

## **Title**

Creating evolvable programming languages and systems

## **Abstract**

Human languages evolve over time in a way that allows the concepts of the environment to be expressed. The evolution is continuous and much like Darwin's evolutionary theories, only the most useful words continue to be used in life. Programming languages also evolve; however, while each new programming language may look similar to the last, it is for all practical purposes a new and incompatible dialect. This paper proposes a method that describes programming language source code in a way that facilitates the evolution in a similar way to that of human languages. We show that the methods described present many opportunities to integrate the requirements of different languages, data and systems.

## **Introduction**

The English language has evolved; yet at no time was there a change-over point in which people dropped an old version and adopted a new one. However in computer programming languages, this is exactly what is done every few years. The result is that the evolution of computer languages is sporadic, with the number of successful languages spread out like the distribution of human languages. C++ and Java are for all useful purposes as compatible as English is to Chinese. However the truth is that the differences between C++ and Java are more like that of Italian and Spanish, which share enough commonality that it is possible to understand portions without having learnt the other language separately. However, trying to convince a Java compiler of these similarities will not bring much success.

We believe that computer languages should be able to evolve in much the same way as human languages. To achieve this requires us to examine some of the features of human languages, and find methods of mimicking these concepts in software. The result is a method that looks to shake the foundations of the last forty years of programming language theory and design.

In itself the requirement for programming languages to evolve like human languages is not a fundamental one. For this reason it is important to first examine the current incompatibility between programming languages. If by providing a method that allows languages to evolve we can also find possible solutions to these other problems, we have provided the groundwork for better solutions to the problem of computer language incompatibility. However, unless we can show that we can improve the ability of software to solve real world problems, this method will be shown to be useless.

## **The Problem**

Given that programming languages do not currently evolve fluidly, we must develop languages for specific purposes. This is demonstrated by the current proliferation of not only languages, but data systems such as XML, formatting systems such as Cascading Style Sheets(CSS), and various other hybrid data solutions. This creates situations where in every language there are concepts which cannot be easily represented. Sometimes this is a bad choice of language for a task. Other times the language is the correct choice for 80% of the code and not correct for 20%. In these scenarios the programmer must live with the burden of developing work-arounds to the problem. If we could evolve a programming language, then we would be able to introduce and mix concepts back into the source to better solve more specific problems.

The current challenge to language designers is that they must design a language that will last. To do this is incredibly difficult. The lexer and parser must be engineered specifically for the language and that specification must continue to capture the set of concepts for which it was designed. Each language designer must attempt to find the balance between many difficult to manage constraints. A good language must be succinct, allow a developer to capture a wide set of problems, be easy to develop with and generally be based on previous languages in order to reduce the associated learning curve. The result is that we now have thousands of languages - each language attempting to solve an individual problem. In truth many of these languages to a large extent share similar syntax and design. However, each small change creates incompatibilities. We are unable to leverage the code between each system.

The divide between languages and data has also grown. The syntax of XML is wildly different to that of a programming language. This creates a situation where a developer must understand and learn the syntax of wildly different systems in order to accomplish a single task. The following illustrates the many different formats that a programmer must master before being able to successfully create a single web page using Java Server Pages(JSP). The single short amount of text contains four very different programming languages.

```
<html>
<head>
  <style>
  body
  {
    color: #000000;
    background-color:99E999;
  }
</style>

  <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
  <title><%= title %>
</head>
<body>
<div id="body">
<p><h1 class="heading"><%= title %></h1>
<%
  ContentServer cs = ServiceResolve.getContentServer();
  String pageContent = cs.getContent( request.getParameter("id" ) );
%>
<p><%= pageContent %>
<p><a href="javascript:document.app1.setParam('chart_type','horbar');
document.app1.repaint()"/>Show Chart</a>
</body>
</html>
```

Figure 1. Four languages in one source code

A larger problem is that the interaction between each of these languages is very limited. The reason for this is obvious; it is difficult to describe the interactions. Syntactical borders are required to ensure that each language parser is only able to parse the sections relevant to it. To expect current parsers to understand how to deal with unknown data is impossible.

