Balisage: The Markup Conference

CodeUp: Marking up Programming Languages and the winding road to an XML Syntax.

David Lee
Lead Engineer
MarkLogic, Inc.
<dlee@marklogic.com>

Abstract

Text markup has a rich history and mature tools and techniques. Programming languages however are rarely marked up except to provide embedded documentation. In this paper I compare the properties of Text and Programming Languages and suggest ways of applying the lessons and tool-chains from the Text Markup tradition to Programming Languages, evaluating the value verses the cost and suggest alternative approaches to achieving some of the value gained using Text Markup in a Programming Language context.

Table of Contents

Why is text marked up?

In any field the experts can get so good at what they do that the original purpose can get lost, or they assume everyone has the same understanding. It can be useful to step back and ask 'What are we doing?', 'Why are we doing it?', "Who is the audience for our work?", "Are our tools and techniques accomplishing our goals?".

I started this navel pondering last year and asked first myself, then informally the professional groups "Why is text marked up?" "Why not just add to the original text?". Immediately these simple questions exposed some assumptions that need clarification before answers could be given.

What is 'Text'

For the purposes of this discussion I will define 'Text' as the 'Written representation of natural languages'. This is a narrow definition, but it is what we tend to mean when talking about 'Marking up Text'. Furthermore I am jumping forward a few centuries and narrow the meaning further to mean 'The machine readable digital encoding of written representation of natural human languages' aka Bytes, Characters, Codepoints, Words, Sentences, Paragraphs in a computer representation.

So now that we know what 'Text' is ... what does that imply ? Humans don't natively encode their thoughts and language directly (aka telepathically) into computers. They speak or write or type onto some media which is then encoded or transcribed into a computer. That very process changes the original and loses information. In speech, intonation and body expressions are lost when transcribed. In writing the media and physical representations (style, color, size, and emphasis) is lost when encoded to computer media. Typing via a keyboard generally limits the representation to uniform characters. Even before computers this was so natural to humans that the expression of language can affect it's meaning that different language is used when speaking and writing. Written (and printed) language contains information with the intent of adding back things missing from the spoken word in the form of style, font, size, spacing, as well usually a clearer and more precise use of language.

Unlike interactive communication, the nature of Text (and recorded speech) extracts itself from time and space. Text is read, it is usually at a different moment in time and a different place in space from when it was authored. Since language requires a shared understanding, much is inherently lost in text. The reader has indirect access at best to the environment or context of the original event. Some of this context is recreated in the text itself through dialog, prose, notes and contextual references, but requires the author to imagine what context may be lost and provide a substitute. This is one reason why ancient texts are more difficult to interpret than modern texts; the author was unable to predict and accommodate for the greater difference in context of the modern reader.

What is the purpose of Text ?

The purpose of Text, as a representation of natural language, is to express and ideally communicate thought. Expressing thought implies an Author. Communicating thought implies an Audience. We were taught to "Consider the audience" when writing, but that is a difficult and often impossible goal. At best we may consider the Intended Audience, but who is to say who may want to read our text in the future and where or when that might be? Since we can't embed our entire context in every writing there is always an assumption as to the shared understanding of the intended audience.

Why mark up text?

Why do we mark up text? Especially digital text and when the markup itself is text, why not just add to the text? Marking up text is essentially just adding more text. But it's a different sort, it is 'out of band'. It represents a different dimension or layer from the text itself. Why do we do this ?

I asked this question to several groups in the hopes of getting opinions I had not previously considered, or at least to clarify my preconceptions. As expected the answers varied in detail greatly by the person. Obviously (in hindsight) people mark up text for different specific purposes and express these in different degrees of abstraction.

Some example responses indicating a range of reasoning, edited and out of context for brevity:

> ... [so] that you could have a single document (with markup) and derive from it a variety of secondary views that were very different.
>
> — Dave Patterson on LinkedIn Groups XML and Related Technologies Network XMLLinkedIn1.

> Because it doesn't tell the whole story. There may be important contextual information that could completely change the meaning of the narrative.
>
> — Paul Monk on LinkedIn Groups XML and Related Technologies Network XMLLinkedIn2.

> ... it separates the "meta" information that was human readable in the style into a format that is machine readable ...as a visual encoding format that is human readable.
>
> — Sean Rushing on LinkedIn Groups XML and Related Technologies Network XMLLinkedIn3.

> ... Annotate things, ...add meta-data to a document. And presentation ...
>
> — Oscar Vives on xml-dev@lists.xml.org XMLDev1.

> ... for the purpose of extending meaning, suggesting presentation or communicating actions
>
> — Philip Fennell on xml-dev@lists.xml.org XMLDev2.

> Mark-up in retail increases the selling price. Mark-up in texts increases the value.
>
> — Liam Quin on xml-dev@lists.xml.orgXMLDev3.

> To make explicit for mechanical processing what is implicit-but-evident in the original.
>
> — Henry S. Thompson on xml-dev@lists.xml.orgXMLDev4.

> ... there was an Original Text that was Divinely Inspired by the Goddesses, and that could not be improved upon by mortals; the task of the humans was to understand these sacred texts, so that a commentary had to distinguish clearly the original and the expansion.
>
> — Liam Quin on xml-dev@lists.xml.org XMLDev5.

Distilling these responses with my own reasoning I propose the following concepts for why we mark up text. The reasons vary by author, context, and purpose. Some reasons are complementary and some are contradictory. Yet still we find markup useful.

- Preserve the original text while adding additional information
- Annotate for purposes of presentation
- Annotate for purposes of action
- Annotate to provide 'layers' of information
- Provide for machine readable processing
- Allow for production of multiple products from the same source (for different media, form factor or audience)
- To add information (context) implicit in the original form
- To encode text so that it may be semantically analyzed

It is relevant to note that many of these reasons only require markup because software to date is not smart enough to actually parse and understand natural languages well enough. With human consumers, the semantics of text representing "Title", "Paragraph", "sentence" "emphasis" are usually quite obvious to the human reader. So are generally the more linguistic attributes such as verb, noun, and phonetics. Even the context is often known implicitly by the reader, say when reading a book by Charles Dickens one may have a reasonably accurate internal model of the Victorian period by which to frame the experience.

Thus much of the value of markup is to encode these things which the primary consumer (humans) already knows to make it easier for computer processing.

Why markup Programming Languages?

As with Text, consider Programming Languages. Why would we want to markup programming languages? If there is missing information why not just add more to the program? Isn't a Programming language already marked up? Similar to Text, these simple questions pose some implications and assumptions.

What is a Programming Language?

At first glance, Programming Languages seem an altogether different beast than natural Languages. The language is artificial and designed for a single purpose - to instruct a computer on how to accomplish a task. All the information necessary for this task is already encoded in the language - so what need is there for markup?

The hidden assumption in these questions is that Programming Languages (or rather it's textual representation - "The Source Code") have a single target audience - the compiler. But this is blatantly untrue. The truth is that Programming Languages not only have multiple audiences for multiple purposes but also usually multiple authors. So what really is a Programming Language?

An instance of Programming Language is expressed as one or more text documents which comprise "The Source Code". These documents are encoded typically in 'plain text' with

a limited range of characters available. The syntax of the programming language is intended to be humanly readable and writable as well as readable by a computer.

What is the purpose of a Programming Language?

A programming language serves multiple purposes. The obvious purpose is to instruct a computer to perform a task. But there are other purposes. A programming language is intended to be written and understood by humans. Similar to a text document, there is an implicit context in which the language resides which may not be necessary for the compiler, but is necessary for humans in authoring, editing, reading and understanding the program.

Different languages recognize and support these purposes to different extents by allowing types of markup as part of the language itself. Comments are the main form, but also common is markup which may be recognized by the compiler or ancillary programs such as pragmas, annotations, and micro-markup within comments such as JavaDoc JavaDoc, HTML HTML etc. The syntax of comments and micro-markup is specific to each language and generally not interoperable. To make things more difficult, languages are often embedded within other languages within the same document. For example a Java JavaScript program which includes HTML that has JavaScript JavaScript , or C JavaScript Program with embedded SQL JavaScript . The syntax rules for what constitutes a comment and where it can be placed can get extremely confusing. Furthermore, most markup defined in programming languages is a point, not a container; e.g. a comment occurs at some point in the text stream but does not explicitly identify it's target (if any) that is to what does the comment refer. It is up to humans and convention to decipher the target, scope, and intent.

