# Balisage

## Series on Markup Technologies

*Balisage: The Markup Conference*

# Invisible XML coming into focus

### *Status report from the community group*


## Tomos Hillman

eXpertML Ltd

<tom@expertml.com>

## John Lumley

<john@johnlumley.net>

## Steven Pemberton

Centrum Wiskunde & Informatica (CWI)

## C. M. Sperberg-McQueen

Black Mesa Technologies LLC

<cmsmcq@blackmesatech.com>

## Bethan Tovey-Walsh

Swansea University

<bytheway@linguacelta.com>

## Norm Tovey-Walsh

Senior Software Developer

Saxonica

<ndw@nwalsh.com>



***Balisage: The Markup Conference 2022***

August 1 - 5, 2022



“Invisible XML coming into focus” copyright © 2022 by Tomos Hillman and others is licensed under CC BY-NC-SA 4.0.



**How to cite this paper**

Hillman, Tomos, John Lumley, Steven Pemberton, C. M. Sperberg-McQueen, Bethan Tovey-Walsh and Norm Tovey-Walsh. "Invisible XML coming into focus." Presented at: Balisage: The Markup Conference 2022, Washington, DC, August 1 - 5, 2022. In

*Proceedings of Balisage: The Markup Conference 2022*. Balisage Series on Markup Technologies vol. 27 (2022). `https://doi.org/10.4242/BalisageVol27.Eccl01`.

## Abstract

Invisible XML has had a long incubation process, but in the last year things have heated up. A W3C Community Group has been formed, the spec has been improved, and implementations have been released or are in various stages of development. This paper gives an overview of iXML in its stable version 1.0 form, with discussion of some of the design decisions that have shaped it, and accounts from implementors of their practical experiences with iXML.

---

# Invisible XML coming into focus

## *Status report from the community group*

## Table of Contents


- Title Page
- Introduction
- About iXML
  - Abstractions
  - What iXML does
  - Processing
  - A Simple Example: Dates
  - Attributes
  - Ambiguity
- What's New
  - Significant changes
  - Updates to the specification
  - Infrastructure
  - What next?
- Implementations
  - ixampl
  - JayParser
  - Hywel
  - Aparecium
  - jωiXML
    - The XPath3.1 grammar
  - NineML
    - CoffeeGrinder
    - CoffeeFilter
    - CoffeePot
    - CoffeeSacks
    - CoffeePress
- About the Authors

# *Invisible XML coming into focus*

## *Status report from the community group*

## Introduction

“What if you could see everything as XML?” (Pemberton 2013). This question, posed by Steven Pemberton at the Balisage conference in 2013, marked the first public appearance of Invisible XML (iXML). Pemberton proposed that documents authored in non-XML formats could be brought into the XML ecosystem via an intermediary technology capable of recognizing the explicit or implicit structure of those documents. This would offer the substantial advantages of the XML stack for data processing without requiring that all documents, under all circumstances, be authored in XML.

Since that initial talk, the ideas of invisible XML have been refined and elaborated (e.g., Pemberton 2016a, Pemberton 2016b, Pemberton 2017, Tovey-Walsh 2022a, Tovey-Walsh 2022b), and scattered reports of prototype implementations soon appeared. Interest in iXML grew slowly but steadily, and in 2021 an iXML Community Group (CG), hosted by the World Wide Web Consortium, was formed. The main task of the CG has been to collaborate on an official version of the language in the form of a published specification. In June 2022, iXML version 1.0 was formally released on invisiblexml.org. The CG also aimed to encourage implementation and uptake. At the time of writing there are six known implementations, in various stages of completeness.

The release of version 1.0 marks a significant step forward for iXML. The existence of an official specification offers implementors a stable target for implementation, and offers users assurance that they can use the current version of iXML without fear of arbitrary changes. In this paper the members of the iXML community group present the completed iXML version 1.0 to the larger XML community.

We begin with an overview of what iXML is, what it does, and how it does it. Some of this may be familiar to those who have been following the development of iXML, but there will be at least a few new details here for everyone, reflecting the recent work of the iXML CG. We will then report on some of the new features recently added to iXML in preparation for the release of version 1.0. Finally, implementors will offer insights into their

practical experiences with iXML, including some which helped to shape their contributions to the iXML 1.0 specification.

## About iXML

### *Abstractions*

One of the key insights which led to the development of iXML was that, in basic terms, data exists as an abstraction distinct from the form in which it is represented. To illustrate this distinction, Pemberton (Pemberton 2016a) uses the example of the number *three*. Numbers are entirely abstract. They have no concrete existence per se. (The abstraction *three* is a property of a group of three sheep and of a group of three books, but neither of these groups is "a three".) A numerical abstraction such as "three" can be represented in a variety of ways, depending on linguistic and cultural contexts:

> 3, III, 0011, 三, ३, ৩, ૩, ੩, 𐄉, ௩, ൩, ೩, 〣, ๓, ༣, ᠓, trois, drie.

One might choose Roman numerals for the volume number of a book, binary notation in a mathematical context, "trois" when writing in French, and so on. In each case, however, the basic denotational content is the same: the underlying numerical abstraction *three* is shared by all of these representations.

This distinction can be applied to all kinds of data, not only to pure mathematical abstractions; the abstraction is simply what remains constant regardless of the concrete details of the representation. We may be obliged or encouraged to represent data in a particular way. An American textbook is likely to use the representation "32℉" to refer to the freezing point of water where a British textbook would use "0℃", but the temperature in question is the same. An author might choose to represent a temperature in JSON:

```
{"temp": {"value": 32; "units": "fahrenheit"}}
```

or in XML:

```
<temp value="32" scale="fahrenheit"/>
```

or

```
<temp>
    <value>32</value>
    <scale>fahrenheit</scale>
</temp>
```

or, indeed, in plain text

```
temp: 32℉
```

The underlying abstraction is the element which remains constant in all of these representations.

### *What iXML does*

iXML takes some input data (an *iXML input stream*) and a formal description of the implicit structure of that data (an *iXML input grammar*). It uses the grammar to create an internal representation of the data, with the structure made explicit. This internal representation can then in principle be used for multiple purposes, including creating an external representation by serializing to a particular markup format. In this process, the iXML notation plays multiple roles: it describes the syntax of the input, it describes how the abstract document that results from parsing should be serialized to XML, but it also serves as a schema for the abstract document, describing its structure. What this means is that the abstract document can be used in different ways, in addition to being serialized to XML. For instance, it can be converted internally in memory to the form needed by the parser, or converted internally into an XDM (Walsh et al., 2017). Nonetheless, after much discussion, the CG decided that one feature which must be shared by all conforming iXML processors is that they serialize their output to XML. This ensures that users can always expect to receive an XML document as output from any processor they choose, even if the implementor offers other representations in addition to an XML serialization. This decision was based on many factors, including the particular merits of XML as an external representation format. Most evidently, anyone with access to an

XML document and tools for processing XML can process it as they choose, or convert the XML into other forms; this is not true in the same way for the internal representation of the document, which is accessible only to the programmer who wrote the iXML processor in question.

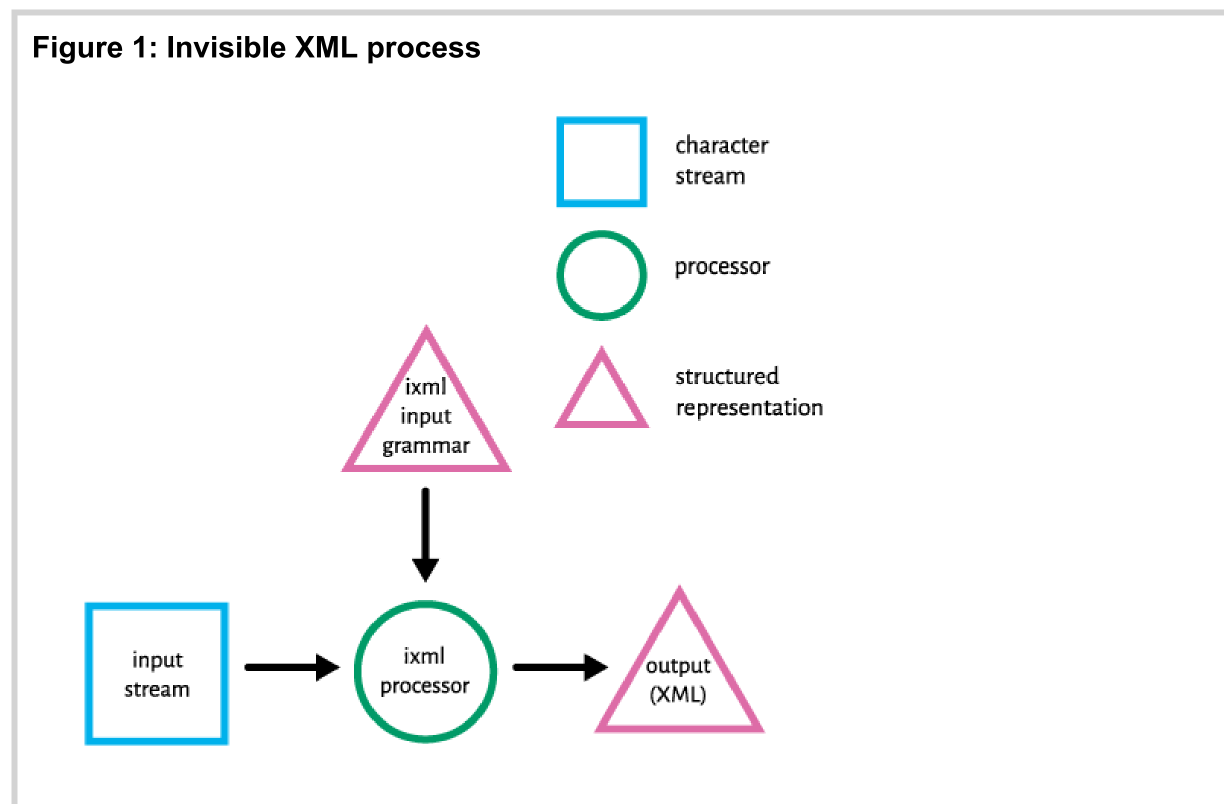
External representation formats vary in representational power; more powerful representation formats are able to represent the features of the underlying abstraction in more detail than less powerful ones. This means that a conversion from a more powerful representation format to a less powerful one will often entail a loss of information.The initial task of an iXML processor is to separate data abstractions from their representations, whether those representations are already explicitly structured (e.g., in a format such as CSV) or only implicitly structured (e.g., using punctuation conventions and natural-language syntax). The second task is to re-represent the data abstractions, using the structural information provided by the grammar to externalize implicit structure. It makes sense to perform this re-representation using a powerful representational format, in order that the final representation is as rich as possible in its encoding of the structures implicit in the input.

