



## JAY PARSER:

An Invisible XML implementation in XSLT

Good Morning, Everyone!

A dark blue square with a thin white border. Inside the square, the text 'TOMOS HILLMAN' is written in large, bold, white capital letters. Below the name, the email address 'tom@eXpertML.com' is written in a smaller, blue font.

**TOMOS  
HILLMAN**  
tom@eXpertML.com

My name is Tom Hillman. I am a principle consultant with eXpertML Ltd, where I spend most of my time working with XSLT for the publishing industry, or giving XML technology training courses. Today I'd like to spend some time talking about a personal project: an open source invisible XML implementation, based on ideas from the Earley parser, and written in XSLT.





I call it the ‘Jay’ parser (a nod to Jay Earley);

In February at XML Prague, I gave a talk on the proof of concept, but I ran out of time before I got to talk about any of the code details: I hope to rectify that today, and share some of the progress that has (or hasn’t) been made since then!



## THE EARLEY PARSER

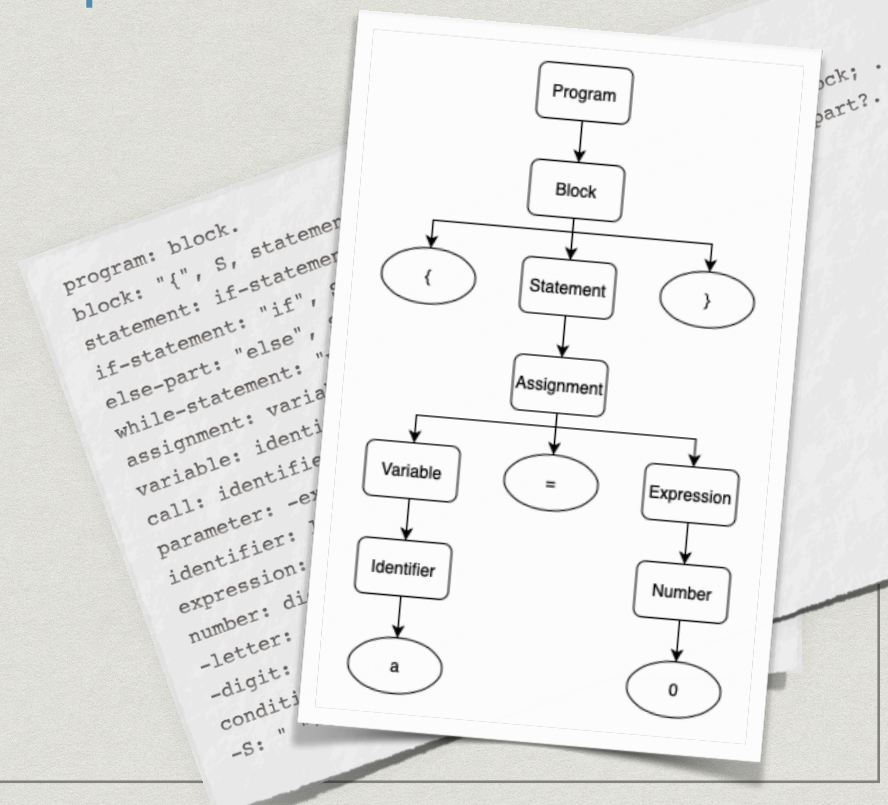
[A brief overview](#)

However, before I telling you about the code, let's just cover some parser basics, and see if I can explain a little about the Earley parsing mechanism.