Why markup programming languages?

Programming languages need to be marked up for generally the same reason as Text. Source code is a human readable text document that has many of the same characteristics and needs as text for human languages. Reasons for marking up Programming Languages include:

- Preserve the original machine readable program while adding additional information for human readers.
- Annotate for purposes of presentation ("pretty printing")
- Annotate for purposes of action ("TODO, FIXME ...")
- Annotate to provide 'layers' of information (Change history, rationale, alternative implementations)
- Provide for machine readable processing besides compilation (document generation, reassembly of sections, flow analysis, code databases and searching)
- Allow for production of multiple products from the same source (documentation, program listings, graphs, overviews ...)
- To add information (context) implicit in the original form (explanation of algorithms, attributions, clarifications)

Programming languages have some pre-existing mechanisms for markup which may suffice for some of these needs. Are these sufficient? Is there benefit in learning from the concepts, tools, and techniques of Text markup?

A very significant difference between computer languages and Text is that computer processes can already (and must) explicitly extract the semantic meaning of most language constructs. The main exception being comments which suffer from the same problems as Text, they are not currently readily parsable by computers.

The problem however, is that even though "The Computer" can parse the semantics of a programming language, it greedily hides this information in the bowels of it's internal data structure giving no heed to the concept that maybe there might be additional consumers who could value from this information. Thus much of this information for which Text Markup is required is only needed because Programming Language consumers (parsers, compilers etc.) are not designed for any other purpose.

Practical uses for Programming Language Markup

In the previous section we discussed some fairly abstract reasons for marking up software. In practical terms what might this buy us? What kinds of tools and value might we expect if the full semantics of programming languages was available to existing markup processing tool-sets?

Some concrete examples might include:

- Automated reorganization of source files by reliable detection of boundary markers in compilation units
- Call-flow and dependency analysis
- Data Mining and document management of a large code base to determine possibilities for reuse, rework or discovering difficult to find code
- Source level code optimization XsltOpt
- Algorithmic analysis of software to help find and fix insecure or inefficient code
- Publication of parts or all of codebases in different combinations and media.
- License management and auditing.
- Detecting potential patentable algorithms.
- Validation of the accuracy of documentation and comments with respect to the actual code.
- Language independent IDE's that can be used across languages and roles.

The problem with Programming Language AS a Markup Language

A common sentiment regarding marking up programming languages is that it is unnecessary because programming languages are themselves a markup language, or at least provide a strict syntax which is computer parsable. There are several problems with this reasoning. A big problem is that the syntax and semantics of programming languages is primarily aimed at one consumer - the compiler. This syntax can be extremely tedious to use for other purposes. In general it requires a complete specific language aware parser to extract the program structure from the syntax. Even extracting comments requires a language aware parser.

For example, consider this Java snippet:

Figure 1

```
System.out.println("This is /* A comment */ or // is this " + /* A comment */ " or // is this ") ; // "A comment"
```

Or this XQuery statement

Figure 2

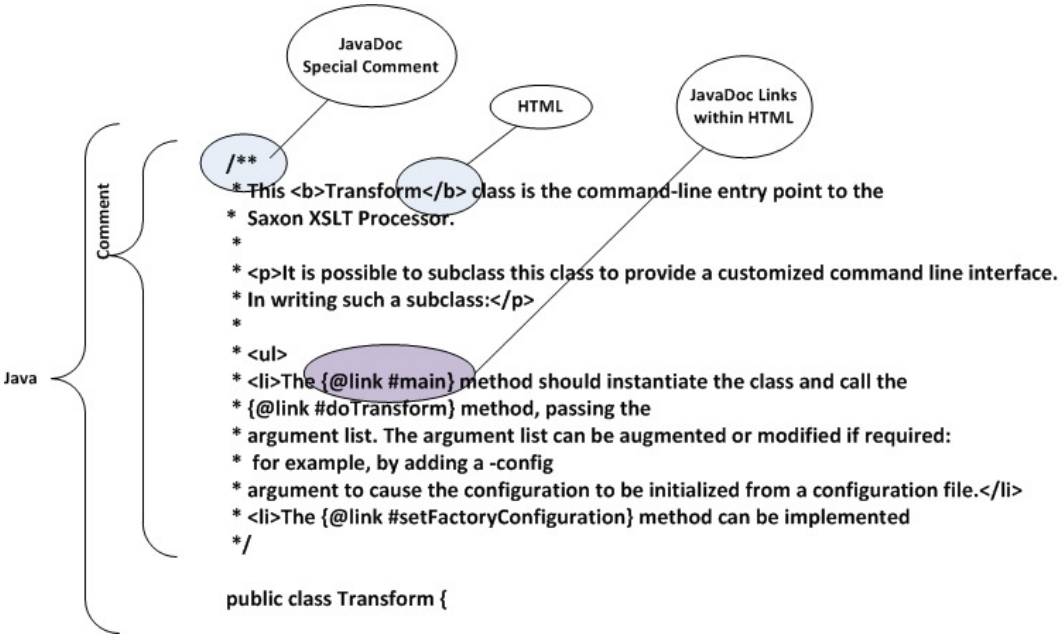
```
return <element>
    (: Is this a comment :) { () (: or is this a comment :) } <!-- What about this -->
```

```
</element>
```

Reliably extracting out the comments from the code is a very tedious task. Furthermore, it is language specific. Every programming language has it's own rules and quirks. Of the same can be said of all languages including XML, however programming languages tend to have non regular structures that are particularly difficult to parse.

But suppose we could easily parse and extract comments, and identify where they were in the code text - what do they mean? By their very nature comments have no explicit semantics. They are just plain text. The traditional way of handling this is to add a kind of micro-format or embedded markup within comments used by particular tools. For example JavaDoc uses a set of special tags as well as HTML to mark up portions of the text within comments. This becomes a very complex 'Russian Doll' layering of different markup languages embedded within each other.[1] Even then, the target audience of JavaDoc is limited. It can be parsed by very few tools and the output is limited to few uses. To target other uses we need different markup - and special tools to parse that.

Figure 3: JavaDoc Example



Nested markup in JavaDoc

From Saxon 9.4 source.

Imagine we decided on a good markup for embedding in comments, there is a fundamental limitation imposed by the syntax - they are missing a target. Comments appear at a single point in the text stream. There is nothing explicit that indicates to what they refer. There are conventions that are generally humanly understandable, such as placing a comment preceding a function means it applies to that function, but the conventions are easily broken.

Consider this simple Java snippet

Figure 4

```
/*
 * This file implements the PixieDust portion of Magic Products.
 */
 class PixieDust {

 // The pixie function
 private int magic = 0;     // Magic
 public int pixie( int dust )
 {
       /*
       * Be careful that magic is not set somewhere else
       */
      return ((magic < 10) ? 0 : magic)   + dust ;  // Conditionally add dust
 }
}
```

To a human it is fairly clear which comment applies to what. One generally follows the convention that block comments precede their target and inline comments follow their target. But from a parsers perspective it's entirely unclear what comment is associated with which language construct, if any. Note, for example, the comment "The pixie function" immediately precedes the variable declaration for "magic" yet is obvious to human readers that it refers to the function declaration instead. The comment "Be careful that magic is not set somewhere else" doesn't target the following expression exactly, it's a more out of bands annotation concerning the algorithm. We could of course start adding markup in the comments to clarify to what they refer.

```
/*
 *  <file name="pixiedust.java">This file implements the PixieDust portion of Magic Products.</file>
 */
 class PixieDust {


  // <function name="pixie">The pixie function</function>

  private int magic = 0;     // <description variable="magic">Magic </description>
  int pixie( int dust )
  {
```

```
       /* <annotation>
       * Be careful that magic is not set somewhere else
         </annotation>
       */
      return ((magic < 10) ? 0 : x)   + dust ;  // Add magic to dust if >= 10 ;  // <explanation reference="((magic < 10) ? 0 : magic)">Conditionally add d

  }
}
```

This is starting to look pretty crazy. Unfortunately this is the essence of what implementations of embedded markup in programming languages end up doing. Furthermore this is still an extremely fragile binding. Without a full language parser available, the comment markup processor has no way of identifying and validating the targets. For example, if I wanted to query "Give me the code for function 'pixie'" the above markup would not be able to answer that.