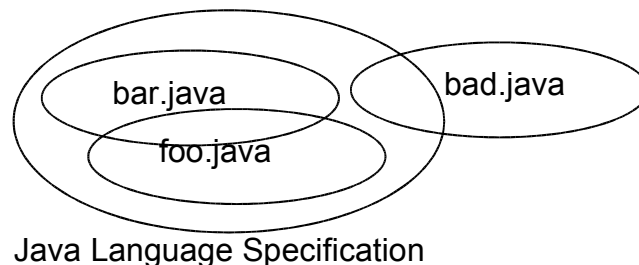
In computer languages there is a certain amount of growth and knowledge which happens with each new language released. Each concept is refined and made clearer and easier to use. The wider developer community expects these well defined concepts in languages. The “if” statement found in languages is now well established in popular languages. Languages that don't build on this knowledge are less likely to be accepted, as they don't build on the knowledge already established with the developers who must use the language. In effect, a dictionary of concepts is slowly being established which a programmer uses in their programming languages. However, while the developer can understand and appreciate the similarities between languages, the compilers can not.

The problems are ingrained in every aspect of how we build programming languages. Unlike natural language it's very difficult to adjust and modify a programming language. The need for compatibility makes the problem difficult. When a language is developed, alongside are requirements of syntax, lexer, parser, libraries, and compiler. If we are to create evolvable languages we need to rethink the fundamental way we develop code.

### Properties of Evolution

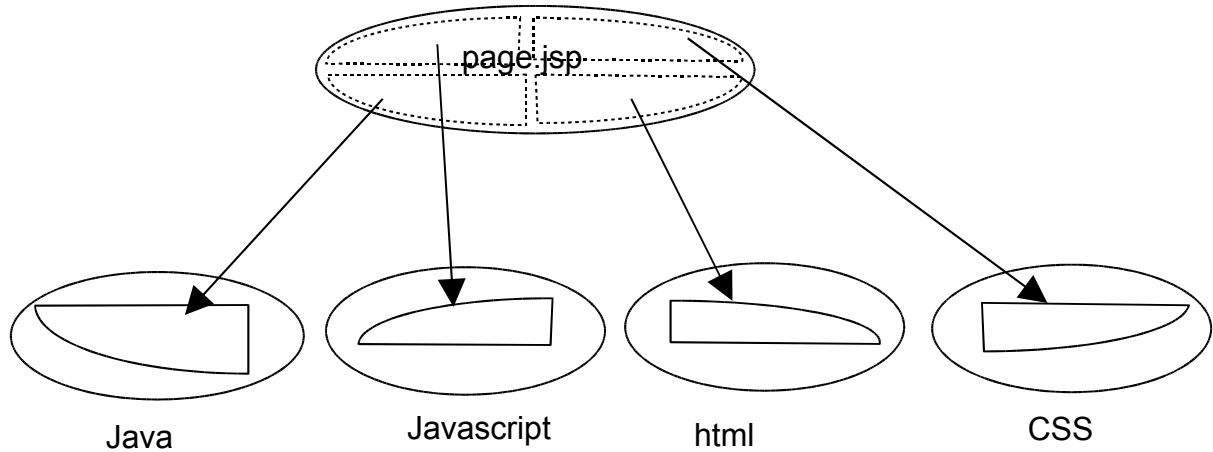
Currently we have very little in our language theories and architectures which attempt to bridge the gap between languages and improve compatibility, or evolvability. But what is important, and what are we trying to achieve with creating an evolvable language?

The first and most important aspect to which we are currently bound, is that a source file must match a known specification. Each source code developed for a compiler must fit within the set of the designed syntax. The following example shows that foo.java and bar.java are valid programs that fit within the set of the java language specification. However, bad.java is a program which does not fit the java language specification and is therefore not valid java.



If we are to create evolvable languages, then we need to find ways to decouple the one to one relationship between all source code files and language specifications. That is, how can we create new concepts in source codes and then later adapt (or teach) our compilers to understand those specific new concepts. In effect allowing the source code and compilers to evolve separately.