# What is a parser?

- \* Takes a string
- \* And a grammar
- \* Returns a tree

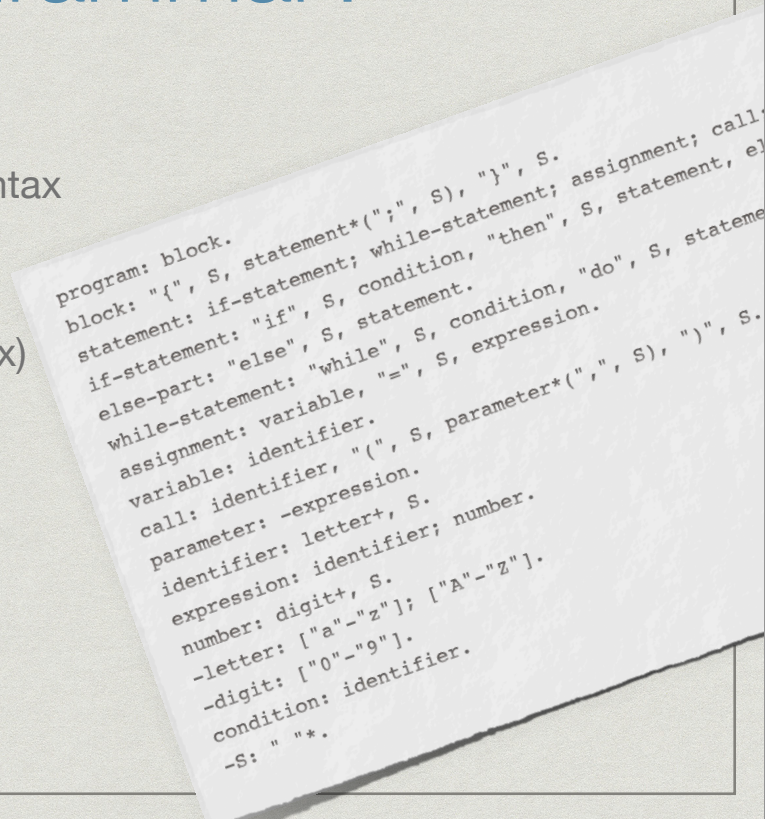


In this case, when I am talking about a parser, I am talking about a program or algorithm that takes two inputs: a string, which we want to parse, and a grammar, which tells us the parsing rules.

A parser takes these things, and attempts to 'understand' the string according to the grammar: in our case it does that by returning the information in the string as a marked-up tree structure.

# What is a Grammar?

- \* Defines a language's syntax
- \* Symbols
  - \* Terminal (strings, regex)
  - \* Nonterminals
- \* Repetition
- \* Optionality



```
program: block.  
block: "{", S, statement*(";", S), "}", S.  
statement: if-statement; while-statement; assignment; call.  
if-statement: "if", S, condition, "then", S, statement, el  
else-part: "else", S, statement.  
while-statement: "while", S, condition, "do", S, stateme  
assignment: variable, "=", S, expression.  
variable: identifier.  
call: identifier, "(", S, parameter*(",", S), ")", S.  
parameter: -expression.  
identifier: letter+, S.  
expression: identifier; number.  
number: digit+, S.  
-letter: ["a"-"z"]; ["A"-"Z"].  
-digit: ["0"-"9"].  
condition: identifier.  
-S: " ".*
```

The grammar is the key to recognising the information in a string conforming to the syntax of a language that grammar defines:

Invisible XML belongs to the EBNF family of grammars, and defines syntax using symbols: symbols can either be 'terminals' - that is they represent literal text, or match patterns in the input - or 'nonterminals' - rules which are themselves made up of symbols.

Grammars may also define repetition, and optionality of these symbols, in much the same way as we are probably all familiar with from regular expression tokens, DTDs or schema.

Note in this example how each line here represents a grammar rule: each rule starts with a non-terminal name, followed by a colon, followed by a sequence or combination of terminal and non-terminal symbols.



# Invisible XML

- \* Invented/discovered by Steven Pemberton in 2013
- \* All data is an abstraction
- \* So expressions/representations can be thought of as equivalent
- \* If it can be parsed with a grammar, we can treat it like XML

Invisible XML was invented by Declarative Amsterdam's own Steven Pemberton in 2013; since all data is an abstraction, he argued that any representation of the same data could be thought of as equivalent, regardless of how it's serialised.

In other words, if it can be parsed with a grammar, we can treat it as XML.

The core of Invisible XML, then, is a grammar language that includes features to control how a parser serialises an input to XML, by converting selected nonterminal grammar rules into elements and attributes, and terminal values into text nodes and attribute values.



# Earley Algorithm (briefly!)

Some useful terms:

- \* **State** tells you what you have left to parse
- \* **Rules** map one **symbol** to a sequence of **terminal** and **nonterminal** symbols

Next it is helpful to consider the Earley algorithm: Here are some useful terms.