We could go further and try to express the more of the structure in comment markup using containment instead of positioning and named references.

Figure 5

```
/*
 *  <file name="pixiedust.java">
 *  <description>This file implements the PixieDust portion of Magic Products.<description>
 */
class PixieDust {
// <variable name="magic">
 int magic = 0;     // <description>Magic</description>
// </variable>
// <function name="pixie">
// <description>The pixie function</description>

int pixie( int dust )
{
      /* <annotation>
      * Be careful that magic is not set somewhere else
        </annotation>
      */
       // <explanation>

      return ((magic < 10) ? 0 : magic)   + dust ; //  <description>Conditionally add dust</description>
     // </explanation>
}
//< /function>
//</file>
```

Now we approach a markup, embedded in comments, that has more semantics. it's also redundant and extremely difficult to read, edit, maintain and validate. it's likely that for these reasons markup in programming languages is general limited to embedding documentation. When simply using markup for documentation (as opposed to the above structure) the markup is "close to" the code but it really doesn't markup the code. This is very useful as documentation that is close to the code is generally more likely to be updated and kept in sync but it's really just a way of managing documentation and adds little if any value to the code itself

Pragmas and Annotations

Another form of programming language markup is Pragmas and Annotations. These are designed to be part of the language itself and thus have semantics as opposed to comments. Java is a good example with a quite expressive annotation capability. The problem with annotations and pragmas however lies specifically with their coupling to the language. Not only are they language specific but have predefined or proprietary semantics and are part of the language itself, rather than meta-data about the language. Thus by adding pragmas or annotations you are changing the program itself; this violates many of the goals of markup in that it changes the underlying semantics.

It is possible that programming languages could be extended to allow for more structured documentation or markup such as annotations that does not affect the semantics of the program in any way. However this would still require a full language parser and would inherently be language dependent.

Making Markup a First Class Citizen

Much of the problem is imposed on us by trying to work within the existing syntax of the language. Either by using comments which have no semantics and complex nested syntax to support markup, or by annotations which have language defined semantics and so affect the underlying code directly. Instead perhaps we can learn from the lessons Text Markup tradition.

With Text Markup there is fundamentally the same problem. The original text is preserved while adding additional data or meta-data. The solution used by markup such as XML is to discard the concept that serialized format of the document itself must be the original text. Rather the text is contained within markup instead of putting markup within the text. The markup processor can parse the document, and if desired reproduce the original text exactly. Just as with human languages, this technique allows a single syntax to be used to markup any number of languages and a common tool-chain to process them. This inversion of representation is not without precedent in programming languages. Preprocessors such as cpp and m4 are a similar concept, and language tool-chains typically have the ability to integrate transformation of source documents prior to compilation. Many modern IDE's have the ability to install plug-ins for working with additional document formats while utilizing the full features of the IDE.

A journey through Literate Programming

A discussion about marking up programming languages is not complete without consideration of Literate Programming LiterateProg.

Literate Programming, as introduced by Donald Knuth, is a very interesting concept. The idea is that commentary (prose) is 'interwoven' with code and that a post processing phase separates the two into either documentation or source code. This has some overlap with the concept of marking up programming language but I feel it is a quite different path.

The perspective I propose is that source code be treated more like Historic or Primary documents. That is that the source code is the primary information and that markup is annotation or meta-data applied to document.

This philosophical distinction has practical implications. With Literate Programming the source code is "tangled". That is snippets of source code are interspersed throughout the prose and not necessarily even in the same order as the final resulting program. This has the advantages of providing an emphasis on providing the prose as the primary source so let's the reader understand the exposition in a more customary way, leaving the actual implementation (the code) as almost a side thought.

This is useful for describing the thought processes of design but I argue that it is problematic for the actual engineering of software. The fact is that software source code (or rather, compilers) is very particular about the exact representation. It takes significant effort to produce source code that is parsed and works exactly as intended. Much of this difficulty is in fact the sort of things literate programming attempts to hide such as ordering and placement of statements, code blocks, and compilation units. I suggest that one reason Literate Programming has not achieved wide adoption is that it makes the everyday programmers task more difficult. Curiously, similar statements have been made about C++ for example. "C++ is an excellent language for hiding the trivial and mundane details of a program... like where the bugs are."unknown I belie tangling already difficult to understand code within prose makes the job of both authoring and debugging the software harder, not easier.

With these thoughts in mind I suggest diverging from the Literate Programming perspective and instead look to the source code as the primary information, not something to be hidden within prose, but rather annotated.

CodeUp - Marking up Programming Languages with XML

Considering that it is not a huge technical hurdle to integrate marked up programming language documents into the life-cycle of development tools, let's consider what possible benefits could be gained and effort and costs would be to apply text markup concepts to programming languages.

"CodeUp" is an experimental family of XML schemas designed to experiment with the application of Text Markup techniques to programming language source code. The premise is that marking up source code can add value and that programming languages can be considered similar text in terms of purpose and techniques. To start we take the initial huge leap and decide that like XML based text markup, source code should be contained within an XML document structure, as opposed to putting XML markup within the programming language syntax. This implies that the marked up document is not directly usable by a consumer which assumes the document is syntactically represented as it's programming language. Rather, like with text markup, a transformation must be applied to the marked up document in order to retrieve the 'original' source document.

By starting with a minimal markup then proceeding to more complexity we can experiment with the value vs costs trade-off implicit with adding markup. Markup involves a cost. There is the cost of applying the markup, but in addition there are costs associated with maintenance, readability and transformation. The more complex the markup the higher the cost. Presumably there comes with higher cost, higher value; but like with text markup value is more difficult to measure than cost. One way of describing value is like potential energy. The value added by markup is potential, but may not be realized depending on the actual use of the content. Thus there is a trade-off between how much we are willing to invest in markup compared to the value we hope to extract. The first foothold in marking up a source code document in XML is to simply wrap the source in a root element. This then gives us a scaffolding by which we can incrementally build more complex markup. This is very simple to do but it also immediately exposes us to indirect costs. Once we wrap a source code document in even the most simple root element it is no longer directly usable by the primary consumer - a compiler! Getting over this first step adds significant cost with very little if any value.

Transparent Markup

A critical requirement for source code markup is that the "original" text needs to be preserved. Similar to Text Documents, "original" may be a temporal quality (i.e. there existed prior in time an "original" document which is later marked up). But a similar case is to consider "original" to mean "what would the text be if there was no markup", regardless of which was authored first or authored at the same time. In both cases the requirements are the same: the "original" must be unchanged by markup and be able to be extracted from a marked up document. The simplest case of this is the use of "transparent markup". This is a premise in many text oriented schemas, that is the concept that if you remove all markup what is left is the "original".

Applying that to CodeUp we can start with the first step, a root element.

Figure 6

```
<program xmlns="http://xml.calldei.com/codeup">
/*
* This file implements the Pixie portion of PixieDust Products.
*/

// The pixie function
int x = 0;     // Magic
int pixie( int dust )
{
/*
* Be careful that x is not set somewhere else
*/
return ((x &lt; 10) ? 0 : x)   + dust;  // Conditionally Add dust

}
</program>
```

This simple step is easy to apply, and in fact could be programmatically applied to a set of source code. There is a complication that the text of the code itself may overlap with XML markup; it needs to be escaped with entity escapes or CDATA sections, but that is fairly easy to accomplish. CDATA markers can easily be added by hand to the beginning and end of the file. A simple text parser or an XML enabled GUI tool can accomplish entity encoding easily and risk free. The direct cost of adding this markup is practically zero. However it exposes the indirect costs: the entire development and compilation ecosystem is now broken. In addition if simple hand entering or editing of the source code is done care must be taken to escape what was source code but in the context of a marked up document would be misinterpreted as markup. More complex escaping may need to be done if the code contained XML-like markup to begin with such as JavaDoc with embedded HTML, especially if the contained markup itself has CDATA sections or entity encoding.