Although it is by no means the only powerful representation format available, XML has unique features which make it particularly appropriate for the purposes of iXML. The richness and sophistication of XML are widely acknowledged, including by some who ultimately choose to use alternative representation formats such as JSON (e.g., Shatnawi et al., 2021, Bahta et al., 2019). Even detractors' criticisms of XML as "utterly verbose" (Lee et al., 2021) only serve to emphasize one of the main attractions of XML for re-encoding data representations into a more powerful format: XML privileges detail and informativity over the compactness which makes JSON an appealing alternative in some other contexts. XML and JSON each have strengths and weaknesses in different areas, meaning that neither can be called the outright best choice for all possible uses (Bourhis et al., 2020). However, XML is a particularly good choice when the aim is to maximize the depth of information represented, and also benefits from a particularly mature and stable ecosystem of supporting technologies (Dou et al., 2020).

### *Processing*

A conforming iXML processor always produces an XML document as output (Pemberton 2022). We have already mentioned that an iXML processor takes two inputs: the *iXML input stream* and the *iXML input grammar* , the latter of which describes the expected structure of the former. If the input grammar successfully describes the structure of the input stream, the XML document output by the processor will be a representation of the input data in XML format, with element names and attributes as defined by the input grammar. If the input grammar does not successfully describe the input stream, the processor will output an XML document containing a failure report. The basic flow of this process is illustrated in fig. 1.

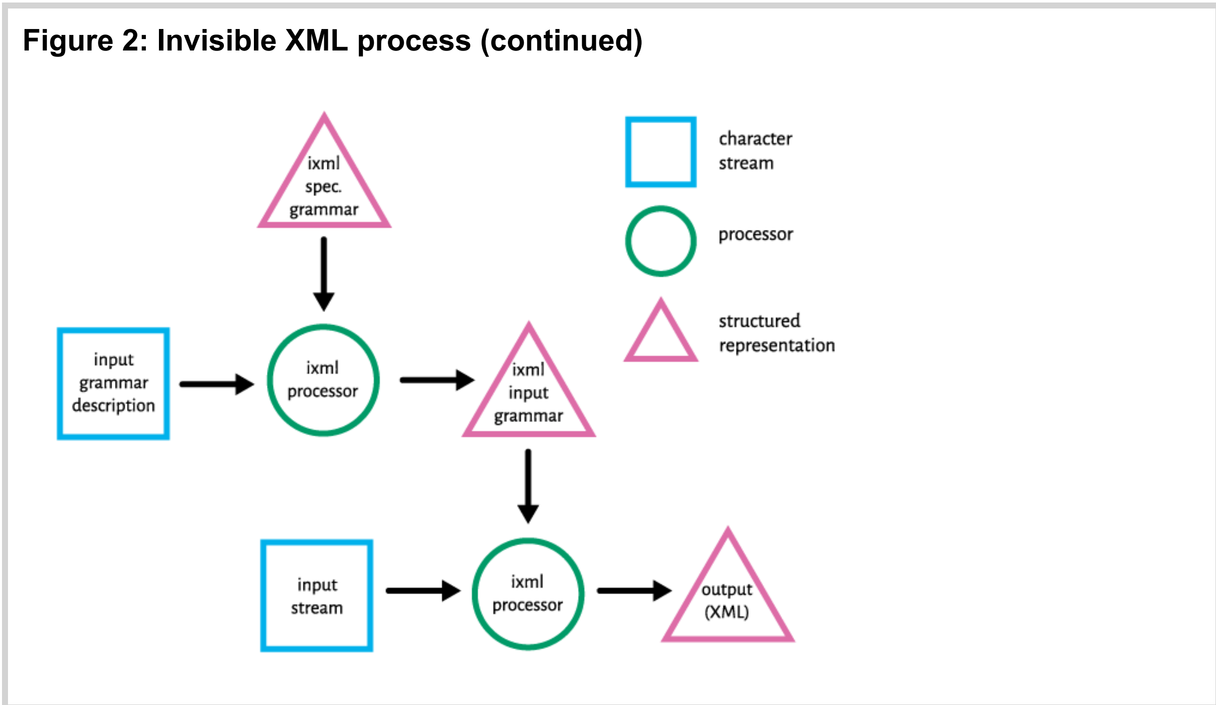
**Figure 1: Invisible XML process**




The input grammar is a structured representation in the iXML format, which is described in the iXML specification. In order to generate this structured representation, an iXML processor will normally first accept a grammar as a character stream and parse it against the iXML specification grammar. This process treats a grammar description like any other input stream, parsing its characters and (provided that the characters form a valid grammar in iXML format) outputting a structured representation of the grammar which can then be used to parse the main input stream, as illustrated in fig. 2. The major difference between this process and the

parsing of the iXML input stream proper is that no XML serialization needs to be produced. The output of this step is an internal representation of the input grammar.

**Figure 2: Invisible XML process (continued)**




The upshot of this is that iXML is, itself, represented using the syntax of iXML. For instance, here is the rule for rule:

```
rule: (mark, s)?, name, s, -["=:"], s, -alts, -".".
```

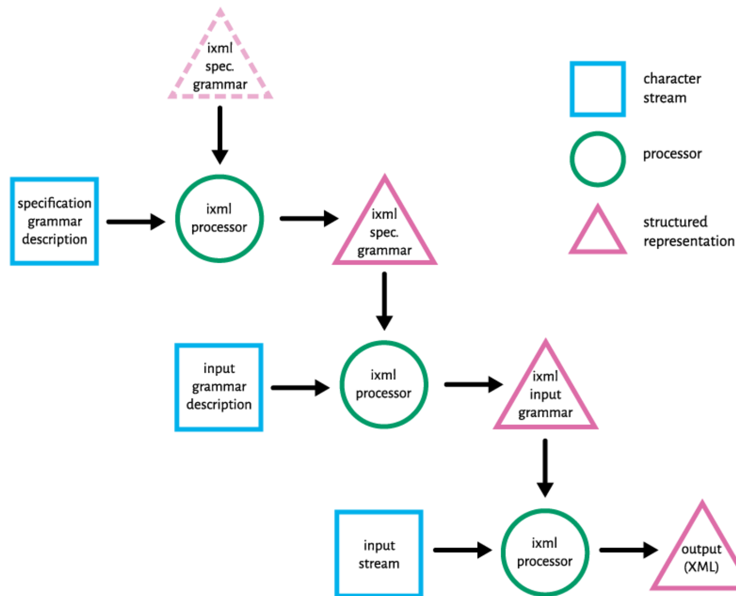
which comes out as XML

```
<rule name='rule'>
    <alt>
        <option>
            <alts>
                <alt>
                    <nonterminal name='mark'/>
                    <nonterminal name='s'/>
                </alt>
            </alts>
        </option>
        <nonterminal name='name'/>
        <nonterminal name='s'/>
        <inclusion tmark='-'>
            <member string='=:'/>
        </inclusion>
        <nonterminal name='s'/>
        <nonterminal mark='-' name='alts'/>
        <literal tmark='-' string='.'/>
```

```
    </alt>
</rule>
```

The entire processing cycle includes a bootstrap phase which produces the structured representation of iXML:

**Figure 3: Invisible XML bootstrapped process**




## *A Simple Example: Dates*

To illustrate the basic functioning of iXML with a necessarily simple example, imagine the input stream is a date of the form

```
29 June 2022
```

We describe the format as a series of rules in iXML. Some rules have only one form, with items separated by commas. Some rules have several alternative forms, separated by semicolons.

```
date: day, " ", month, " ", year.
day: digit, digit?.
month: "January"; "February"; "March"; "April";
       "May"; "June"; "July"; "August";
       "September"; "October"; "November"; "December".
year: digit, digit, digit, digit.
digit: ["0"-"9"].
```

Processing the input using this description and serializing it to XML gives:

```
<date>
    <day>
        <digit>2</digit>
        <digit>9</digit>
    </day>
    <month>June</month>
    <year>
        <digit>2</digit>
        <digit>0</digit>
        <digit>2</digit>
        <digit>2</digit>
    </year>
</date>
```

What you can see is that the rules define XML elements whose contents are one of the alternatives contained in the rule.

