in Prolog. This would extend the uniformity further across the spectrum of document specification.
- **Computational content**. The example shows *computational content*, i.e. document content, in our case a list of names rendered as English text, obtained by computer processing at the moment the document is accessed. This is not directly (when the same effect is obtained in current documents it is often achieved through an architecturally cumbersome use of extended URIs which are processed across a CGI interface) possible in marked up representations such as HTML, and presents an opening for future research. The ISO-HTML language is intended as a simple delivery vehicle and is not rich enough to provide an archival form. Extracting the delivery language from the archival language on an on-demand basis will become increasingly common and will be facilitated by representations which assist the computation of content.

- **Emphasis on declarative language**. SGML based document grammars are, like Prolog, declarative languages, however they do not allow the direct specification of additional agent behaviour. Such behaviour is usually provided in current web offerings by imperative programs shipped via extended usage of the HTTP protocol. The logic documents we propose avoid this uncomfortable cohabitation, and provide a cleaner architectural base: documents are inherently declarative and should always be treated as such.

Fourth approach: A logic based document description language

The third approach imported and exported SGML documents and relied on SGML technology. SGML is a complex language. It is not possible to build parsers using tools such as Lex and Yacc. The parser must be handcrafted. Luckily an excellent parser has been made freely available by James Clark, but its use calls for further system engineering.

The SGML language has its origins in several text processing systems from the 60's and early 70's. The creation of the international standard was a remarkable achievement, and has turned into a success story despite the too many baroque features. It now has a considerable following, particularly in large organisations.

There have been proposals to simplify SGML and make it easier to develop tools. XML is an example of this, but XML is now being considered as a dialect of SGML. It may one day become an official profile.

Now is not the time to propose a replacement for SGML but it is time to begin work that will be needed when a replacement is finally called for. Such work needs to address the specification of the semantics as well as the syntax of document instances.

Logic programming has much to offer: a true declarative programming idiom, grammar productions, effective parsing, constraint programming, and an affinity with data base techniques.

A clear migration path must be available for the mass of documents marked up in SGML-specified document types. In addition any new proposal must be able to support:

1. The specification of a grammar to which the document is to conform. There is currently a movement to abandon document grammars and rely entirely on style sheets and ignore what you don't understand error management, but grammars and validation will always be required for high value applications.
2. Full internationalization, including simultaneous texts in multiple languages.
3. Sufficiently powerful naming to cover the need for local identifiers, formal public identifiers [Goldfarb p.382] and universal resource identifiers.
4. Mechanisms for associating structure with text in the manner of SGML's elements, and mechanisms for providing the elements with attributes in a cleaner, recursive manner.
5. Facilities for assigning and maintaining a security classification for documents and parts of documents.
6. One or more techniques for providing semantics and transformations for the documents. This includes both the formal definition of the semantics and the transformations, the programming of the transformations and the style sheets for document renderings.
7. A module system which will cover the needs for entities and subdocuments.

8. Techniques for specifying time lines and synchronisations.

SGML as an idea supposes that the text of the document will be exposed to the user: the equivalent in the database world is to expose the internal physical details of the blocking of records on the storage media to the customer. The database world has long since abandoned this in favour of *views* of the data perceived through queries expressed in a query language.

The same evolution may be desirable in the document description world. In other words, whatever replaces SGML is not a physical markup language but a means of specifying a logical view of a marked up document. The physical markup details are left to the ingenuity of the developer, in the same way that the internals of a Prolog processor are not exposed to the LP programmer.

Conclusion

Taking Loke and Davison's LogicWeb as a first approach, the paper has discussed alternative approaches to web logic programming in an attempt to get further beyond results already achieved with imperative methods.

The second approach has concentrated on SGML, the foundation of HTML. It has dug deeper into the structure of a web page and attempted to achieve a more intimate relationship between document elements and their attributes, and a Prolog program. This is achieved at the cost of more complexity.

The third approach attempted to reduce the complexity by expressing the document and the additional logic programming in the same language. We have shown how any SGML-based document may be expressed as a *logic document*. This places the logic programmer in a position to achieve by simple means hyperdocument behaviour which would otherwise require considerably more engineering. It also shows a separation between the author's markup of a text document and the structure used for processing.

Document structuring will be receiving increasing attention in the future. The XML activity led by the World Wide Web Consortium is attempting to produce a context free dialect of SGML to facilitate the production of web tools. The approach is essentially SGML based and assumes that the source document will contain both the source text and the markup which could be inserted manually.

The intention of the fourth approach described in this paper is to get beyond this continuation of 1970s techniques and propose a deep review of document description taking advantage of ideas more thoroughly anchored in computer science and facilitated by the use of logic programming.

Future work

There are many areas in logic document management which require further work. Some examples are:

1. Evaluate the relative performance of processing logic documents using Prolog, SP-based systems and Thot-based systems. Performance aspects include not only the processing time but also the time taken to create document and corpus queries and develop the application programs required to process the results.
2. Investigate the automatic processing of an SGML document type declaration to produce equivalent

rules constraining a logic document. This work should take acount of the need to tolerate structural errors during document manipulation.
3. Extend the work to address the HyTime architectural extensions of SGML for time-based multimedia documents.
4. Evaluate the relative expressiveness and efficiencies of the DSSSL, CSS and pure logic programming to provide semantics for logic documents.
5. Investigate the advantages of using logic programming systems other than Prolog for document structuring and processing.

Acknowledgements

The author is very pleased to acknowledge the help of his co-editor David Abrahamson in preparing the ISO-HTML specification. He is also indebted to the IETF members and W3C staff members who have commented and advised on the development of ISO-HTML. John Buford made many comments which have helped to improve this paper.

Further acknowledgements for the major revision

The anonymous referees who reviewed this paper for the workshop made many pages of valuable comments for which the author is indebted and which have led to a major revision. Although he is unable to cite them, the author is very happy to acknowledge their work.

The errors, the omissions and the obscurity are the author’s own work.

Last revision May 22nd 1997