One solution to the problem of being unable to mix language specifications is to create source files which contain multiple languages. The example of a JSP page which contains multiple languages is like the following diagram. Each language is defined separately, and each section of the code must fit within its respective specification. The borders of each specification are clear and interaction between each system is minimal.



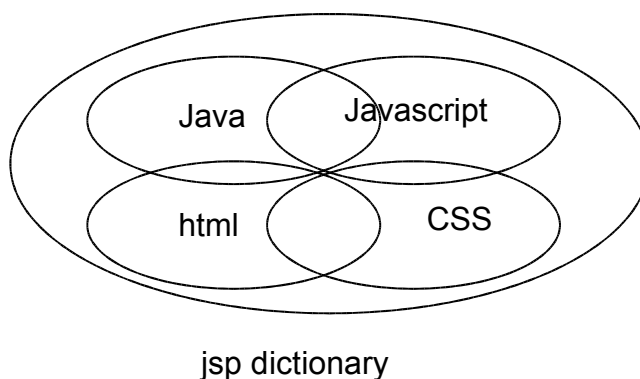
Problems of each language may be better solved using a combination of language elements. Currently the borders between languages are strict and coarse grained, forcing elements of languages to be re-implemented in each language. For instance both Java and Javascript implement their own basic expression elements.

To create an evolvable language we need to:

- Allow elements of each language to be re-used.
- Create a fine grained description of elements which allows language interaction.
- Allow fine grained interaction between compilers.

To achieve the ability of fine grained descriptions requires that we describe our languages in such a way that they can be re-used. Using the human language approach of a dictionary of terms provides a perfect analogy. Any term can be added to a dictionary to describe a language concept. The dictionary can then be used as a reference in which to check if the concept is understood by the dictionary.

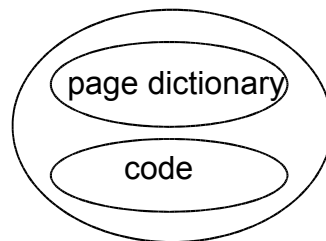
Using the dictionary approach and our previous example, we might create a specification which contains all the required elements. The single dictionary contains a description of each concept from all languages found in our page.jsp example earlier.



The intersection points are those elements which are shared and re-used between languages. The page.jsp is a file which partially matches elements of all of the specifications.

The ability to join and re-use elements is one feature of an evolvable language. The problem is that in a similar way to current languages, it creates a restriction that the source file must still match the published dictionary. While we have improved the ability for languages to share constructs we have not improved the ability for the source file to evolve.

One possible solution is to include a dictionary of terms with the source file. This allows the source file to describe the elements it uses without any external specification. In essence the source file describes its own code.



page.jsp containing its own dictionary

This concept can be compared with an XML file which contains its own XML Schema. The code contained in the file matches the specification of its own page dictionary. No other external reference is required. However, for this page.jsp source code to be useful its code must be able to be read and understood by a compiler or interpreter. The “page dictionary” must still be a subset of the “jsp dictionary” defined earlier.

By allowing the source code to contain its own dictionary and be matched against its interpreters we are able to decouple the source code specification from the compiler. This decoupling allows the source code to evolve more freely from the compiler.

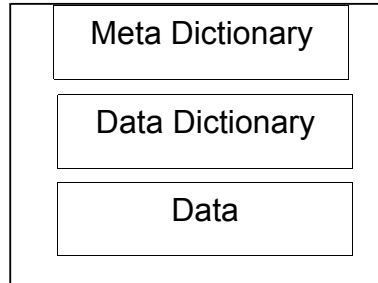
These concepts can not be easily achieved using current lexer and parser technology. The page.jsp example would need to create a dictionary of concepts which captured object oriented language concepts, hypertext markup language concepts and cascading style sheet concepts. The solution we propose is based on capturing these concepts using a binary dictionary of concept descriptions captured with and describing the contents of a source file.