In this case we are not interested in the `digit` elements, so we change that one rule to exclude it from the serialization:

```
-digit: ["0"-"9"].
```

Processing the input with this new description gives:

```
<date>
    <day>29</day>
    <month>June</month>
    <year>2022</year>
</date>
```

We will now add a date format option of the following style:

```
29/06/2022
```

A rule is added to the description:

```
date: day, " ", month, " ", year;
      day, "/", nmonth, "/", year.
day: digit, digit?.
month: "January"; "February"; ...; "December".
nmonth: digit, digit?.
year: digit, digit, digit, digit.
-digit: ["0"-"9"].
```

Note that since a month in the new format has a different syntax from a `month` in the original one, it has to have a different name. Processing the input with this description gives:

```
<date><day>29</day>/<nmonth>06</nmonth>/<year>2022</year></date>
```

As this shows more clearly, all characters in the input end up in the serialization by default, and tags are effectively just placed around parts of the input to expose structure. However, characters which do not interest us can be omitted from the serialization in the same way as rules:

```
date: day, -" ", month, -" ", year;
day, -"/", nmonth, -"/", year.
```

We can specify that the input consists of one or more dates:

```
dates: date+.
```

but in this case they have to be right next to each other, with no intervening spaces. Better, then, to allow any number of spaces after a date:

```
dates: (date, " "*)+.
```

Another possibility is specifying a *separator*, in this case consisting of a comma and a single space:

```
dates: date++", ".
```

for input like:

```
29/06/2022, 31 December 2022, 1/1/2023
```

For input consisting of dates on separate lines, you can use

```
dates: (date, cr?, lf)+.
-cr: -#d.
-lf: -#a.
```

### *Attributes*

Rules can be specified as producing attributes rather than elements. In this case, the text resulting from application of the rule becomes the value of the attribute:

```
date: day, -" ", month, -" ", year;
day, -"/", nmonth, -"/", year.
@day: digit, digit?.
@month: "January"; "February"; ...; "December".
@nmonth: digit, digit?.
@year: digit, digit, digit, digit.
-digit: ["0"-"9"].
```

Processing with input

```
29/6/2022
```

gives

```
<date day="29" nmonth="6" year="2022"/>
```

### *Ambiguity*

We might want to restrict a `day` to be in the range 1-31, and an `nmonth` to the range 1-12:

```
day: "0"?, ["1"-"9"];
     ["12"], ["0"-"9"];
     "3", ["01"].
nmonth: "0"?, ["1"-"9"];
        "1", ["012"].
```

We might also find it necessary to allow both month-day and day-month date formats:

```
date: world; us.
us: nmonth, -"/", day, -"/", year
world: day, -"/", nmonth, -"/", year.
```

It is now possible that an input will satisfy the grammar in more than one way. Processing the input `04/10/2022`, for example, offers two valid parses:

```
<date>
    <us>
        <nmonth>04</nmonth>
```

```
        <day>10</day>
        <year>2022</year>
    </us>
</date>
```

and:

```
<date>
    <world>
        <day>04</day>
        <nmonth>10</nmonth>
        <year>2022</year>
    </world>
</date>
```

A conforming iXML processor must serialize one valid parse to XML as output, and must also report that the parse was ambiguous. For example, processing the input `04/10/2022` using Pemberton’s *ixampl* implementation produces:

```
<!-- AMBIGUOUS
The input from line.pos 1.1 to 1.11 can be interpreted as 'date' in 2 different
ways:
1: us[:1.11]
2: world[:1.11]
-->
<date ixml:state="ambiguous" xmlns:ixml="http://invisiblexml.org/NS">
<us>
<nmonth>04</nmonth>
<day>10</day>
<year>2022</year>
</us>
</date>
```

The CG chose not to specify how processors should choose which parse tree is represented in the output. This decision was largely motivated by a desire to avoid interfering unnecessarily with implementation choices. In very simple terms, the recognition of ambiguities may vary depending on the parsing technique used. A fuller discussion of the difficulties with ambiguity is given below, explaining in detail the reasoning behind the CG's decision.

## What's New

### *Significant changes*

In the last year, the working group has mostly been polishing the language and improving the specification. However, there have also been three significant changes to the language.

Firstly, the syntax for separators has changed from

```
date+", "
```

to

```
date++", "
```

This change was made to improve usability. Given the prior syntax, if an author accidentally omitted a comma, turning

```
input: word+, "!".
```

into

```
input: word+  "!".
```

the result was still syntactically valid. This kind of mistake was felt to be difficult to locate, particularly in large grammars.

The second major change is the introduction of a method for inserting text. While it was previously possible to exclude characters that were part of the input from the XML output (using `-"..."`), there is now a symmetrical notation for adding new characters to the output, using `+"..."`.

For instance

```
number: pos; neg.
-pos: +"+", digit+.
-neg: +"-", -"(", digit+, -")".
```

would serialize numbers like `123` as `+123`, and numbers like `(123)` as `-123`.

The third change deals with character sets. The notation `["0"-"9"]` matches any single character in the range. Similarly, the notation `~["0"-"9"]` matches any single character not in the range. Originally such a character set was not allowed to be empty, since this was thought to have no useful

purpose, so `[]` was disallowed. However we realized that `~[]` did have a useful meaning ("match any single character"), and consequently empty character sets are now allowed.

### *Updates to the specification*

Changes to the spec since the community group began its work have tended to make the text more explicit and complete. The specification now mentions several possible parsing algorithms, for example, in addition to Earley parsing. Most prominently, perhaps, the group has added explicit rules for conformance of grammars and processors, a few of which have already been mentioned above.

- Conforming grammars must match the iXML specification grammar.

  In addition, various non-structural requirements are imposed.
- Nonterminals which are to be serialized as XML element or attribute names must be legal XML names.
- Character data to be written to the output must be legal XML characters (although the input may include non-XML characters).
- Hexadecimal numbers used for encoded literals must fall within the Unicode character range.
- Character classes used in character-set expressions must be classes defined by Unicode.
- Character ranges must be well formed (that is, their starting point must not follow their ending point).
- The output to be produced must be well-formed XML.

  Note that it is not always feasible to prove that a grammar will produce well-formed output for all possible inputs; some errors may be detected only dynamically, in the presence of input which would cause ill-formed output if serialized in the normal way.

Explicit rules for conformance of grammars have also been added. Processors must accept all conforming grammars, detect errors in non-conforming grammars, produce a parse tree for any input stream which is

recognized by the supplied input grammar, not produce parse trees for input streams which do not match, and so on.

Some effort has gone into clarifying the behavior of processors in cases where the input is ambiguous; to our surprise, a crisp definition of *ambiguity* in the iXML context has proven elusive: depending on the internal structure of the processor, ambiguity may or may not be flagged for a given input grammar and input stream.[1]

### *Infrastructure*

In addition to the specification, the community group is working to develop infrastructure useful to implementors and users of iXML. A collection of sample iXML grammars has been started on the community group's github site (Invisible XML CG, eds., 2022) and the group plans to use it to provide iXML grammars representing published notations. This repository can serve both as a library of useful grammars, for notations of broad interest, and as illustrations of iXML usage. Among the grammars currently available are:

- XPath 3.1.
- URI and IRI.
- ISO 8601 dates.
- The Oberon programming language.
- ABNF (Augmented BNF for Syntax Specifications), the grammar notation defined by RFC 5234 and used in IETF specifications.
- ISBN (International Standard Book Number); the grammar checks the correct calculation of the check digit in ISBN-13 numbers.

For implementors, test cases are needed. So far, a collection of a few hundred test cases has been built, ranging from toy grammars describing fragments of CSS or other well known notations, to larger grammars, to very small grammars aimed at finding and exposing errors in the logic of pursuers. The test cases can be browsed on the Web.

Some stylesheets for manipulating iXML grammars may be found in the Gingersnap project; they have been used, inter alia, to generate test cases for iXML, to measure test suite coverage, and to generate Relax NG schemas describing the XML documents generated by iXML grammars.

### *What next?*

With the publication of the 1.0 specification of iXML this past June, iXML has passed a major milestone. The availability of useful iXML processors allows people outside the community group to experiment with iXML to solve real-life problems and to build applications using iXML.

If you have been waiting for iXML to mature a bit before looking into it, then: the time is now.