The State (or state number) is a reference which stores the input that has yet to be parsed.

Rules map one nonterminal symbol to a sequence of terminal and non-terminal symbols (just like we say earlier in the grammar example).



# Earley Algorithm (briefly!)

Starting with the first rule of the grammar, build a table of “Earley Items” storing:

- \* The current rule
- \* The position in the current rule
- \* The current state
- \* The state when the rule evaluation began

Traditionally, the Earley parser starts by evaluating the first rule of the grammar, and uses it and the symbols it references to build a table of “Earley items”.

This table stores the current rule being tested against the input,

The position in the current rule as the rule is being processed,

The current state,

And the state when the rule evaluation began.

These can be thought of as a series of atomic ‘partial parses’

We can also talk about items which ‘complete’ a rule - that is, where we have reached the end of a rule definition.



# Earley Algorithm (briefly!)

Once we have an item in the table that

- \* completes a rule that
- \* starts in the initial state
- \* (and ends in the end state)

we have a valid (complete) parse.

Eventually the table should contain an item that completes a rule, which starts at the initial state, and ends at the end state - and this gives us a valid parse.

Actually, there may be more than one valid parse result, and it should be possible to generate every one of them from the complete table of Earley Items.





**SHALL WE TALK ABOUT CODE?**

So, let's talk a bit about the actual code.

# EARLEY TREES

Earley tables and Earley items sound well and good, but one of my goals for the JayParser was to write a parser that could be extended by XSLT developers - for instance to dynamically expand entity substitution.

Since what we want out of the process is a tree, I wanted to find a way to make the most of XSLT's template matching and tree handling, so I began to explore the idea of making a tree instead of a table, an idea inspired by Michael Sperberg-McQueens paper on the subject at balisage 2017.



# Earley Trees

- \* **Rules** are stored as containing elements; children are one or more alternatives
- \* Position in the rule is position in the tree
- \* **State** is stored as a sequence of strings as a parameter
- \* Infinite recursion is avoided by holding a map of **visited** rules in particular states
- \* **Terminals** are held as 'leaf' nodes
- \* **Visited Nonterminals** are held as 'leaf' node 'links'

In an Earley Tree, rules are stored as elements, which can contain one or more alternative parses, themselves containing further trees as representations of the symbols of that rule.

The position within the rule, is simply the document position within the tree.

State is stored as an integer, which refers to the index of a sequence of all possible states - there are other, more efficient ways of doing this, but remember that I hoped one day to do things like entity substitution, so I needed a way of being able to change the input string as it is parsed!

To eliminate duplication of effort (and risk of infinite recursion), we also need to store a map of visited rules and states.

Terminals are held as 'leaf' nodes, and visited non-terminals are recorded as references, so that they can be reconstructed later.



# Pruning the Earley Tree

- \* Exclude elements that fail or are empty
- \* Copy/expand nonterminal references
- \* Choose from alternatives & remove unnecessary structural elements

It turns out that the Earley Tree that gets created is what is known in parsing circles as a type of ‘parse forest’ - a set of possible parses, valid or invalid, combined using ‘AND’ and/or ‘OR’ relationships.