## The Solution Technology

The method we present is based on a new data format originally designed for data communications. This new format is called Argot, and is also developed by the author. Argot is the subject of a paper of its own and for this reason we have only provided a brief description of the elements of Argot. Argot was originally designed for data agreement in distributed systems. However, applied to languages provides the required elements to create evolvable languages.

Argot takes a unique approach to the problem of data format compatibility by removing the need for a specification to be agreed upfront. Argot instead transports the data description as a component of the data itself.

Argot uses a dictionary to give meaning to the data. A single Argot file or message contains data, the data dictionary which describes the format of the data, and a meta dictionary which describes the format of the data dictionary.



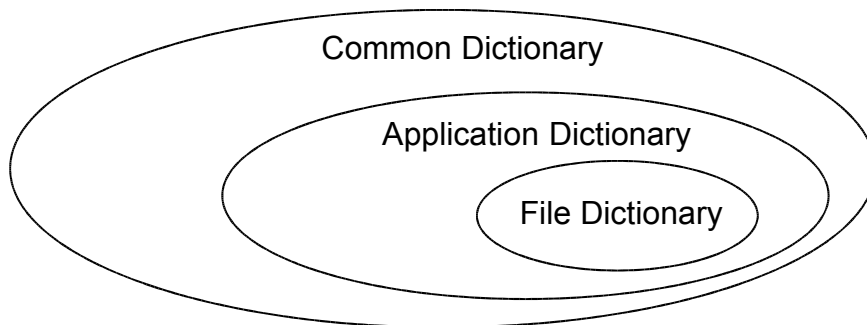
Argot file format

The data dictionaries contain the format specification of each type of data used, in an equivalent way to a language dictionary that contains descriptions of words. Only data that has been described in the data dictionary portion of the file can be encoded into the data portion of the file. The Meta Dictionary contains a description of each data type used to describe elements of the data dictionary. A core section of the Meta Dictionary contains twenty four type descriptions which are described using only definitions of those data types. This creates a purely self-referencing and self-describing data file with no external dependencies.

By removing these external dependencies, Argot allows systems and devices to communicate data in evolving formats, while removing many inherent problems. Data formats may be as complex or as simple as an application requires, and may be extended without removing meaning for other applications that share the data. These features were designed with distributed systems, data storage, embedded applications, and proprietary web applications in mind.

An Argot designer uses the concept of a common dictionary to choose the required data types for an application. The common dictionary is able to grow over time and can contain information from a variety of domains. Each concept in the dictionary is defined within the context of other types in the dictionary. After the data types have been chosen from the common dictionary, an application dictionary is created. The application dictionary is a subset of the common dictionary and only includes the data types the application needs to communicate.

When the application creates an Argot enabled file, the file itself contains the complete description of the data being sent. The file's dictionary is a subset of the data types of the application dictionary. In this way each application and file in a system contains all the knowledge about the information it is able to process or data it contains.



The receiver of an Argot enabled file is able to read the dictionary and compare the data types of its own dictionary with that of the files. Once the types of the file dictionary have been matched with that of the application reading the file, the data can be read. This completely removes the need for a static common domain schema. Each application and file in effect contains its own schema.

The ability to compare dictionaries and ensure that the formats are compatible gives Argot its flexibility. The use of a common dictionary allows concepts to be updated and changed without the versioning consequences found with externally defined schemas such as used in programming languages or XML Schemas.

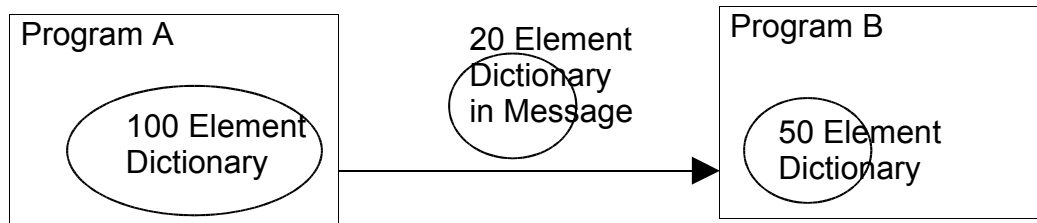
When a new concept is introduced into a system, the programmer adds the additional new code into the applications that must process the new concept. The Argot library allows the new marshaling code to be added without disrupting other parts of the application.