Compilers

The most obvious problem is that now any consumer of the source code, such as compilers, which previously expected it to be syntactically identical to the original document is broken.

That is likely why this approach is rarely taken in practice. However the solution for automated processing (compilers, parsers etc.) is actually quite easy. Most development tool-sets accommodate preprocessing quite easily as a normal and expected procedure. For example in a Makefile or ANT script it is quite easy to add a step which would consume the CodeUp document and produce the original source. A Very simple XSLT XSLT, XQuery XSLT or other XML processor could be used.

The following XSLT is sufficient extracting the data from this example. In fact this is sufficient for any markup which follows the constraints of transparent - that is it removes all markup and leaves the underlying text unchanged. Equivalent to this is using any XML parser and outputting only text nodes.

Figure 7: Simple XSLT for transparent markup

```
<?XML version="1.0" encoding="UTF-8"?>
<XSL:stylesheet xmlns:XSL="http://www.w3.org/1999/XSL/Transform" version="2.0">
<XSL:output method="text"/>
<XSL:template match="*">
    <XSL:value-of select="."/>
</XSL:template>
</XSL:stylesheet>
```

Applying this transformation as part of the compilation process is quite simple in most programming environments. A somewhat more complex transformation if source line numbers and filenames are to be preserved, but again the techniques are well known and commonly used in other situations.

Editors and IDE's

Authoring and Editing marked up code is another challenge. Once the code no longer parses as the original source language syntax, any editors or IDE's which are designed for the specific language will no longer work as expected. Simple things like syntax highlighting will be broken. More complex features such as refactoring, class browsing, outlining etc. will not work. This can be a major hurdle to embracing code markup.

Fortunately the IDE's with the more complex features tend to also have more customization abilities. Eclipse Eclipse is a prime example which is designed to be extended to multiple languages and syntaxes. Another be possibility is that once code is in XML form, then XML editors can be used as intelligent code editors. In fact the more mark-up applied to source code the more XML editors can be leveraged as software editors. However it is difficult to imagine how a pure XML editor could match the features of a language specific IDE even if every token was marked up. Modern IDE's use incremental compilation and language awareness to do things like method name completion which would be impossible to achieve with static schemas alone. An XML editor using a language schema may be able to know that a *for loop* needs a body but unless they can reflect on the full parse tree of the whole compilation unit they couldn't for example know what methods a particular instance of an object are public or even what variables are in scope.

For authors that use less specific editors the opposite is the problem. XML markup itself poses no problems in a text editor, but it also provides no benefit. Much like the authors of Text markup, editing marked up code with non-XML aware editors poses challenges to many authors. For those comfortable authoring with markup syntax the experience may be easy, but for those who are not, the use of a specialized editor or macros in customizable editors (like Emacs Emacs) may be more productive.

The rocky road from A Text Syntax to an XML Syntax

Once we've accomplished the initial step of converting a programing language to an XML Document, and all that entails including editors, IDE's, tool-chains and work-flows we can start to explore the benefits and costs of enriching the markup.

Continuing under the premise of "Transparent Markup" what can be improved while maintaining the original text as source code? Much of this can be done as language neutral markup because it is entirely optional. Strip out the markup and you get the source code. Markup need only be informative or helpful but need not be semantically perfect.

Imagine the following 'improvements' to the original sample.

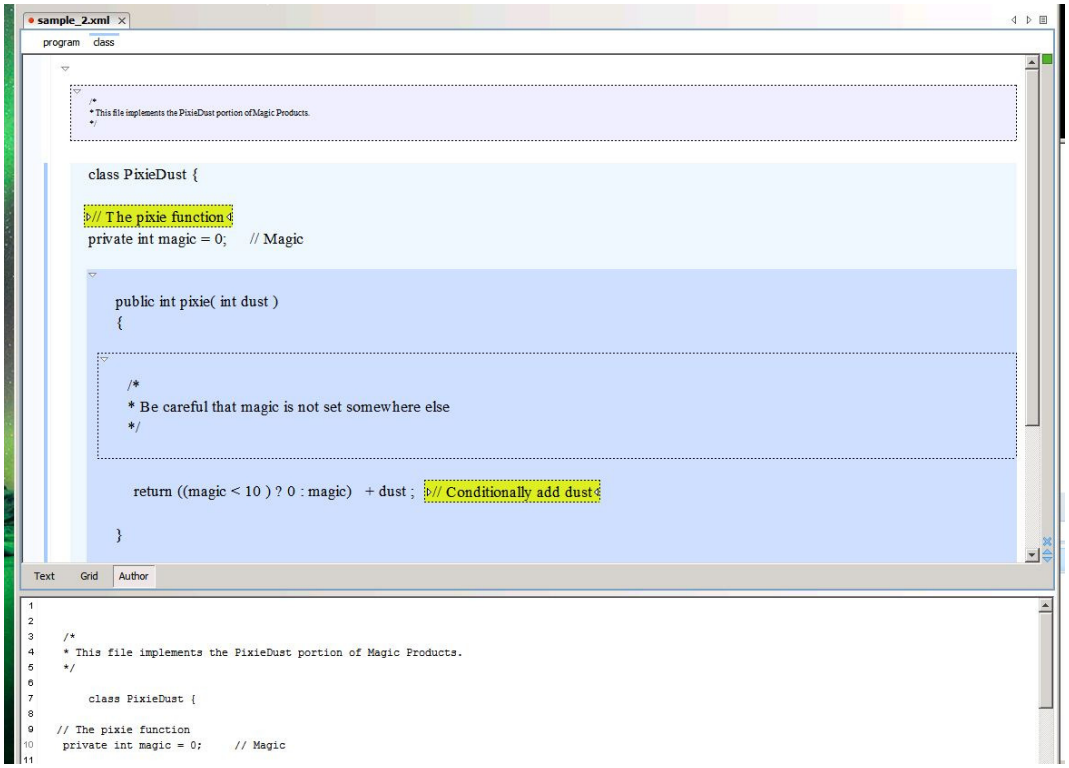
Figure 8

```
<program xmlns="http://xml.calldei.com/codeup" language="Java" file="pixiedust.java">
    <comment scope="file">
    /*
    * This file implements the PixieDust portion of Magic Products.
    */
    </comment>
    <class>
    class PixieDust {

    <comment inline="true">// The pixie function</comment>
    private int magic = 0;     // Magic
    <method>
    public int pixie( int dust )
    {
      <comment>
      /*
      * Be careful that magic is not set somewhere else
      */
      </comment>
      return ((magic &lt; 10 ) ? 0 : magic)   + dust ;  <comment inline="true">// Conditionally add dust</comment>

    }
    </method>
    </class>
}
</program>
```

This begins to add the sort of semantics common in text markup while still being transparent. The simple XSLT still produces the original source code by stripping all markup, and using an XML IDE starts to provide value in terms of authoring

For example the following is the Author view in Oxygen Oxygen [2]with a very simple CSS applied to the schema

Figure 9: Oxygen Author mode view of the above source.