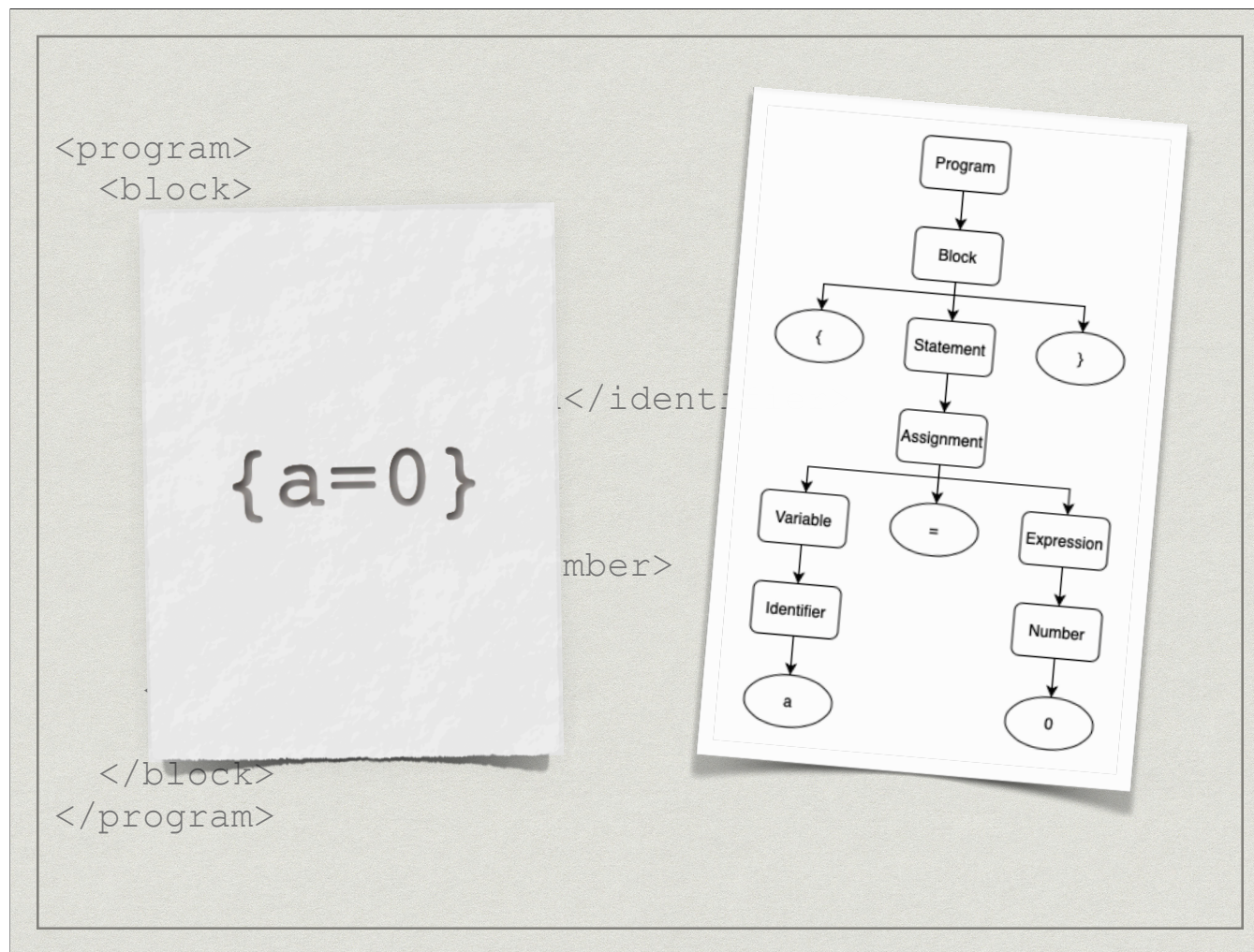
All we need to do to it to return a valid parse is prune away all of the invalid parses, and choose any one from any valid parses.

We can do that by excluding empty elements, or any non-terminals that fail to complete.

visited non-terminal references can be copied in order to expand them,

and where there are alternatives, we can simply choose one - in our case, we simply take the first - and then remove the now unnecessary structural elements that present the choice.





Here's the proof of concept example from February this year: a very simple input file, and the resulting XML representation.

# Transforming the Grammar

- \* Invisible XML defines its own grammar
- \* Therefore there is an XML representation
- \* JayParser works by transforming the XML representation of a given grammar into an Earley Tree, using the input string as a supplied parameter.

The two inputs to our parser are normally presented as strings:

The string to be parsed, which we want our parser to represent as a tree, and the grammar itself.

But the Invisible XML Spec defines its own grammar;

This means that the grammar can also be represented as an XML tree.

It seems to me that XSLT's strongest use case is when transforming trees, especially tree-to-tree transformations.

So it makes sense that the JayParser uses this XML representation as the tree input in a tree to tree transformation, processing it with the input as a parameter, and producing then pruning the resulting Earley Tree.