Argot and its dictionary format are binary. This ensures the most flexibility and only a small overhead for dictionary information during communications. The binary nature of the dictionary and data also allows the Argot library reading the data to be small, making the library and format appropriate for embedded applications.

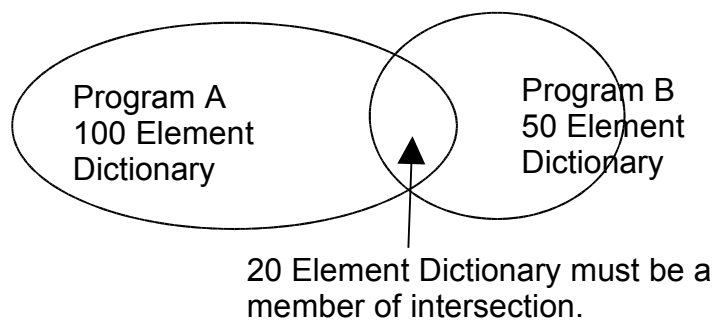
Argot allows a file or application to contain its own specification. This removes the need for a file to directly adhere to external schemas. As a direct result, two systems, which have previously had no contact, are able to discover dynamically if they have common data types to allow communications between them.

The most interesting feature of Argot is its ability to store a full binary description of the data along side the data in a file. It uses a dictionary approach, only storing the descriptions of data used in the file. As described, this creates some interesting properties for data agreement between systems.

A program which creates an Argot file has access to a set of types in a dictionary - each type with its own description. Used in communications, Argot allows systems that have developed independently to communicate. In the following example, a program A is sending a message to Program B. Program A contains a dictionary reference of 100 elements. Program B contains a dictionary reference of 50 elements. The message contains 20 elements which are a subset of Program A's dictionary.



This can be re-illustrated using the following set theory:



As is shown, for program B to understand the message sent from program A, the two dictionaries must contain a common set of elements. The 20 elements contained in the message must be a member of the intersection between both sets.

Each program can ensure strong typing of each type of data received. Any change to the elements used in the message, if not present in the intersection, will be picked up by Program B when reading and comparing dictionary elements.

The examples shown are examples of how Argot operates on a small scale. However the benefits arise when Argot is used on a much larger scale. A global dictionary may contain a large set of commonly used concepts which can be communicated. Much like normal language, a field of endeavour may create a more specialised set of concepts which it uses to communicate.

## Applying Argot to Language Evolution

As already stated the problem is that a language can't change to meet new needs. This makes it difficult to support new concepts. Argot allows mixing of concepts between languages and data while still maintaining a strong specification. Argot however only defines the format of the information. It does not define the meaning or semantics of the information. For example, Argot defines an ordered set of types to be recorded to a file as:

```
meta.sequence:
{
    meta.array(
        @u8["size"],
        @meta.expression["type"]
    )
};
```

The meta.sequence is defined by how it is stored in the file. The sequence is an array of other expressions where a meta.expression is abstract and can be represented by a number of concrete representations. Argot uses this description to compare structures. It does not attempt to represent meaning.

To define the concepts we use in programming languages in Argot requires capturing the syntax of each of these concepts. We present here some basic language elements in a hypothetical language called Spark. Spark is designed to be a very basic programming language to test the theories of using Argot as a language representation. It builds on Argot's basic data types to create a scripting language. A portion of Spark's Argot dictionary might look as follows:

```
// The base expression is abstract.
Spark.expression: abstract;

// Define basic addition between two expressions.
Spark.math.addition:
{
    Spark.expression["leftside"],
    Spark.expression["rightside"]
};

// Create a mapping between abstract expression and addition.
Spark.expression#addition: map( #Spark.expression, #Spark.math.addition );

Spark.if:
{
    Spark.expression["condition"],
    Spark.statement["true"],
    meta.optional( Spark.statement["false"] )
};

// Define a name as a short utf8 string.
Spark.name: u8utf8;
Spark.type: Spark.name;

Spark.variable: { Spark.name, Spark.type };

Spark.block: meta.array( u16["size"], Spark.expression );
Spark.program: Spark.block;
```