The Dead End of Transparent Markup

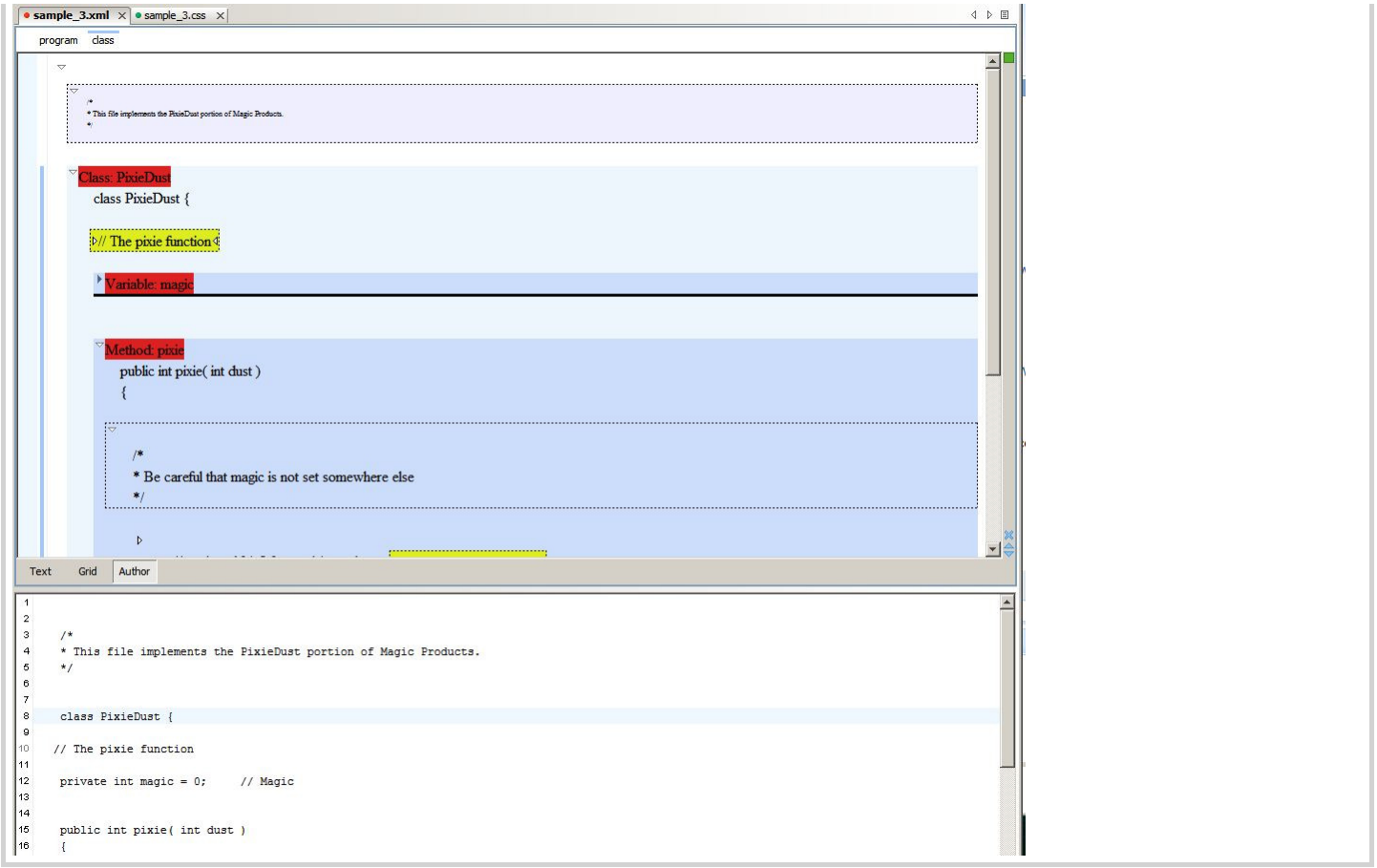
The previous attempt has some valuable markup but still lacks significant semantic value. For example the question "what is the method called 'pixie'" or "To which language element is a comment referring" cannot be answered with only this XML markup. Suppose we go further and embed more of the language semantics within the markup, while still staying with the goal of Transparent markup. Something like this is an obvious next step.

Figure 10

```
<program xmlns="http://xml.calldei.com/codeup" language="Java" file="pixiedust.java">
    <comment scope="file">
    /*
    * This file implements the PixieDust portion of Magic Products.
    */
    </comment>
    <class name="PixieDust">
    class PixieDust {

   <comment inline="true">// The pixie function</comment>

   <variable scope="private" name="magic" type="int" initializer="0">
    private int magic = 0;     // Magic
   </variable>

    <method name="pixie" scope="public" return="int">
    <parameter name="dust" type="int"/>
    public int pixie( int dust )
    {

    <comment>
    /*
    * Be careful that magic is not set somewhere else
    */
    </comment>
        <statement type="return" expression="((magic &lt; 10 ) ? 0 : magic)    + dust">
        return ((magic &lt; 10 ) ? 0 : magic)    + dust ;  <comment inline="true">// Conditionally add dust</comment>
    </statement>
    }

    </method></class>
    }
</program>
```

With a little CSS magic Oxygen Author mode displays the additional semantics including outlining and collapsing. Using the XSLT in Figure 7 the result is the original text.

Figure 11: Oxygen Author mode view of the above source.



At this point we have captured a significant amount of the language semantics entirely in XML Markup and can indeed perform some semantic queries such as "What is the body of the pixie function" or "What are the method names, parameters and return values in the pixiedust.java file". We could use this information to achieve many of the goals of Markup such as production of different representations, limited semantic queries, publication, aggregation even code refactoring - all using XML tools only.

Unfortunately to achieve this we have suffered some of the main problems of embedded markup: that of duplication of information, fragility and difficulty authoring, editing and viewing. Also the markup does not embed the full semantics of the language. For example call graphs or object dependencies cannot be generated because the semantics of expressions is not modeled. It seems we have reached a dead-end as it's unlikely software authors are willing to put the effort into this level of markup considering that it's really duplicating the code in two different ways serving the author little gain and adding great effort. Furthermore the chances of the markup being valid over the life-cycle of the program are slim. It would take a very dedicated author to make sure that the entire markup is updated as code is updated and there is no automatic way of validating it without comparing the results of a full language parser against the markup. So in reality the value is negative and the cost is high.

Semi-transparent Markup

Considering the dead end attempting pure transparent markup, let's back up and reconsider the original goals. When we wanted to maintain the integrity of the "original text" of the programming language, perhaps the goal can be achieved by relaxing the concept of "original". Instead, considering we have already added a process of transformation to extract the "original", perhaps it's sufficient that the transformation itself can produce some of the text as a side effect of markup instead of requiring that all output be literal text. As long as the results are identical to the *intended* original we can consider it semantically equivalent.

For purposes of this discussion I will call this markup style "Semi Transparent". That is, a combination of markup and plain text, when transformed with a known transformation is able to produce what would be the "original" text if we did not use markup.

Consider this simple fragment

```
class PixieDust { body }
```

One could represent this using markup but without duplication by something like

```
<class name="PixieDust">
body
</class>
```

These two are semantically equivalent. Using a simple transformation the *Intended Text* could easily be produced but without duplication of data.

Expanding on this let's see what the full sample might look like using a semi-transparent markup.