## Implementations

The progress of iXML implementations is possibly the most important practical development we can report. There are now several publicly available implementations of iXML 1.0, as well as others announced as being in development (Invisible XML CG, eds., 2022). Two of these, JayParser and Aparecium, are prototype or proof-of-concept implementations. They have served the purpose of demonstrating that iXML is implementable within the standard XML technology stack, but they both share a common characteristic of prototype implementations: they tend to be resource intensive. That is to say, they run slow and take a lot of memory. Both implementors have hopes of improving the situation, but for now these implementations are useful mostly for demonstrating what might be possible. For practical work, less resource-intensive implementations are needed. Fortunately, there are currently three of these, which run two or three orders of magnitude faster than the proof-of-concept implementations.

The following sections are authored by individual implementors. Some are brief introductions to implementations released or in development; others are longer and more substantive. All six implementors have also been involved in the CG, and found that practical experience of implementing iXML was a significant asset in the work of refining the specification.

### *ixampl*

The first implementation of iXML was created by Steven Pemberton to support the development of the language; it is further described in Pemberton 2016b and Pemberton 2022b.

It is written in the interpreted Very High Level Language ABC (Guerts et al., 1990). It is offered as a RESTful web-service application, by submitting a grammar and input either via (a webpage) or via a command-line interface.

Details of how to access the web service are kept up to date in the iXML tutorial (Pemberton 2022c).

It was deployed successfully at Declarative Amsterdam 2021 with multiple simultaneous users, and has been used on grammars the size of the XPath grammar.

### *JayParser*

Tomos Hillman's JayParser (Hillman 2020) implements an Earley parser in XSLT, with the goal of making it easy to integrate ixml processing in an XSLT work flow. Its current main challenge is that it makes exuberant use of memory, which means that it is limited to using small sample grammars to parse very very small inputs.

### *Hywel*

*Hywel* is a Python implementation of iXML being developed by Bethan Tovey-Walsh. There is, as yet, no publicly available version of Hywel, although a version 1.0 is tentatively promised for late 2022. The implementation aims to be of particular use for linguistic tasks such as part-of-speech tagging. Optimizing Hywel for use with large inputs and large grammars is therefore currently the primary obstacle to its release as a useful tool. The pragma syntax developed by Hillman and Sperberg-McQueen (Hillman et al., 2022), will form an integral part of Hywel's natural-language-processing architecture.

### *Aparecium*

Aparecium (Sperberg-McQueen 2019) provides an XQuery implementation callable as a library function: a call to `aparecium:parse-string($input, $grammar)` takes a string to be parsed and a string containing an ixml

grammar and returns the parse tree in XML (or, in cases of failure, an XML document describing the failure). Alternative functions in the API allow the input and grammar to be given as URIs instead of strings.

Using Aparecium to load a non-XML resource for processing in an XQuery module is almost as simple as calling the `doc()` function to load an XML resource. If you want to query a collection of electronic business cards in vCard format, for example, and return all the cards of your contacts at Amalgamated Interkludge, the module might look something like this:[2]

```
import module namespace aparecium
  = "http://blackmesatech.com/2019/iXML/Aparecium"
  at "my-lib/ap/Aparecium.xqm";

let $cardfile := "contacts/work.vcf",
    $grammar := "my-lib/grammars/vcard.ixml",
    $cards := aparecium:parse-resource($cardfile, $grammar)

return $cards//property[@name="ORG"]
        [contains(value,"Amalgamated Interkludge")]
```

Aparecium is usable for inputs of a few tens or hundreds of characters but currently suffers from non-linear performance even on deterministic grammars: doubling the size of the input quadruples the time required to parse the input (or worse). This is not inherent in the Earley parsing algorithm used, so the developer hopes to improve the situation by focused attention to performance issues, sometime in the near future. Real soon now.

### *jωiXML*

*jωiXML* is An InvisibleXML processor for a JavaScript/SaxonJS environment. The impetus for its development was that, being involved in the InvisibleXML community group, I felt that I didn't understand some of the issues sufficiently to make good contributions, particularly those associated with ambiguity. And in such a circumstance a tactic I often use is to learn by building my own implementation. I had some experience in building hand-written parsers when further developing the XPath expression compiler for SaxonJS (Lumley 2017a). I also value highly using an environment where I can see a lot of the 'innards working', with my

preference being to use SaxonJS in the browser to provide the interface for control and display with an additional JavaScript module to contain the main code. The Chrome browser has a sufficiently useful Javascript debugger to be able to focus on areas of concern.

The overall architecture would be a parser written in JavaScript consisting of three main parts —; a set of classes representing the main productions of the iXML grammar (`Rule`, `NonTerminal`, `Charset` etc.), a parser that would take the text of an input grammar and construct the internal 'parse tree' from linked instances of those classes and a 'grammar' object which, loaded with a set of rules from such a parse tree, can then parse input text strings and, if the parse is successful, generate the declared XML result tree. This JavaScript would be loaded into a web page whose dynamic activity was controlled by an XSLT program compiled for SaxonJS execution. Thus all the complexity of detailed display, control, loading grammars, running test suites and so forth is handled at a high level, using the XSLT mechanisms we are comfortable with.

Communication from the XSLT with the parser was handled by two actions — `jwl:compileGrammar($grammarSource as xs:string)` which invokes the JavaScript-defined 'grammar factory' and parses/compiles the supplied iXML (or also XML form) grammar, returning a `Grammar` JavaScript object. Parsing an input then involves invoking the `parse()` method of that `Grammar`, by the SaxonJS invocation function `ixsl:call($grammar,$input as xs:string)` and currently the return value is a map containing parsing success, result trees, timing information and the internal parsing states (used for debugging —; see below).

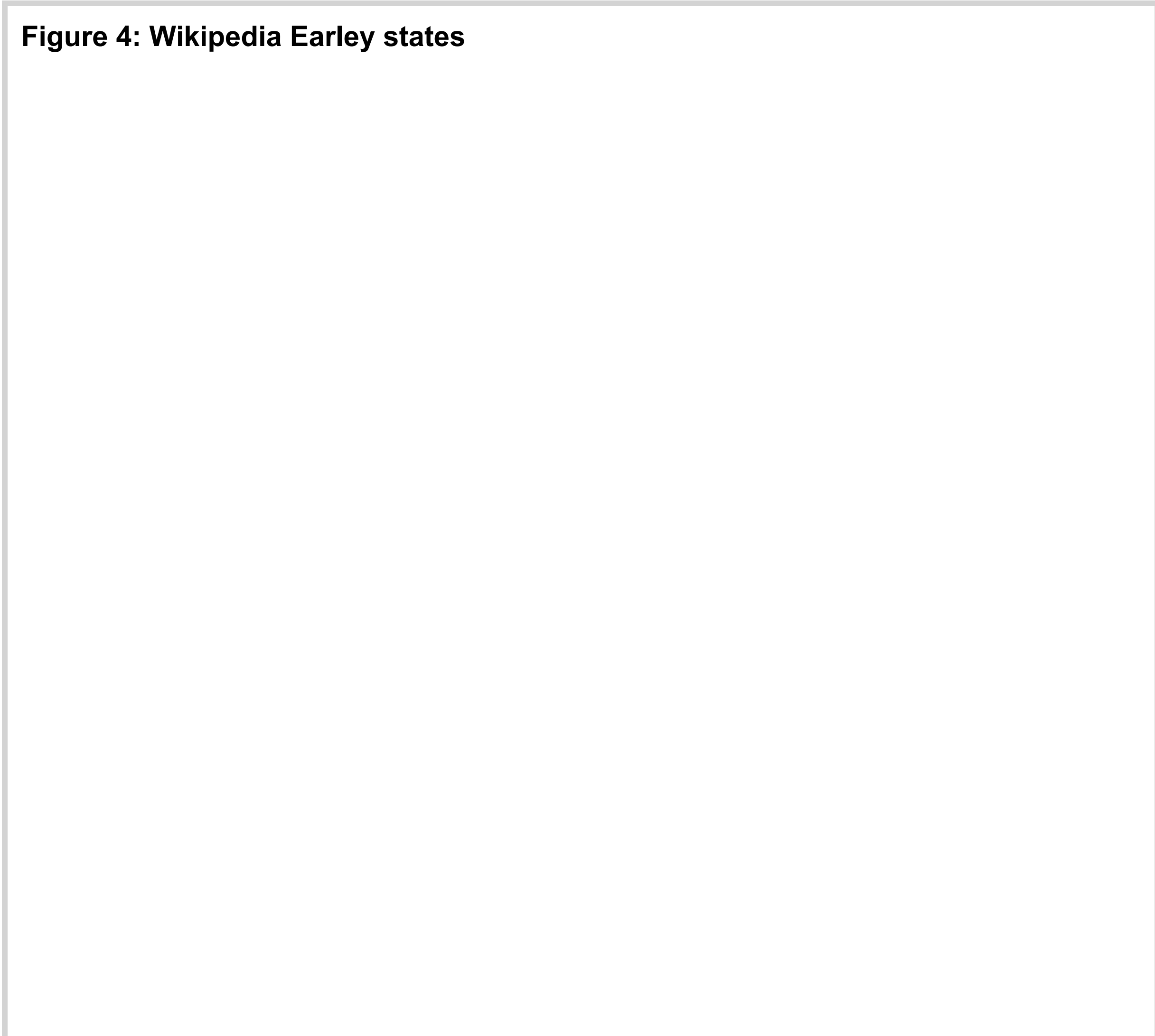
One approach for designing the 'grammar parser' of an iXML processor is via a bootstrap, using the XML version of the iXML specification grammar, parsable with current XML readers, to initialize the parsing engine to accept an iXML grammar. This is then used to parse the application grammar, which being collected, is used to re-initialize the engine to parse future selected application inputs.