Let's take a look at a couple of structures that we use to do so



# The Visited Map

- \* A manifest of which rules are visited in which states
- \* Also records which end states are given by the visit
- \* Needs to be updated every time a new rule (or set of alternatives) is visited

As the Earley tree is created, we need to know whether each rule or set of alternatives being processed have already been processed in the current state.

If a rule has already been visited, we will also need to know which states might result, so that we know where to continue processing.

This is stored as the 'visited' map which can be passed as a tunnel parameter through the process.

This map not only needs to be checked each time a rule is visited, but needs to be added if the rule has not been encountered, or updated if it results in a complete partial parse.



# The States sequence

- \* Any given state can be represented as an integer
- \* That integer represents a stored string
- \* The list of stored strings need to be updated every time a terminal is encountered

We also have the idea of the States sequence

A state in JayParser is a number, an integer, which itself represents a stored string, which I shall call the state string. Specifically, it is the index of that stored string in a sequence of all possible state strings.

This sequence of state strings also needs to be updated as the Earley Tree is created, with a new state string being added every time a terminal symbol correctly matches the current state string.



# CONTROLLING PROCESS ORDER

This need to update the visited map and the States sequence before processing the next node at any given point is probably the biggest challenge in writing the Jay Parser.

We can't simply process all of the children of any given node, because each node depends on the values of both of these variables that result from processing the previous node (in document order).



# Controlling Process Order

- \* The **visited** and **state** information needs to be passed:
  - \* from sibling to following sibling
  - \* from children to parent
  - \* NOT from parents to children
- \* We need to override the default processing order (preferably preserving the pull processing idea)

This means that the visited and state information needs to be passed from sibling to following sibling, and from children to the parent.

tunnel parameters will only get you so far: they pass information from parents to children

This means that we need to override the default XSLT processing order, being careful to preserve some sense of the pull processing idea!



# Controlling Process Order

- \* Use named template instead of applying templates
- \* Process children as a variable before forming the parent
- \* Calculate parent values separately (custom functions & modes)

When working on the XML Prague proof of concept, I decided to use a couple of named templates wherever I would normally have applied templates.

These processed children nodes as a variable before returning a parent node - sort of processing the tree from the leaves back to the trunk.

I then calculated visited and state information separately for each node, using custom functions and modes.

I wasn't very happy with this at the time - it seemed inefficient as you were essentially processing the same node multiple times, and it turned out that there were more fundamental reasons not to use this approach: it is hard to ensure that building these data structures in this way actually produces consistent values for siblings in the grammar input.



# Map approach

- \* When building the Earley Tree, templates return maps including:
  - \* Earley tree of each child
  - \* States definition
  - \* Visited map
- \* Still need some named templates to process siblings, but now uses `xsl:iterate`

The proof of concept for XML Prague worked for one given grammar, but not yet in the general case. Additionally, it did not make enough use of some of the new XSLT 3 features that are now available to us.

So I decided to rewrite the parser; in the mode that builds the Earley tree for all but the outermost template, I decided to return not just the Earley Tree, but a map.

This map includes the Earley tree, but also the states definition and the visited map. Templates can combine maps intelligently, retaining updated visited and state information, and combining the Early tree leaves and 'saplings'. This ensures each node need only be visited once, and information remains consistent.

We still need to use a named template approach, but rewriting these to use `xsl:iterate` resulted in much more readable understandable code than the previous mechanism which used recursive `apply-templates` instructions.



# OPTIONALITY & REPETITION

The Invisible XML specification suggests an approach for implementing options and repetition: since the grammar language already has structures to deal with alternatives and “empty” results from parses, we can rewrite optional symbols, and symbols that repeat.



# Rewriting repeat0

```
<xsl:template match="repeat0" mode="e:parseTree"
as="map(*)">
  <xsl:variable name="GID" select="(@gid, local-
name()||generate-id())[1]"/>
  <xsl:variable name="equivalent" as="element(alts)">
    <alts>
      <alt>
        <empty/>
      </alt>
      <alt>
        <repeat1 gid="{ $GID }">
          <xsl:copy-of select="@*, node()" />
        </repeat1>
      </alt>
    </alts>
  </xsl:variable>
  <xsl:apply-templates select="$equivalent"
mode="#current"/>
</xsl:template>
```

Here's an example which rewrites an optional repeat, "Repeat0"

Note that this template, together with almost all templates of this mode, returns the map containing not only the Earley Tree, but also visited and state metadata.