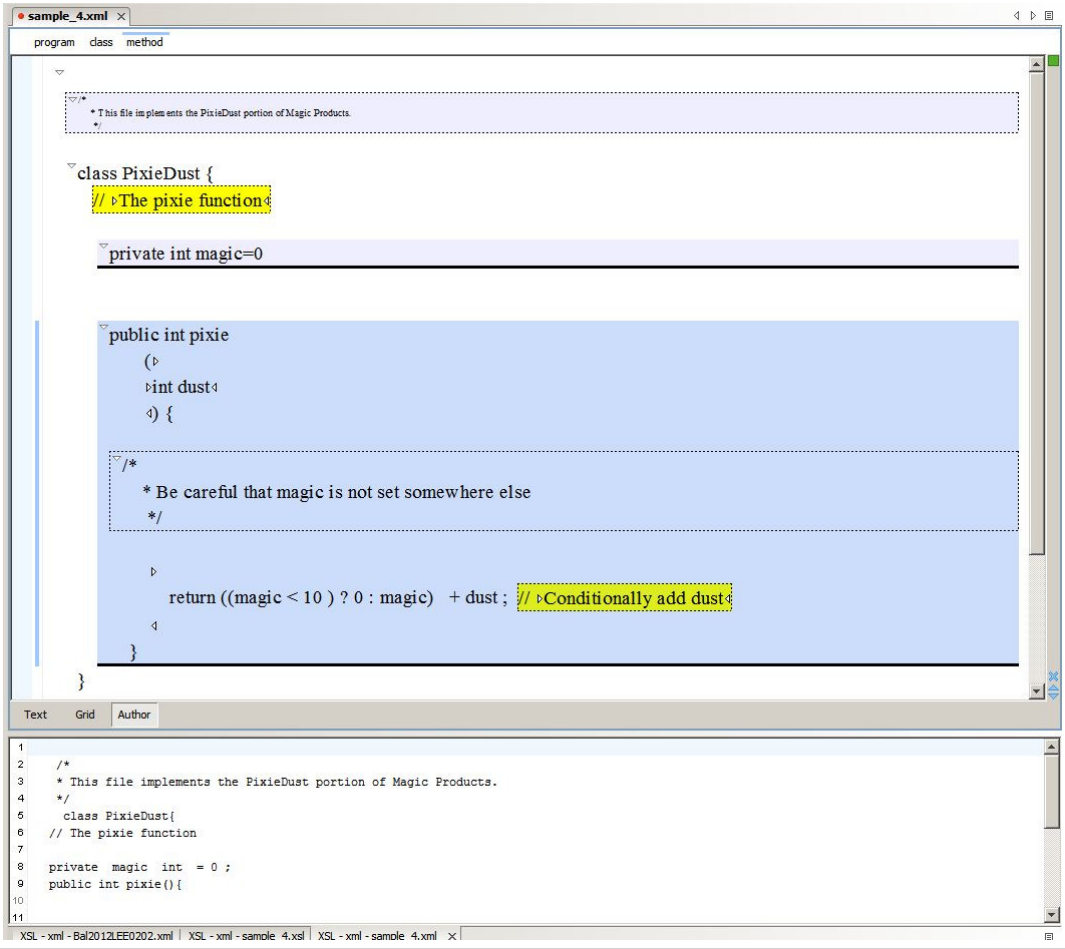
Figure 12

```
<program xmlns="http://xml.calldei.com/codeup" language="Java" file="pixiedust.java">
    <comment scope="file">
    * This file implements the PixieDust portion of Magic Products.
    </comment>
    <class name="PixieDust">
   <comment inline="true">The pixie function</comment>
```

```
  <variable scope="private" name="magic" type="int" initializer="0"/>
  <method name="pixie" scope="public" return="int">
  <parameter name="dust" type="int"/>
   <comment>
   * Be careful that magic is not set somewhere else
   </comment>
       <statement>
           return ((magic &lt; 10 ) ? 0 : magic)   + dust ;  <comment inline="true">Conditionally add dust</comment>
       </statement>
   </method></class>
</program>
```

With some more CSS magic Oxygen Author mode displays the additional semantics.

Figure 13: Oxygen Author mode view of the above source.



A slightly more complex XSLT is required to produce the *Intended Text*

Figure 14

```
<?XML version="1.0" encoding="UTF-8"?>
<XSL:stylesheet xmlns:codeup="http://xml.calldei.com/codeup" xmlns:XSL="http://www.w3.org/1999/XSL/Transform" version="2.0">
    <XSL:output method="text"/>


    <XSL:template match="codeup:comment">
        <XSL:if test="@inline"><XSL:text>// </XSL:text></XSL:if>
        <XSL:if test="not(@inline)"><XSL:text>/*</XSL:text></XSL:if>

        <XSL:value-of select="text()"/>
        <XSL:if test="not(@inline)"><XSL:text>*/</XSL:text></XSL:if>

    </XSL:template>

    <XSL:template match="codeup:variable">
        <XSL:value-of select="@scope"/>
        <XSL:text>  </XSL:text>
        <XSL:value-of select="@name"/>
        <XSL:text>  </XSL:text>
        <XSL:value-of select="@type"/>
        <XSL:text>  = </XSL:text>
        <XSL:value-of select="@initializer"/>
         <XSL:text> ;</XSL:text>
     </XSL:template>


    <XSL:template match="codeup:class">
        <XSL:value-of select="@scope"/>
        <XSL:text> class </XSL:text>
        <XSL:value-of select="@name"/>
        <XSL:text>{</XSL:text>
        <XSL:apply-templates select="node() "/>
        <XSL:text>}</XSL:text>
    </XSL:template>
```

```
    <XSL:template match="codeup:method">
        <XSL:value-of select="@scope"/>
        <XSL:text> </XSL:text>
        <XSL:value-of select="@return"/>
        <XSL:text> </XSL:text>
        <XSL:value-of select="@name"/>
        <XSL:text>(</XSL:text>
        <XSL:apply-templates select="parameters"/>
        <XSL:text>){</XSL:text>
        <XSL:apply-templates select="node() except element(parameters)"/>
        <XSL:text>}&#10;</XSL:text>
    </XSL:template>

</XSL:stylesheet>
```

Towards a fully XML syntax

As we can see from Semi-Transparent markup there is an obvious path towards a fully XML Syntax. If you can mark up classes, function, variable declarations, statements, comments, why stop there? One could envision a markup fully detailing the semantics of every aspect of the programming language. A next step might be encoding control flow (if/while/switch), and expressions. Is there an obvious end to this path? At what point do we achieve a fully XML syntax, and is it even desirable?

A look to what is considered XML Syntax languages can show the possible results. XSLT, for example, is considered an XML Syntax language. But XSLT is not fully marked up! Consider this snippet in XSLT 3.0.

Figure 15: XPath embedded in XSLT XPathXSLT

```
 <?XML version='1.0'?>
<XSL:stylesheet
    version="3.0"
    xmlns:XSL="http://www.w3.org/1999/XSL/Transform"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
<XSL:template match="/">
        <XSL:variable name="f1" select="
            function($x as xs:integer) as (function(xs:integer) as xs:integer){
                    function ($y as xs:integer) as xs:integer{
                        $x*$x + $y * $y
                    }
            }
        "/>
        <XSL:value-of select="$f1(2)(3)"/>
</XSL:template>
</XSL:stylesheet>
```

Here we have a non-XML syntax child language (XPath 3.0 XPath) embedded in a XML syntax language (XSLT).

XQueryXXQueryX represents a more "pure" XSLT syntax in that even XPath expressions within XQueryX are fully decomposed and marked up as XML. However XQueryX is generally considered unreadable and unwritable by humans except in very small doses. At some point, adding markup or expressing every semantic detail as markup becomes untenable - at-least for humans.

As an example just the select expression in the above XSLT translated to XQueryX is shown

Figure 16

```
<?xml version="1.0"?>
<xqx:module xmlns:xqx="http://www.w3.org/2005/XQueryX"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://www.w3.org/2005/XQueryX
                               http://www.w3.org/2005/XQueryX/xqueryx.xsd">
  <xqx:mainModule>
    <xqx:queryBody>
      <xqx:inlineFunctionItemExpr>
        <xqx:paramList>
          <xqx:param>
            <xqx:varName>x</xqx:varName>
            <xqx:typeDeclaration>
              <xqx:atomicType xqx:prefix="xs">integer</xqx:atomicType>
            </xqx:typeDeclaration>
          </xqx:param>
        </xqx:paramList>
        <xqx:typeDeclaration>
          <xqx:parenthesizedItemType>
            <xqx:typedFunctionTest>
              <xqx:paramTypeList>
                <xqx:sequenceType>
                  <xqx:atomicType xqx:prefix="xs">integer</xqx:atomicType>
                </xqx:sequenceType>
              </xqx:paramTypeList>
              <xqx:sequenceType>
                <xqx:atomicType xqx:prefix="xs">integer</xqx:atomicType>
              </xqx:sequenceType>
            </xqx:typedFunctionTest>
          </xqx:parenthesizedItemType>
        </xqx:typeDeclaration>
        <xqx:functionBody>
          <xqx:inlineFunctionItemExpr>
            <xqx:paramList>
              <xqx:param>
                <xqx:varName>y</xqx:varName>
                <xqx:typeDeclaration>
                  <xqx:atomicType xqx:prefix="xs">integer</xqx:atomicType>
                </xqx:typeDeclaration>
              </xqx:param>
            </xqx:paramList>
            <xqx:typeDeclaration>
              <xqx:atomicType xqx:prefix="xs">integer</xqx:atomicType>
            </xqx:typeDeclaration>
            <xqx:functionBody>
              <xqx:addOp>
                <xqx:firstOperand>
                  <xqx:multiplyOp>
                    <xqx:firstOperand>
```

```
                    <xqx:varRef>
                      <xqx:name>x</xqx:name>
                    </xqx:varRef>
                  </xqx:firstOperand>
                  <xqx:secondOperand>
                    <xqx:varRef>
                      <xqx:name>x</xqx:name>
                    </xqx:varRef>
                  </xqx:secondOperand>
                </xqx:multiplyOp>
              </xqx:firstOperand>
              <xqx:secondOperand>
                <xqx:multiplyOp>
                  <xqx:firstOperand>
                    <xqx:varRef>
                      <xqx:name>y</xqx:name>
                    </xqx:varRef>
                  </xqx:firstOperand>
                  <xqx:secondOperand>
                    <xqx:varRef>
                      <xqx:name>y</xqx:name>
                    </xqx:varRef>
                  </xqx:secondOperand>
                </xqx:multiplyOp>
              </xqx:secondOperand>
            </xqx:addOp>
          </xqx:functionBody>
        </xqx:inlineFunctionItemExpr>
      </xqx:functionBody>
    </xqx:inlineFunctionItemExpr>
  </xqx:queryBody>
 </xqx:mainModule>
</xqx:module>
```