Two drawbacks of this process are that firstly it requires a moderately performing parsing engine (e.g., an Earley parser) to run before anything concrete can happen and secondly parsing an iXML grammar with a generic engine will perforce be slower, though more flexible, than one

purpose-written. For these reasons I chose to write my iXML parser directly in JavaScript, generating the internal class tree describing productions. In fact this was probably the easiest part of the whole development, aided by the fact that the changes made to the iXML specification over this period were relatively modest.

I then turned to writing the Earley parser, choosing to 'build my own' rather than taking an off-the-shelf version, partly as a learning exercise. To do this, the example given in the Wikipedia article (https://en.wikipedia.org/wiki/Earley_parser#Example) which parses an arithmetic expression against a simple grammar was my first target. This was exceptionally useful as I had a view of the expected internal states through the Earley parse of a simple arithmetic expression and could ensure that my implementation generated similar information, as shown below:

**Figure 4: Wikipedia Earley states**



| (state no.) | Production | (Origin) | Comment |
|---|---|---|---|
| S(0): • 2 + 3 * 4 | | | |
| 1 | P → • S | 0 | start rule |
| 2 | S → • S + M | 0 | predict from (1) |
| 3 | S → • M | 0 | predict from (1) |
| 4 | M → • M * T | 0 | predict from (3) |
| 5 | M → • T | 0 | predict from (3) |
| 6 | T → • number | 0 | predict from (5) |
| S(1): 2 • + 3 * 4 | | | |
| 1 | T → number • | 0 | scan from S(0)(6) |
| 2 | M → T • | 0 | complete from (1) and S(0)(5) |
| 3 | M → M • * T | 0 | complete from (2) and S(0)(4) |
| 4 | S → M • | 0 | complete from (2) and S(0)(3) |
| 5 | S → S • + M | 0 | complete from (4) and S(0)(2) |
| 6 | P → S • | 0 | complete from (4) and S(0)(1) |
| S(2): 2 + • 3 * 4 | | | |
| 1 | S → S + • M | 0 | scan from S(1)(5) |
| 2 | M → • M * T | 2 | predict from (1) |
| 3 | M → • T | 2 | predict from (1) |
| 4 | T → • number | 2 | predict from (3) |
| S(3): 2 + 3 • * 4 | | | |
| 1 | T → number • | 2 | scan from S(2)(4) |
| 2 | M → T • | 2 | complete from (1) and S(2)(3) |

**Figure 5: jωiXML Earley states**

**Parser states**

| # | id | charPos | Production | origin | Comment |
|---|---|---|---|---|---|
| **S(0): •2+3*4** | | | | | |
| 1 | 0 | 0 | (1)P→•S | 0 | start rule |
| 2 | 0 | 0 | (2)S→•S,"+",M | 0 | predict from (1) |
| 3 | 1 | 0 | (3)S→•M | 0 | predict from (1) |
| 4 | 0 | 0 | (4)M→•M,"*",T | 0 | predict from (3) |
| 5 | 1 | 0 | (5)M→•T | 0 | predict from (3) |
| 6 | 0 | 0 | (6)T→•["1"-"4"] | 0 | predict from (5) |
| **S(1): 2•+3*4 char '2' (codepoint 50,#32)** | | | | | |
| 1 | 0 | 0 | (1)T→["1"-"4"]• | 0 | scanned from S(0)(6) |
| 2 | 1 | 0 | (2)M→T• | 0 | complete from (1) and S(0)(5) |
| 3 | 1 | 0 | (3)S→M• | 0 | complete from (2) and S(0)(3) |
| 4 | 0 | 0 | (4)M→M,•"*",T | 0 | complete from (2) and S(0)(4) |
| 5 | 0 | 0 | (5)P→S• | 0 | complete from (3) and S(0)(1) |
| 6 | 0 | 1 | (6)S→S,•"+",M | 0 | complete from (3) and S(0)(2) |
| **S(2): 2+•3*4 char '+' (codepoint 43,#2b)** | | | | | |
| 1 | 0 | 1 | (1)S→S,"+",•M | 0 | scanned from S(1)(6) |
| 2 | 0 | 1 | (2)M→•M,"*",T | 2 | predict from (1) |
| 3 | 1 | 1 | (3)M→•T | 2 | predict from (1) |
| 4 | 0 | 2 | (4)T→•["1"-"4"] | 2 | predict from (3) |
| **S(3): 2+3•*4 char '3' (codepoint 51,#33)** | | | | | |
| 1 | 0 | 2 | (1)T→["1"-"4"]• | 2 | scanned from S(2)(4) |

Once this example was working and all the parse states seemed to correspond, I turned to generating the resulting parse tree, which involved i) adding 'derivation pointers' between the Earley states and ii) building a 'bottom-up' traversal from the completion states through the derivation tree, generating sequences of text, attribute and element nodes (in a browser-supported DOM tree) according to the 'mark' properties of the non-terminal rules and references and attaching them to parent elements on the return.

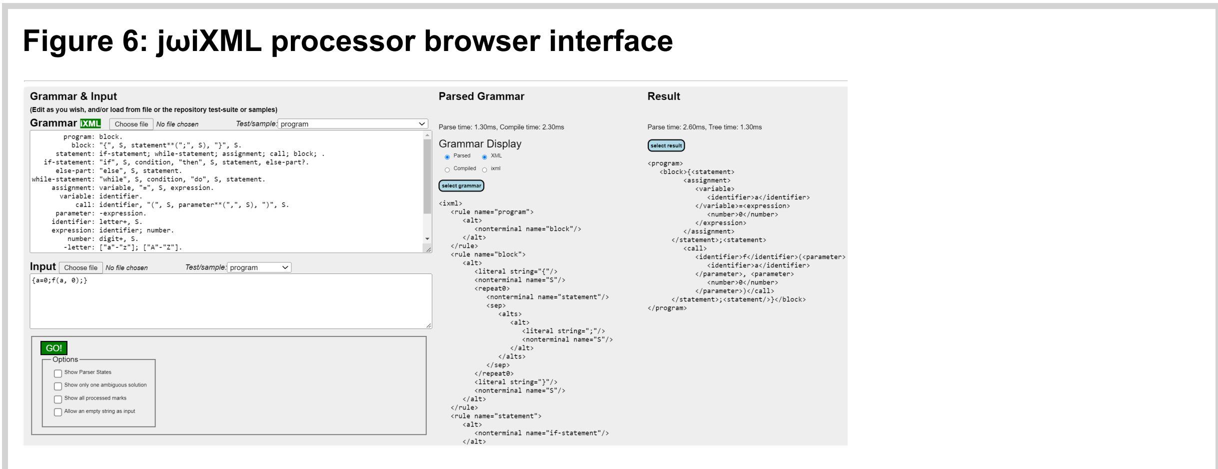
This simple example did not handle any optional or multiple constructs (`?`,`*`,`++Sep` etc.), so the next step was to transform the supplied grammar, canonicalizing any such forms using the substitution rules described in the iXML specification —; this was not tricky, but the browser interface was

expanded to be able to view the compiled grammar in various formats. However this opened a Pandora's box of plentiful opportunity for ambiguity, which required complex (and hard-to-debug) code to back-propagate and combine ambiguities — even now there are occasional internal errors triggered by incorrect combination. And there were also possible infinite ambiguities to be handled :-(.

Working this up through a series of increasingly larger and more complex cases, a significant landmark was the first time it was possible to parse the iXML grammar with itself — taking around 250–300ms it was reasonably quick. At this point it was worthwhile starting to do some more systematic testing so a testdriver, again operating in the browser, written in XSLT and driven from the test-suite files in the InvisibleXML repository was constructed.

The next step was to make the engine more easily usable, so the design expanded to allow grammars and input to be loaded and edited (entirely within the browser) through editable text areas, drop-down selectors that could load tests and samples from the InvisibleXML repository and selectors from the user's local filestore. Finally it was placed publicly at https://johnlumley.github.io/jwiXML.xhtml

**Figure 6: jωiXML processor browser interface**




## The XPath3.1 grammar

One of the goals of the InvisibleXML Community Group is to show that the technology can have a role as a lightweight front-end parser for XML workflows even on an industrial scale and as such various larger and more

complex grammars have been constructed and tested. One that I have had much experience of is that of XPath 3.1, used extensively within an XSLT compiler, so it was natural to see how a modification of the EBNF (https://www.w3.org/TR/xpath-31/#id-grammar) into iXML would work.

Generally the transcription was fairly straightforward, with a few additional pseudo-productions needing to be added. For example, to allow (variable) operators to be projected as attributes in the resulting parse tree (e.g., `("+"|"-")` becoming `@AddOp: s?,'+',s?; s,'-',s.`) This worked pretty well and the resultant grammar parsed sample XPath expressions of moderate complexity successfully.