The concepts shown are only a small fraction of the requirements for a basic language. However, even the small number of expressions demonstrates how similar the Argot definitions are to a BNF. These element descriptions are a basic text representation of what is captured in an Argot binary dictionary file. The relationship of the description is much closer to that of an Abstract Syntax Tree. A compiler would simply use its dictionary to read the source file directly into an AST. From this point all traditional compiler theory and techniques can be used.

This does not specify the semantics of each type. This method does not remove the need for semantic meaning to be described in external documentation. By describing the syntax and allowing the editor or compiler to check the same syntax is being used, we are able to

assume the same semantic meaning. If a change to language semantics requires no change to the syntax, then a new type should be added to the dictionary to show the difference.

Given a basic set of language dictionary definitions we can describe the contents of a short program source file encoded with Argot. As described earlier the file will contain three sections: the meta dictionary; data dictionary; and data.

Meta Dictionary:

Contains the standard core data type descriptions which are used to describe the types used in the description of the types in the Data Dictionary.

Data Dictionary:

```
variableName: u8utf8; // short utf8 string for variables.
expression: abstract;
addition: { expression["left"], expression["right"] };
expression#addition: map( #expression, #addition );
expression#variableName: map( #expression, #variableName );
expression#u16: map( #expression, #u16 );
assignment: { variableName["assign"], expression["expression"] };
statement: abstract;
statement#assignment: map( #statement, #assignment );
block: array( u8["size"], statement["code"] );
```

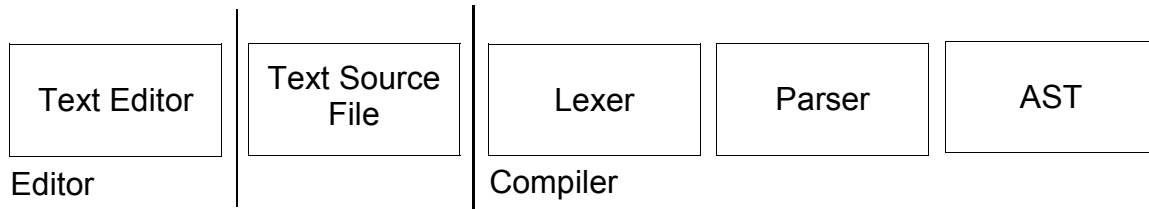
Data:

```
block
  assignment
    variableName: X
    u16: 13
  assignment
    variableName: Y
    u16: 17
  assignment
    variableZ
    addition
      variableName: X
      addition
        variableName: Y
        11
  assignment
    variableZ
    addition
      variableZ
      1
```

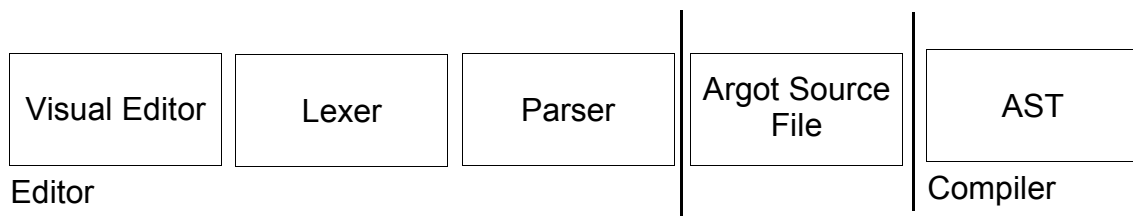
As the example shows, the description of every type used in the Data of the file is captured with its description in the Data Dictionary section of the file. The whole language dictionary is not stored with the file; only those structures that are utilised are stored. In many cases we expect that the specification of the data will be larger than the actual data, however, given the cost of storage this is unlikely to be an issue.