First of all we create a unique ID for the repetition, using the local name and the generate-id() function. We can use this to add the alternatives structure to the visited map, which should ensure we don't get stuck in an infinite loop.

Next we build an alternative structure, using the same model as the input grammar. An optional repeat is really equivalent to the two alternatives of an empty parse, or else a non-optional repeat.

Since this returns a single element, we can simply apply-templates in the parse tree mode, which will also return a map.

Next, we should take a look at the non-optional repeat.



# Rewriting repeat1

```
<xsl:template match="repeat1" mode="e:parseTree" as="map(*)">
  <xsl:variable name="GID" select="(@gid, local-name() ||
generate-id())[1]"/>
  <xsl:variable name="equivalent" as="element(*)">
    <xsl:sequence select="child::*[not(self::sep)]"/>
    <alts gid="{ $GID }">
      <alt>
        <empty/>
      </alt>
      <alt>
        <xsl:sequence select="sep"/>
        <xsl:copy>
          <xsl:attribute name="gid" select="$GID"/>
          <xsl:copy-of select="@*, node()"/>
        </xsl:copy>
      </alt>
    </alts>
  </xsl:variable>
  <xsl:call-template name="e:process-siblings">
    <xsl:with-param name="siblings" select="$equivalent"/>
  </xsl:call-template>
</xsl:template>
```

The non-optional repeat template should look fairly similar! Again, we are looking to return a map, and again, we are generating an ID for use with the visited map.

Actually, we saw in the optional repeat example previously that we specified this ID as an attribute; we need to preserve this value as we rewrite parts of this tree, since every time we create one of these rewrite ‘equivalent’ variables, they are technically a new document, and will have a different value of generate-id().

A non-optional repetition is equivalent to the content of whatever is being repeated, followed by an optional set of any defined separator together with a copy of the original repeat.

We can also deal with optional symbols in a similar way.





## RESULTS

So much for the theory, how does it work?



# Implementing iXML

- \* Invisible XML specification is still in beta
- \* Community group is forming
- \* Test suite planned

Well, before we get to that, I'd like to spend a little time discussing what it means to be an implementation of invisible XML

In a sense, it's impossible to say that the JayParser is an Invisible XML implementation, because the Invisible XML Specification is still in beta, and there is no conformance suite defined, as yet.

The good news is that a working group has been formed since Balisage this year, and Steven Pemberton, Michael Sperberg-McQueen and myself have been meeting monthly, and that we are looking at defining and publishing a conformance test suite. Of course, we would welcome involvement from the wider community.

My primary method of testing is a modified version of the 'dog-food' test: parsing the invisible grammar itself (slightly tweaked to ensure that there is an example of every rule represented).



# Mixed Bag!

- \* Proof of concept still works...
- \* 90% implementation of iXML (estimate)
- \* Performance and scaling issues

In terms of progress, it's not as good news as I would like!

My proof of concept still works after the rewrite, which is a start!

My goal of implementing Invisible XML is still a way off - the new code attempts to implement all features, but some grammars (such as the dogfood test) still get stuck in infinite loops - so there are some bugs that still need to be squashed.

My sense is that a working parser is actually very close, but unfortunately I had run out of time before this conference.

The biggest problem at the moment, though, seems to be with the performance and scaling: parsing the invisible XML Grammar itself currently takes nearly an hour before failing; there is still plenty of work to do, which leads nicely on to:



**CONTRIBUTING**



# JayParser on github

- \* <https://github.com/eXpertML/JayParser>
- \* Contributions welcome!



If anyone would like to have a look at the code itself, or wishes to contribute to the project, it is available on github





Thanks for listening to my talk - my apologies for anyone hoping to hear of a complete and working parser implementation, all I can say is that sometimes success is merely iterative failure, and I'm not giving up just yet! :)