One of the problems with the XPath grammar is its *depth* —; the production for a simple integer has to go though 21 earlier productions to be reached and hence for many minor expressions the full parse tree is very large but with large sections that are very *thin*. One of the techniques to reduce these trees is to mark some rules such that if the element generated would only have one child, and no attributes, then ignore it in the serialization. Using a pragma mechanism to mark such rules, a simple parse of the grammar to its XML form, followed by an XSLT transform to change such rules (and references) and final serialization back to iXML can achieve this, making it a much more practical tool.

The next stage was to stress-test it by attempting to parse all the ~22,000 XPath expressions contained in the QT3 test-suite. A simple XSLT program was written and ran successfully, taking around 10 minutes to complete the task, finding initially a few hundred failures and then finally only 11. But in that process of refining the grammar against the samples, some possibly significant constraints to avoid extensive (and potentially exponential) ambiguity propagation have arisen, illustrating the limitations of the context-free grammar supported by iXML. These include:

- XPath defines some (binary) operators the character sequences of which *could* occur in a name (an `NCName`), such as `eq` or `div` and do *not* require to be surrounded by whitespace in situations in which they are unambiguously (according to XPath rules) acting as operators, such as `5div6` .

Such rules rely on having a tokenizer or lexical scanner upstream of the parser (where rules like “if it can be part of a name, make it part of the name” are easy to implement). Invisible XML, by contrast, does not require or assume a separate lexical scanner, and in writing an iXML grammar we must make whitespace before and after such operators either optional or required; we cannot, when writing a *context-free* grammar, make the choice depend on the context. If we make the whitespace around such operators optional, we are likely to encounter the ambiguity of “It this an operator or part of a name?”. The worst culprit is `-` (minus/hyphen) which whilst not a letter, can appear within `NCName`s and letting that have both roles typically causes exponential ambiguity growth with many large XPath expressions.

- To remove ambiguity, XPath also mandates some extra-grammatical constraints, which again are out of scope for iXML.

  Some names for function calls (`element`, `if`) are reserved as they conflict with node types or language construct keywords. Occurrence indicators (`?`,`*`,`+`) in possibly ambiguous cases have to bind to the closest `SequenceType` production rather than acts as possible arithmetic operators — there are a few other similar restrictions. Such extra-grammatical constraints are, in the nature of the case, outside the scope of an invisible XML grammar.

Another area requiring care (applicable to other grammars) is to ensure that optional whitespace doesn't get “double accounted”, such as being defined both in a non-terminal production and in the reference to it. But all these restrictions aside, this has demonstrated that large and complex grammars can be supported with iXML.

### *NineML*

NineML is a family of grammar parsing tools (and related projects). The number "Nine" in the name is a play on the "ix" of "ixml" reinterpreted as a Roman numeral; another illustration of Pemberton's (Pemberton 2016a) point about the numerical abstractions behind written numeral formats. Initial development of NineML has focused on a set of tools designed to run on the Java Virtual Machine (JVM). It currently consists of five related projects.

| | |
|---|---|
| **CoffeeGrinder** | A JVM API for building a grammar parser. |
| **CoffeeFilter** | A JVM API for processing Invisible XML documents. |
| **CoffeePot** | A command-line tool for parsing documents with Invisible XML grammars. |
| **CoffeeSacks** | A set of Saxon extension functions that make Invisible XML processing possible within XSLT or XQuery transformations. |
| **CoffeePress** | A set of XProc 3.0 steps that make Invisible XML processing possible with XProc pipelines. |

(It's Java-based. There are coffee puns. That's just how it is.)

## CoffeeGrinder

CoffeeGrinder is a Java API for building grammars and using those grammars to parse inputs. A grammar is constructed with API calls to a `SourceGrammar` object:

```
ParserOptions options = new ParserOptions();
SourceGrammar grammar = new SourceGrammar(options);

NonterminalSymbol S = grammar.getNonterminal("S");
NonterminalSymbol A = grammar.getNonterminal("A");
NonterminalSymbol B = grammar.getNonterminal("B");
TerminalSymbol a = new TerminalSymbol(TokenCharacter.get('a'));
TerminalSymbol b = new TerminalSymbol(TokenCharacter.get('b'));

grammar.addRule(S, A);
grammar.addRule(S, B);
grammar.addRule(A, a);
grammar.addRule(B, b);
```

This grammar is equivalent to the following Invisible XML grammar:

```
S = A | B.
A = 'a'.
B = 'b'.
```

At this level in the NineML stack, a `SourceGrammar` is a much simpler abstraction than an Invisible XML grammar. There are no built-in features for repetition or separators, for example, and alternatives have to be spelled out explicitly as the example shows for `S`. The grammar does support character classes and ranges, so it isn't necessary to spell out every possible matching terminal literally.

To parse an input, create a parser and call `parse()` on the input:

```
GearleyParser parser = grammar.getParser(options, S);

String input = "a";

GearleyResult result = parser.parse(input);

if (result.succeeded()) {
    System.err.printf("\"%s\" matches the grammar", input);
} else {
    System.err.printf("\"%s\" does not match the grammar", input);
}
```

The output from a parse is a “parse forest”. The parse forest is a data structure that represents all of the possible parses of the input with the grammar. This includes all of the ambiguous parses and even “infinitely ambiguous” parses in the case of grammars that contain loops.

There are a variety of APIs for walking the forest to extract one of the parses.

By default, CoffeeGrinder uses an Earley parser (Scott 2008). It is in the nature of Earley parsers that for some grammars and some inputs they make a very large number of predictions that are ultimately unused. This can cause memory and performance issues.

Recently, a GLL parser (Scott 2019) has also been added. (Credit to Dimitre Novatchev for suggesting that a GLL parser might have better performance.) Unlike an Earley parser which applies a particular “predict, scan, complete” algorithm to its input, the GLL parser begins by compiling the grammar into a “program” that is then executed to parse the input.

The GLL parser is orders of magnitude faster than the Earley parser for some inputs, but for most cases where the results are comparable, the GLL parser seems to be a little bit slower. The GLL parser does generally produce a smaller parse forest, but it is less well tested and has a few small issues with particular kinds of ambiguous grammars.

It’s worth pointing out that no effort has been made to optimize the GLL parser and there are several interesting avenues to explore, including the possibility that the grammar might be “compiled down” to Java bytecode so that the JVM’s “hotspot” compiler could optimize it.

## CoffeeFilter

Like CoffeeGrinder, CoffeeFilter is a JVM API. Typical use would look something like this:

```
ParserOptions options = new ParserOptions();
InvisibleXml ixml = new InvisibleXml(options);

String grammar = "S = A|B. A = 'a'. B = 'b'.";
InvisibleXmlParser parser = ixml.getParserFromIxml(grammar);

String input = "b";
InvisibleXmlDocument document = parser.parse(input);

String xml = document.getTree();

System.out.println(xml);
```

The `InvisibleXml` object provides a parser, this step transforms the Invisible XML grammar into a grammar that can be understood by `CoffeeGrinder`. This process introduces new rules and new nonterminals (along the lines suggested in the “Hints for Implementors” in the Invisible XML specification).

A parser can, in turn, be applied to an input, returning a document. Methods on the document allow you to retrieve one or more trees in a variety of ways.

## CoffeePot

CoffeePot is a command-line tool. It reads an Invisible XML grammar and an input and produces a serialized result. It will accept grammars in either text or XML form. It can be configured to cache parsed grammars which makes the process run a little faster. The input to be parsed can be provided directly on the command line or via a file or URI reference.

Consider the following small data file, `capitals.txt`, describing the latitude and longitude of some capital cities.

```
Dublin, Ireland: 53.35N 6.27W
Cardiff, Wales: 51.48N 3.18W
Edinburgh, Scotland: 55.95N 3.19W
Belfast, Northern Ireland: 54.61N 5.93W
```

You might write the following Invisible XML grammar, `capitals.ixml`, to parse it:

```
CityInfo = data++NL, NL*.
data = Capital, comma, Country, colon, Latitude, space, Longitude.
Capital = name.
Country = name.
Latitude = -north | -south.
Longitude = -east | -west.
north = decimal, -'N'.
south = +'-', decimal, -'S'.
east = decimal, -'E'.
west = +'-', decimal, -'W'.
-decimal = ['+' | '-']?, ['0'-'9']+, ('.', ['0'-'9']+)?.
-comma = -',', -' '*.
-colon = -':', -' '*.
-space = -' '+.
-sep = -',', -' '*.
-name = [L], ~[',' | #a]*.
-NL = -#a.
```

You can then convert the data to XML with CoffeePot:

```
$ coffeepot -g:capitals.ixml -i:capitals.txt --pretty-print --log:info
I: Parsing UTF-8 ixml grammar from
file:/Volumes/Projects/nineml/pot/scraps/capitals.txt
I: Parsing 163 tokens with Earley parser.
I: Parse succeeded, 163 tokens in 0.04s (4289.5 tokens/sec)
<CityInfo>
   <data>
      <Capital>Dublin</Capital>
      <Country>Ireland</Country>
      <Latitude>53.35</Latitude>
      <Longitude>-6.27</Longitude>
   </data>
   <data>
      <Capital>Cardiff</Capital>
      <Country>Wales</Country>
      <Latitude>51.48</Latitude>
      <Longitude>-3.18</Longitude>
   </data>
   <data>
      <Capital>Edinburgh</Capital>
      <Country>Scotland</Country>
      <Latitude>55.95</Latitude>
      <Longitude>-3.19</Longitude>
   </data>
   <data>
      <Capital>Belfast</Capital>
      <Country>Northern Ireland</Country>
      <Latitude>54.61</Latitude>
      <Longitude>-5.93</Longitude>
   </data>
</CityInfo>
```

Using `--pretty-print` makes the output easier to read; using `--log:info` tells us a little bit about the processing. Default values for both of these options (and many others) can be specified in a configuration file. In practice, you'd probably also use `-o:` (or `--output:`) to write the XML into a file.