Translation from XQuery 3.0 to XQueryX XQueryXApplet

Further down the path of an XML Syntax

There is much further we can take this exploration. For examples the comments are still unreferenced; they appear inline with code but no clear semantic reference to which they refer. That could be solved with cross reference tags. Statements and expressions are yet to be fully decomposed. The full syntax of statements and expressions could be marked up. If we keep on this path we will end up where XSLT and ultimately XQueryX have. A fully semantic markup that in this author's opinion is impractical for human authoring and consumption.

There is a reason that programming languages are largely expressed using more unstructured human readable syntax. That is that humans are ultimately the main creators and consumers of the language. Even though a computer has to interpret the language, so it must be quite explicit and unambiguous, humans have to write, read, maintain, and debug it.

That doesn't mean there isn't room for improvement. As we have seen with markup such as JavaDoc there is still a strong desire to add layers of semantics to programming languages. A universal markup for programming languages may well be a practical and useful tool. And possibly some compromise between no markup and fully semantically marked up might be a 'sweet spot' for some programming languages.

The Golden Road (to Unlimited Devotion)

Like the song of the same name GoldenRoad, applying markup to Programming languages to it's full conclusion may require unlimited devotion. Even if every single statement, expression and symbol was marked up the resulting information would not include all the information that a compiler understands by parsing the plain text due to the compiler understanding the semantics of the entire corpus. This means even if we follow the Golden Road to it's end we wont end up with the information a language specific parser does. So in a way this seems like a dead end road. What can markup gain us?

With Text markup, the data added by markup supplies us with generally not available with the original text. With Programming Languages, the information contained in the program statements themselves is ultimately available, even if deeply hidden in the bowls of the compiler. Perhaps it is best if that in information be pulled from the compiler's grasp instead of duplicated by markup. But what of information the compiler discards, is unaware, or is unimportant to the compiler but useful to humans ? That may be a realm ripe for markup.

So what information in software and it's incarnation in programming language text not available or not of interest to the compiler? Clearly comments and documentation come to mind. But there is more

The very tedium of engineering required to construct a good program, and which the compiler assumes but generally not part of the compilers undemanding, or discarded as part of the compilation may be ripe for markup: information such as the structure of a complex system. What pieces need to go together and what is irrelevant (and may be changed). What is the rationale behind seemingly arbitrary choices such as naming conventions, compilation units and program source structure. For example in version of troff Troff which I was fortunate enough to port in the mid 80's from Unix to VMS, the C Source was written (originally ported from PDP11 assembly) assuming that global variables were aligned in memory in the order declared. This was implicit but required by the program and broke in exiting ways on VMS where the C compiler ordered variables alphabetically not by declaration order. Perhaps Markup may have indicated this requirement?

Sometimes certain functions, variables and comments should belong together. Markup could indicate this where a compiler does not. Functions describing a public and private interface could be indicated in languages where the compiler does not provide such a facility. There are many areas where the compiler is ignorant but the programmer has important knowledge. Markup could be effectively used for these situations.

Conclusion: To Mark up or Not to mark up

The path to marking up programming languages is neither straight nor objective. There are branches along the path that depending on the techniques and goals may or may not be easy or valuable. There is certainly pain (cost) to be had by adding markup, but there is also potential value. Much of the rationale behind Text markup also applies to programming languages and we have shown that similar techniques and tools can be used for both. But as with Text, it is a continuum not an absolute. Depending on your effort, willingness, perspective and desire for additional value, varying types and amounts of markup may be appropriate. Traditional embedded markup can be easily simplified by this approach. I have shown that one can approach the problem incrementally - even some markup can be valuable yet easy to introduce. The hardest part is accepting and implementing the inversion of programming language *within* markup as opposed to markup within the programming language syntax. By placing the language within the markup we are able to use standard markup processing tools largely programming language agnostic as well as keeping the source relatively clean compared to embedding markup within language constructs. Another issue is when markup overlaps the information inherent in the programming language itself. It is unlikely that programmers will be willing to duplicate information the

compiler will understand without the markup. Furthermore it is generally a bad engineering practice to duplicate information as the multiple versions lead to maintenance issues (two watches rarely show the same time).

On the other hand, existing programing language and markup tools are readily available to accomplish many tasks with little effort. XML-centric editors could be used for programming IDE's, with some limits. Common programming IDE's can be enhanced with understanding of markup. The compilation tool-chain has for decades been easily extendible to transformation of source documents. By taking the initial step and effort of tooling up for marked up source there is a wide spectrum of value that can be achieved with incremental cost. But and at what value and at what cost ?

Markup for high value and low cost

Examining what types of markup are required to encode what semantic information leads one wondering "is it worth it?". There are some goals with moderate value that can be achieved at low cost, but much which can be achieved only with very high cost if at all, and the value is debatable.

Presuming you consider the retooling necessary for wrapping programming languages within markup (inversion) a fixed sunk cost, what goals can be achieved with minimal additional work?

Source Structural Decomposition

With fairly minor markup, mainly wrapping all classes, methods and their corresponding dependent variable declarations and comments in a kind of structural markup, structural decomposition can be achieved.

By this I mean you can automatically recognize and reorganize your codebase. This might be useful for bulk refactoring or for presenting a database of searchable code snippets. But how useful and reliable will this actually be? Without full knowledge of dependencies (data, call and usage graphs) and subtle details of the language, reorganizing based on simple source file structural boundaries is fragile. It has some value but also high risk.

Another use for structural markup is providing an IDE with the ability to do outlining, expansion, collapsing, indexing etc. This is definitely a useful feature. However language specific IDE's commonly provide this feature already. The main advantage of achieving this through markup would be to be able to use a language agnostic editor, such as shown through the examples using Oxygen. But since there exists language specific IDE's that do this as good or better than could be done with simple structural markup alone, it's unlikely a developer would move from a more functional IDE to a less functional one especially considering the cost of adding the markup.

Associating comments with language elements

As discussed previously, it is not at all obvious to a computer parser what comments are associated with which language elements. Even the simple case of "block comments precede the element" and "inline comments are after the element" is frequently incorrect. A human usually has no problems detecting the association but without additional hints (markup) a computer often cannot get this right.

A little markup (low cost) could provide this association easily by containment. However what value is this? To human readers it's little value because they can already usually associate the comment by context. To commuters it might ultimately be of some, but it's not immediately obvious. Since the compiler cannot readily interpret the comment, other uses need to be found where this association is valuable.

Adding additional structure

Not all necessary structure of a program is available as explicit notation in every programming language. A simple example is the separation of variable declaration and use. Variables generally need to be declared before use but it is up to the engineer to arrange the source such that the compiler sees the declaration first. Markup may aid in making explicit the implicit dependencies of pieces of software in order to create structure beyond that which the language allows. Such a markup structure could facilitate refactoring, dependency analysis and reuse.