Creating a binary source file puts additional requirements on the way we create editors. One aspect already mentioned is that languages should build on concepts already known by developers. Obviously developers will not be able to edit or create binary source files themselves. To provide the developer an experience that they are comfortable with, we are able to design editors using traditional lexer and parser concepts.

A traditional developer, programming, edit and compile workflow includes the following:



An Argot based developer, programming, edit and compiler workflow will reshuffle to create the following:



A visual editor may include a lexer and parser that present the source file as a traditional looking language file. The editor saves the file and dictionary to the Argot Source File. The compiler uses its own dictionary to read the Argot source file directly into an AST directly.

Pushing the lexer and parsing back into the editor provides the user with an experience more like a traditional programming environment. Modern editors such as Eclipse already contain code folding and other elements which distance the original text based source code. The same ability to display source code as text in a way the user expects also allows the editor to display the same code in different forms.

In effect, the editor also holds the Argot source file in an AST. This provides more opportunities for refactoring tools to be created in editors. It may also provide more opportunities for optimisations to be performed on the original source code.

As discussed there are already a large number of languages which hold a common set of concepts. We envisage that parsers would be able to capture these languages and convert them into an Argot based format. Of course every language contains concepts which are unique and finding a single common dictionary would not be possible. However by finding the elements which are common, we can begin to create well defined concepts which can be reused between languages.

By allowing the mixture of visual and code based editors, the development of the original page.jsp may look very different in an Argot editor. The editor is likely to visually display page elements while still providing text based editing capabilities for traditional code elements. While we have not yet explored this area, we suspect that many of the techniques used to develop code and data browsers will be useful in the development of Argot based language editors.

Obviously one area that has become very important in recent times is XML. XML Schema is an important area that defines the valid syntax of data structures. We are currently also developing a program called Axle, which creates Argot based dictionaries from XML Schemas. This will eventually allow any XML based structure to be embedded within our language source code, effectively recreating the JSP page concept presented earlier.

### **Argot Opportunities**

The original concept for using Argot for language evolution came from a virtual serialization computer technique used in a remote procedure call method. The method used Argot to describe byte codes of a virtual machine and send those byte codes to a remote computer for execution. These byte codes provide the same concept of a virtual machine as does Java. An obvious extension of the evolution of languages is to create a full set of Argot dictionary elements of a virtual machine. In this way the user can combine both low level virtual machine instructions with high level language concepts in a single file.

Creating and developing a virtual machine that understands a set of Argot based byte codes could create opportunities for evolvable virtual machines. One area this may prove useful is specialised applications such as 3D environments. The virtual machine can be extended to understand a new set of specific graphics and 3d matrix math instructions. A programmer is able to embed these instructions directly into their language. The compiler can be designed to output either the embedded instructions, or convert the instructions into a more generic set of emulation instructions. This obviously blurs the line between compiler and emulation where the virtual machine becomes a device which interprets more specific instructions; whereas a compiler is one that interprets higher order instructions.

We have also found that the elements which describe the base Argot syntax is similar to that of the Spark language. In effect, the elements of Argot are the core of a language. We expect that the language elements we presented for Spark may eventually merge with the underlying Argot data marshalling model. Eventually the Argot data model will be defined using a subset of a larger programming dictionary of elements.

Already discussed is how we can use the dictionary based system to create and evolve programming languages and virtual machine byte code. An area not yet discussed or investigated is macro concepts; the concepts for higher level language requirements. Describing architectural structures and constraints, workflow based concepts involving people and processes, or describing deployment of software across clustered or parallel environments. We believe the Argot approach allows all of these concepts to be captured and evolve building a language on top of already defined dictionaries.

### **Future Directions**

This paper provides a rough sketch of the potential for Argot to be used to create evolvable programming languages. There is still a large amount of work to be done in studying how we can create extensible compilers, extensible editors, and various other extensible tools to support our ability to extend languages and source files. We are excited by the potential of these future possibilities, and you can expect a more concrete deployment of these ideas in the near future.