Conformant behavior for an Invisible XML processor is to produce XML. In some environments, at least for simple data like this, JSON may be a more convenient format. CoffeePot will do that for you:

```
$ coffeepot -g:capitals.ixml -i:capitals.txt --format:json | jq .
{
  "CityInfo": {
    "data": [
      {
        "Capital": "Dublin",
        "Country": "Ireland",
        "Latitude": 53.35,
        "Longitude": -6.27
      },
      {
        "Capital": "Cardiff",
        "Country": "Wales",
        "Latitude": 51.48,
        "Longitude": -3.18
      },
      {
        "Capital": "Edinburgh",
        "Country": "Scotland",
        "Latitude": 55.95,
        "Longitude": -3.19
      },
      {
        "Capital": "Belfast",
        "Country": "Northern Ireland",
        "Latitude": 54.61,
        "Longitude": -5.93
      }
    ]
  }
}
```

In fact, this data is clearly tabular so perhaps CSV would be better:

```
$ coffeepot -g:capitals.ixml -i:capitals.txt --format:csv
"Capital","Country","Latitude","Longitude"
"Dublin","Ireland",53.35,-6.27
"Cardiff","Wales",51.48,-3.18
"Edinburgh","Scotland",55.95,-3.19
"Belfast","Northern Ireland",54.61,-5.93
```

Producing different output formats is intended to emphasis the fact that Invisible XML is about data abstractions. There are no plans to support any sort of arbitrary transformations of the output (move these columns around, for example). We have better tools for that.

## CoffeeSacks

CoffeeSacks provides a library of functions that you can call from XPath using Saxon. The functions allow you to parse a grammar, parse an input against the grammar, and manage the environment in which the functions operate. Basic usage is:

```
java -cp …:CoffeeSacks-ver.jar \
     net.sf.saxon.Transform \
     -init:org.nineml.coffeesacks.RegisterCoffeeSacks \
     your transformation options
```

The `RegisterCoffeeSacks` function makes the functions available from XPath. You can then call `cs:grammar()` to parse the grammar and `cs:parse-uri()` (or `cs:parse-string()`) to parse the input against the grammar.

Like CoffeePot, the default output is XML, but it is possible to persuade the parse functions to return a `map{}` analogous to JSON.

Here's an example stylesheet:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:cs="http://nineml.com/ns/coffeesacks"
                xmlns="https://nwalsh.com/ns/country-list"
                exclude-result-prefixes="#all"
                version="3.0">

<xsl:output method="xml" encoding="utf-8" indent="yes"/>
<xsl:strip-space elements="*"/>

<xsl:param name="ixml" select="resolve-uri('capitals.ixml', static-base-uri())"/>
<xsl:param name="txt" select="resolve-uri('capitals.txt', static-base-uri())"/>

<xsl:template name="xsl:initial-template">
  <xsl:variable name="grammar" select="cs:grammar($ixml)"/>
  <xsl:apply-templates select="cs:parse-uri($grammar, $txt)"/>
</xsl:template>

<xsl:template match="CityInfo">
  <CountryList>
    <xsl:apply-templates select="data">
```

```
      <xsl:sort select="Country"/>
    </xsl:apply-templates>
  </CountryList>
</xsl:template>

<xsl:template match="data">
  <Country>
    <xsl:apply-templates select="Country, *[not(self::Country)]"/>
  </Country>
</xsl:template>

<xsl:template match="Country">
  <Name>
    <xsl:apply-templates/>
  </Name>
</xsl:template>

<xsl:template match="*">
  <xsl:element name="{local-name(.)}" namespace="https://nwalsh.com/ns/country-
list">
    <xsl:apply-templates/>
  </xsl:element>
</xsl:template>

</xsl:stylesheet>
```

This will parse the same `capitals.txt` file with the Invisible XML grammar and produce a transformed result. A namespace has been added, the rows have been sorted, and some elements have been renamed:

```
<CountryList xmlns="https://nwalsh.com/ns/country-list">
   <Country>
      <Name>Ireland</Name>
      <Capital>Dublin</Capital>
      <Latitude>53.35</Latitude>
      <Longitude>-6.27</Longitude>
   </Country>
   <Country>
      <Name>Northern Ireland</Name>
      <Capital>Belfast</Capital>
      <Latitude>54.61</Latitude>
      <Longitude>-5.93</Longitude>
   </Country>
   <Country>
      <Name>Scotland</Name>
      <Capital>Edinburgh</Capital>
      <Latitude>55.95</Latitude>
      <Longitude>-3.19</Longitude>
   </Country>
   <Country>
      <Name>Wales</Name>
      <Capital>Cardiff</Capital>
      <Latitude>51.48</Latitude>
      <Longitude>-3.18</Longitude>
   </Country>
</CountryList>
```

### CoffeePress

CoffeePress is an extension step for XML Calabash 3.0 to process Invisible XML documents in XProc pipelines.

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
                xmlns:cx="http://xmlcalabash.com/ns/extensions"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                exclude-inline-prefixes="cx xs" version="3.0">

<p:output port="result"/>

<p:declare-step type="cx:invisible-xml">
     <p:input port="grammar" sequence="true" content-types="any"/>
     <p:input port="source" primary="true" content-types="any"/>
     <p:output port="result" content-types="any"/>
     <p:option name="parameters" as="map(xs:QName, item()*)?"/>
     <p:option name="fail-on-error" as="xs:boolean" select="true()"/>
</p:declare-step>

<cx:invisible-xml>
  <p:with-input port="grammar">
    <p:document href="capitals.ixml" content-type="text/plain"/>
  </p:with-input>
  <p:with-input port="source">
    <p:document href="capitals.txt" content-type="text/plain"/>
  </p:with-input>
</cx:invisible-xml>

<!-- more steps here -->

</p:declare-step>
```

Of the various libraries, it’s currently the least well developed.

## References

[Bahta et al., 2019] Bahta, Rahwa and Mustafa Atay. “Translating JSON Data into Relational Data Using Schema-oblivious Approaches”. doi:https://doi.org/10.1145/3299815.3314467. At ACM Southeast Conference – ACMSE 2019 – Session 2: Short Papers. Kennesaw, GA, USA. April 18-20, 2019. ISBN: 978-1-4503-6251-1.

[Bourhis et al., 2020] Bourhis, Pierre, et al. “JSON: Data model and query languages”. *Information Systems*. Volume 89 (2020). ISSN 0306-4379. doi:https://doi.org/10.1016/j.is.2019.101478.

[Dou et al., 2020] Dou, T., Kaszubowski Lopes, Y., Rockett, P. et al. “GPML: an XML-based standard for the interchange of genetic

programming trees”. *Genet Program Evolvable Mach* 21, 605–627 (2020). doi:https://doi.org/10.1007/s10710-019-09370-4.

[Grune/Jacobs 1990/2008] Grune, Dick, and Ceriel J. H. Jacobs. 1990/2008. *Parsing techniques: a practical guide*. First edition New York et al.: Ellis Horwood, 1990. Second edition [New York]: Springer, 2008.

[Guerts et al., 1990] Geurts, Leo et al. *The ABC Programmer's Handbook*. Prentice-Hall, 1990. ISBN: 0-13-000027-2. https://www.cwi.nl/~steven/abc/programmers/handbook.html

[Hillman 2020] Hillman, Tomos. “XSLT Earley: First Steps to a Declarative Parser Generator”. Presented at XML Prague, 2020, Prague, Czech Republic. In *XML Prague 2020 Conference Proceedings*, pp. 231-249 (2020). [online] https://archive.xmlprague.cz/2020/files/xmlprague-2020-proceedings.pdf#d6e8096

[Hopcroft/Ullman 1979] Hopcroft, John E., and Jeffrey D. Ullman. 1979. *Introduction to automata theory, languages, and computation*. Reading, Mass.: Addison-Wesley, 1979.

[Hillman et al., 2022] Hillman, Tomos, et al. “Pragmas in Invisible XML as an extensibility mechanism”. Presented at Balisage: The Markup Conference 2022, Washington, DC (Virtual Event), August 1–5, 2022. In *Proceedings of Balisage: The Markup Conference 2022*. Balisage Series on Markup Technologies, vol. 27 (2022). doi:https://doi.org/10.4242/BalisageVol27.Sperberg-McQueen01.