Markup with high cost

Semantic Analysis

The really high value possibilities lie in the area of semantic analysis. For example call graphs, dependency checkers, flow control graphs, algorithmic analysis etc. These are areas that could be of extremely high value. Unfortunately they have high cost in terms of both markup and execution. To do something as simple as a call graph, all expressions and identifiers would need to be marked up. In addition full language semantics would need to be understood in order to resolve overloads, virtual functions, namespaces, indirect calls etc. It is not sufficient to simply mark up every bit of the program; a semantic analysis requires full understanding of the language itself.

Essentially the entire semantics of the program would need to be marked up and available to analysis. To achieve this would require markup as verbose complex as XQueryX Figure 16 as well as a knowledge engine that understands the meaning of the language constructs.

Experience has shown that people do not want to author in this level of markup. The history of mathematics and programming languages has generally been towards concise and humanly readable syntax. In fact many of the 'pure XML' syntax languages like XSLT and XProcXPath have active groups attempting to reduce the markup in the language providing for an alternate syntax which does not require the verbosity of XML markup. Furthermore describing the information inferred by a compilation of programming languages is extremely difficult in pure XML vocabularies. So even if the syntax was available to XML tools as fully marked up software, the semantics the representation is difficult.

Extracting semantics

As discussed there are a few areas which adding markup may be of little effort and add value that is not already explicit in the language. In particular are in areas of documentation, comments and higher level structures. However the highest value, as well as the highest cost is in the semantics of the underlying language structure which is surfaced by the actual language syntax.

Unlike text however this semantics is entirely computable. Unfortunately it is largely inaccessible as it lies in the black hole of the interior of the compiler or language specific parser and generally unavailable for other purposes.

Suppose that were not true. Suppose that the semantics represented by a program were exposed in a unified syntax such as XML and in a open public schema. Then most of the high value hoped for by making up programming languages could be achieved instead by extracting the semantic model from the compiler.

This could lead to a very powerful ecosystem of software analysis not bounded by the constraint and narrow vision of the compiler. Such a data model should not be difficult to generate by enhancements to existing tools as all the knowledge is already available to the compiler. It could be supplemented by additional markup within the source and then combined to produce a comprehensive data-model that can be stored, queried, analyzed and processed using existing markup tool-chains.

I propose that the difficult work is in designing such a schema and encoding the semantics of the language. Can there be a reasonable schema representing a programing language model which is language neutral? Or do different languages need their own schemas. Perhaps a tree of schemas with some common representation of generic programming language constructs with branches for particular languages. In addition, to do semantic analysis requires understanding the semantics of the model not just the structure.

Again we can look to the Text Markup work for design patterns on how to do accomplish this. The road is still open for inquiring minds and brave souls.

Conclusion: Proposals

I suggest that there is value and small cost in marking up programming languages for the purposes of comments, documentation, and most importantly, to make explicit the higher level structure of the program which the programming language itself does not have the capability to describe. However for annotation which duplicates the information inferable from the programming language itself the cost is high and the value is likely negative due to inherent problems with information duplication. The method with least impact on existing tools is embedded markup within comments. However I suggest that since there is high value and fairly low cost, if implemented widely, that the concept of Transparent Markup be considered and implemented as an option by tools makers (particularly in IDE's).

However the information contained within the programming language itself is of high value outside simply compilation. I propose that it is worthwhile to expose this information sourced in the text but only currently available in the compiler as explicitly generated data. Compilers should be enhanced to optionally generate standardized meta information as a byproduct of compilation.

The combination of high level source markup, and compiler generated information could then be utilized by analysis processes. This presents some key areas for future research and consideration.

Standardized Transparent Markup

A standardized schema or family of schemas for transparent markup should be considered. If a standard or convention is developed this may encourage tools makers to adopt the concept into IDE's and other components of the software lifecycle. It should be a fairly easy task to accept a form of transparent markup - even if only to completely ignore it. Simply allowing software to be marked up without interfering with the toolchain opens it to more diverse purposes. If built into IDE's then problems with escaping special markup characters could be minimized. The IDE itself may be able to add some markup based on out of band information such as project structure. Rich text editing support could be added for sections which allow it such as currently supported in JavaDoc. The co-existence of transparent markup with the entire toolchain of software development should be a fairly low cost and high value proposal.

Standardized compiler data structure outputs

Compilers should not be black holes for information. A standard or convention for representing the information consumed and produced by compilers should be considered. If there was a common data structure defined for representing the semantic information derived as a side effect of compilation then compilers could output this as part of the compilation process and expose this rich source of data for other uses. Designing a structure or schema which can be easily produced by a variety of compilers as well as usefully consumed by other processes may be a challenging task, but I suggest it is an avenue that could lead to great rewards.

References

[XSLT] XSL Transformations (XSLT) http://www.w3.org/TR/xslt

[XSLT] XQuery 1.0: An XML Query Language http://www.w3.org/TR/xquery/

[JavaScript] JavaScript aka ECMA-262http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

[JavaScript] The C programming language aka ISO/IEC 9899http://www.open-std.org/JTC1/SC22/WG14/www/standards.html#9899

[JavaScript] The Java programminghttp://www.java.com

[JavaScript] Structured Query Language (SQL)http://en.wikipedia.org/wiki/SQL

[Oxygen] Oxygen XML Editor http://www.oxygenxml.com

[Eclipse] The Eclipse Foundation http://www.eclipse.org/

[Emacs] GNU Emacs http://www.gnu.org/software/emacs/

[CSS] Cascading Style Sheets http://en.wikipedia.org/wiki/Cascading_Style_Sheets

[XPath] XML Path Language (XPath) 3.0 http://www.w3.org/TR/xpath-30/

[XPath] XProc: An XML Pipeline Language http://www.w3.org/TR/xproc/

[XMLLinkedIn1] Dave Patterson on LinkedIn Groups XML and Related Technologies Network, Jan 2012

[XMLLinkedIn2] Paul Monk on LinkedIn Groups XML and Related Technologies Network, Jan 2012

[XMLLinkedIn3] Sean Rushing on LinkedIn Groups XML and Related Technologies Network, Jan 2012

[XMLDev1] Oscar Vives on xml-dev@lists.xml.org, 2012-01-19

[XMLDev2] Philip Fennell on xml-dev@lists.xml.org, 2012-01-19

[XMLDev3] Liam Quin on xml-dev@lists.xml.org, 2012-01-19

[XMLDev4] Henry S. Thompson on xml-dev@lists.xml.org, 2012-01-19

[XMLDev5] Liam Quin on xml-dev@lists.xml.org, 2012-01-19

[JavaDoc] JavaDoc http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html

[HTML] HyperText Markup Language - HTML 4.01 Specification http://www.w3.org/TR/html4/

[XPathXSLT] XPath in XSLT http://www.stylusstudio.com/tutorials/intro-xslt-3.html

[XQueryX] XML Syntax for XQuery 1.0 (XQueryX) http://www.w3.org/TR/xqueryx/

[XQueryXApplet] From the Grammar Test page for XQuery 3.0 http://www.w3.org/2011/08/qt-applets/xquery30/

[LiterateProg] Knuth, Donald E. (1992). *Literate Programming.* ISBN: 978-0-937073-80-3

[GoldenRoad] The Golden Road (To Unlimited Devotion) http://artsites.ucsc.edu/GDead/agdl/goldroad.html

[Troff] troff - a text formatting language http://en.wikipedia.org/wiki/Troff

[XsltOpt] Writing an XSLT Optimizer in XSLT http://www.saxonica.com/papers/Extreme2007/EML2007Kay01.html

---

[1] It can be argued that this markup is actually all part of JavaDoc and not an actual nesting of markup, however the issues involve with parsing the markup remain.

[2] For the purpose of this paper, experiments and samples were implemented using Oxygen Author. This tool provides for fairly simple customization using CSS Stylesheets `CSS` and supports a "WYSIWYG" style of authoring XML marked up documents in addition to a text view. This is not to propose this tool as the solution to the problem of editing, but rather as an example of how existing tools XML editors can be leveraged for the purposes of authoring and editing software. This focuses on the language neutrality enabled by a consistent markup vocabulary regardless of the underlying programming language. Ideally, if these concepts were embraced a more specialized software development IDE (such as Eclipse) should be able to provide a more enhanced experience.