[Invisible XML CG, eds., 2022] Invisible XML CG, eds. *Sample Invisible XML Grammars*. https://github.com/invisibleXML/ixml/tree/master/samples

[Invisible XML CG, eds., 2022] Invisible XML CG, eds. *The Invisible XML Homepage* . https://invisiblexml.org/

[Lee et al., 2021] Lee, Junhee, et al. “SJSON: A succinct representation for JSON documents”. *Information Systems*. Volume 97 (2021). ISSN 0306-4379. doi:https://doi.org/10.1016/j.is.2020.101686.

[Lumley 2017a] Lumley, John, Debbie Lockett, and Michael Kay. “XPath 3.1 in the Browser”. In *Proceedings of XML Prague 2017*, pp. 1-18

(2017). [online] http://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf

[Pemberton 2013] Pemberton, Steven. “Invisible XML”. Presented at Balisage: The Markup Conference 2013, Montréal, Canada, August 6–9, 2013. In *Proceedings of Balisage: The Markup Conference 2013*. Balisage Series on Markup Technologies, vol. 10 (2013). doi:`https://doi.org/10.4242/BalisageVol10.Pemberton01`. On the web at http://www.balisage.net/Proceedings/vol10/html/Pemberton01/BalisageVol10-Pemberton01.html. Revised version (January 2014) at https://homepages.cwi.nl/~steven/Talks/2013/08-07-invisible-xml/invisible-xml-3.html

[Pemberton 2016a] Pemberton, Steven. “Data Just Wants to Be Format-Neutral”. Presented at XML Prague, 2016, Prague, Czech Republic. In *XML Prague 2016 Conference Proceedings*, pp. 109-20 (2016). [online] https://archive.xmlprague.cz/2016/files/xmlprague-2016-proceedings.pdf#d6e2656. On the web at https://homepages.cwi.nl/~steven/Talks/2016/02-12-prague/data.html

[Pemberton 2016b] Pemberton, Steven. “Parse Earley, Parse Often: How to Parse Anything to XML”. Presented at XML London, 2016, University College London, London, UK, 4-5 June 2016. In *XML London 2016 Conference Proceedings*, pp. 120-126 (2016) [online] https://xmllondon.com/2016/xmllondon-2016-proceedings.pdf#page=120. On the web at https://homepages.cwi.nl/~steven/Talks/2016/06-05-london/xml-london.html

[Pemberton 2017] Pemberton, Steven. “On the Descriptions of Data: The Usability of Notations”. Presented at XML Prague, 2017, Prague, Czech Republic. In *XML Prague 2017 Conference Proceedings*, pp. 143-159. [online] https://archive.xmlprague.cz/2017/files/xmlprague-2017-proceedings.pdf#page=155

[Pemberton 2022] Pemberton, Steven. “Invisible XML Specification”. Published by the Invisible Markup Community Group on the web at https://invisiblexml.org/1.0/

[Pemberton 2022b] Pemberton, Steven. “A Pilot Implementation of ixml”. In *Proc. XML Prague 2022*. 2022. Pgs 41-50. [online] https://archive.xmlprague.cz/2022/files/xmlprague-2022-proceedings.pdf#page=51

[Pemberton 2022c] Pemberton, Steven. *Hands On iXML*. CWI. 2022. http://www.cwi.nl/~steven/ixml/tutorial/

[Scott 2019] Scott, Elizabeth, Adrian Johnstone, and L. Thomas Van Binsbergen. “Derivation representation using binary subtree sets”. *Science of Computer Programming* 175, 63-84 (2019). doi:`https://doi.org/10.1016/j.scico.2019.01.008`.

[Scott 2008] Scott, Elizabeth. “SPPF-Style Parsing From Earley Recognizers”. *Electronic Notes in Theoretical Computer Science* 203, 53-67 (2008). doi:`https://doi.org/10.1016/j.entcs.2008.03.044`.

[Shatnawi et al., 2021] Shatnawi, Hazim and H. Conrad Cunningham. “Encoding Feature Models Using Mainstream JSON Technologies”. doi:`https://doi.org/10.1145/3409334.3452048`. At *ACM Southeast Conference – ACMSE 2021 – Session 1: Full Papers*. Virtual Event, USA. April 15-17, 2021. ISBN: 978-1-4503-8068-3.

[Sperberg-McQueen 2019] Sperberg-McQueen, C. M. “Aparecium: An XQuery / XSLT library for invisible XML”. Presented at Balisage: The Markup Conference 2019, Washington, DC, July 30 – August 2, 2019. In *Proceedings of Balisage: The Markup Conference 2019*. Balisage Series on Markup Technologies, vol. 23 (2019). doi:`https://doi.org/10.4242/BalisageVol23.Sperberg-McQueen01`.

[Tovey-Walsh 2022a] Tovey-Walsh, Norm. “Invisible XML”. On the web at https://www.xml.com/articles/2022/03/01/invisible-xml/

[Tovey-Walsh 2022b] Tovey-Walsh, Norm. “Writing Invisible XML Grammars”. On the web at https://www.xml.com/articles/2022/03/28/writing-invisible-xml-grammars/

[Walsh et al., 2017] Walsh, Norman, et al. (eds). *XQuery and XPath Data Model 3.1*. https://www.w3.org/TR/xpath-datamodel/

[1] The technical background may be summarized briefly.

In the case of grammars in Backus / Naur Form (BNF) or similar notations, formal language theory defines ambiguity as the existence of more than one leftmost (or rightmost) derivation of a sentence; sometimes it is defined as the existence of more than one production tree, which amounts to the same thing. Extended BNF notations like the one used by Invisible XML generate the same set of languages as BNF grammars, but derivation works differently, and the authorities we have consulted define neither derivation nor ambiguity with respect to EBNF grammars.

Note also that ambiguity is a property of a sentence parsed with a given grammar; the same sentence might be unambiguous with respect to a different grammar for the same language. Some iXML processors work by translating the input grammar into an equivalent BNF grammar and then using algorithms defined for BNF grammars to parse the input. In many cases, more than one BNF grammar will be equivalent to the input iXML grammar; a sentence may be ambiguous against one of those grammars but not against another.

We do not wish to require a particular translation from EBNF to an equivalent BNF, and so we face the situation that some implementations will encounter ambiguity in the raw parse trees of a sentence where others do not. We have therefore found ourselves obliged to allow some variation in results among conforming processors as regards the detection of ambiguity.

For a formal definition of ambiguity, see Hopcroft/Ullman 1979 section 4.3, or Grune/Jacobs 1990/2008 section 3.1.2. Of particular interest here is the discussion of extended notations for context-free grammars in section 2.3.2.4 of Grune/Jacobs 1990/2008.

[2] The example assumes an ixml grammar like the one for parsing vcards which was posted earlier this year on the xsl-list mailing list.

---

## Invisible XML coming into focus

### *Status report from the community group*


***Tomos Hillman***

eXpertML Ltd

<tom@expertml.com>


Tom Hillman has worked as an XML practitioner and consultant for fifteen years, doing everything from documentation to IT support and administration to workflows for digital publishing to conference organization to XML database management and consultancy.


***John Lumley***

<john@johnlumley.net>


A Cambridge engineer by background, John Lumley created the AI group at Cambridge Consultants in the early 1980s and then joined HPLabs Bristol as one of its founding members. He worked there for 25 years, managing and contributing in a variety of software/systems fields, latterly specialising in XSLT-based document engineering, in which he subsequently gained a PhD in early retirement. Rarely happier than when writing XSLT to write XSLT to write XSLT, he spent the next several years helping develop the Saxon XSLT processor for Saxonica, including developing the XSLT-based XSLT compiler now used in SaxonJS. Now in proper retirement for a couple of years he still likes to 'potter' with XSLT and is currently working on a JavaScript-based processor for Invisible XML to attach to SaxonJS.


***Steven Pemberton***

Centrum Wiskunde & Informatica (CWI)


Steven Pemberton is a researcher, author, public speaker, and occasional broadcaster, affiliated with the CWI, The Dutch National Research Centre for Mathematics and Informatics. His research is broadly in interaction, and how the underlying software architecture can better

support users. He is the chair of the W3C Community Group on Invisible XML.


***C. M. Sperberg-McQueen***
Black Mesa Technologies LLC
<cmsmcq@blackmesatech.com>


C. M. Sperberg-McQueen is the founder of Black Mesa Technologies LLC, a consultancy specializing in the use of descriptive markup to help memory institutions preserve cultural heritage information. He co-edited the XML 1.0 specification, the Guidelines of the Text Encoding Initiative, and the XML Schema Definition Language (XSDL) 1.1 specification.


***Bethan Tovey-Walsh***
Swansea University
<bytheway@linguacelta.com>


Bethan Tovey-Walsh is a PhD student in Applied Linguistics and Welsh at Swansea University. She is funded by the CorCenCC corpus of modern Welsh, and created the Welsh part-of-speech tagger now used by the project. She previously worked for OUP as a content architect and as a researcher for the Oxford English Dictionary.


***Norm Tovey-Walsh***
Senior Software Developer
Saxonica
<ndw@nwalsh.com>


Norm Tovey-Walsh is currently a senior software developer at Saxonica Ltd, working from his home in Swansea, Wales. Previously, he was employed by MarkLogic Corporation, Sun Microsystems, Arbortext, and O’Reilly Media (then O’Reilly & Associates).

---