Adenine: A Metadata Programming Language

Dennis Quan DQUAN@MIT.EDU
David F. Huynh DFHUYNH@AI.MIT.EDU
Vineet Sinha VINEET@AI.MIT.EDU
David Karger KARGER@THEORY.LCS.MIT.EDU
MIT Artificial Intelligence Laboratory, 200 Technology Square, Cambridge, MA 02139 USA

1. Introduction

Haystack (Huynh *et al.*, 2002), our personal information repository, uses a shared metadata store and a system of agents for helping the user manage his or her information. Agents use the shared metadata store for storing the user's information, their own state, as well as interface and endpoint information on how to contact each other.

Metadata in the Haystack environment is expressed according to the Resource Description Framework (RDF) (RDF, 1998). In essence, RDF is a format for describing semantic networks or directed graphs with labeled edges. Nodes and edges are named with uniform resource identifiers (URIs), making them globally unique and thus useful in a distributed environment. Node URIs are used to represent objects, such as web pages, people, agents, and documents. A directed edge connecting two nodes expresses a relationship, given by the URI of the edge.

In a system such as Haystack, a sizeable amount of code is devoted to creation and manipulation of RDF-encoded metadata. We observed early on that the development of a language that facilitated the types of operations we frequently perform with RDF would greatly increase our productivity. As a result, we have created Adenine. An example snippet of Adenine code is given in Figure 1.

```
# Prefixes for simplifying input of URIs
@prefix : <urn:test-namespace:>

:ImportantMethod rdf:type rdfs:Class

method :expandDerivedClasses ;
rdf:type :ImportantMethod ;
rdfs:comment "x rdf:type y, y rdfs:subClassOf z => x rdf:type z"
          # Perform query
          # First parameter is the query specification
          # Second is a list of the variables to return, in order
          = data (query {
                    ?x rdf:type ?y
                    ?y rdfs:subClassOf ?z
          } (List ?x ?z))

          # Assert base class types
          for x in data
                    # x[0] refers to ?x and x[1] refers to ?z
                    add { x[0] rdf:type x[1] }
```

Figure 1. Sample Adenine code.

2. Syntactic Support for RDF

The motivation for creating this language is twofold. The first key feature is making the language's syntax support the data model. Introducing the RDF data model into a standard object-oriented language is fairly straightforward; after all, object-oriented languages were designed specifically to be extensible in this fashion. Normally, one creates a class library to support the required objects. However, more advanced manipulation paradigms specific to an object model begin to tax the syntax of the language. In languages such as C# and Python, operator overloading allows programmers to reuse built-in operators for manipulating objects, but one is restricted to the existing syntax of the language; one cannot easily construct new syntactic structures. In Java, operator overloading is not supported, resulting in verbose APIs being created for any object oriented system. Arguably, this verbosity can be said to improve the readability of code.

On the other hand, lack of syntactic support for a specific object model can be a hindrance to rapid development. Programs can end up being three times as long as necessary because of the verbose syntactic structures used. This is the reason behind the popularity of domain-specific programming languages, such as those used in Matlab, Macromedia Director, etc. Adenine is such a language. It includes native support for RDF data types and makes it easy to interact with RDF containers (*i.e.*, graphs of RDF) and services.

3. Portable Representation

The other key feature of Adenine is its ability to be compiled into RDF. The benefits of this capability can be classified as portability and extensibility. Since 1996, bytecode virtual machine execution models have resurged as a result of Java's popularity. Their key benefit has been portability, enabling interpretation of software written for these platforms on vastly different computing environments. In essence, bytecode is a set of instructions written to a portable, predetermined, and byte-encoded ontology.

Adenine takes the bytecode concept one step further by making the ontology explicit and extensible and by replacing byte codes with RDF. Instead of dealing with the syntactic issue of introducing byte codes for new instructions and semantics, Adenine takes advantage of RDF's ability to extend the "object code" graph with new instructions. One recent example of a system that uses metadata-extensible languages is Microsoft's Common Language Runtime (CLR). In a language such as C#, developer-defined attributes can be placed on methods, classes, and fields to declare metadata ranging from thread safety to serializability. Compare this to Java, where support for serializability required the creation of a new keyword called transient. The keyword approach requires knowledge of these extensions by the compiler; the attributes approach delegates this knowledge to the runtime and makes the language truly extensible. In Adenine, RDF assertions can be applied to any instruction.

4. Comparison with Lisp

These two features make Adenine very similar to Lisp, in that both support open-ended data models and both blur the distinction between data and code. However, there are some significant differences. The most superficial difference is that Adenine's syntax and semantics are especially well-suited to manipulating RDF data. Adenine is mostly statically scoped but has dynamic variables that address the current RDF containers from which existing statements are queried and to which new statements are written. Adenine's runtime model is also better adapted to being run off of an RDF container. Unlike most modern languages, Adenine supports two types of program state: in-memory, as is with most programming languages, and RDF container-based. Adenine in effect supports two kinds of closures, one being an in-memory closure as is in Lisp, and the other being persistent in an RDF container. This affords the developer more explicit control over the persistence model and makes it possible for agents written in Adenine to be distributed.

5. Syntax

The syntax of Adenine resembles a combination of Python and Lisp, whereas the data types resemble Notation3. As in Python, tabs are used to denote lexical block structure.

Adenine is an imperative language, and as such contains standard constructs such as functions, for loops, arrays, and objects. Function calls resemble Lisp syntax in that they are enclosed in parentheses and do not use commas to separate parameters. Arrays are indexed with square brackets as they are in Python or Java. Also, because the Adenine interpreter is written in Java, Adenine code can call methods and access fields of Java objects using the dot operator, as is done in Java or Python. The execution model is quite similar to that of Java and Python in that an in-memory environment is used to store variables; in particular, execution state is *not* represented in RDF. Values in Adenine are represented as Java objects in the underlying system.

Adenine methods are functions that are named by URI and are compiled into RDF. To execute these functions, the Adenine interpreter is passed the URI of the method to be run and the parameters to pass to it. The interpreter then constructs an initial in-memory environment binding standard names to built-in functions and executes the code one instruction at a time. Because methods are simply resources of type *adenine:Method*, one can also specify other metadata for methods. In the example given, an *rdfs:comment* is declared and the method is given an additional type, and these assertions will be entered directly into the RDF container that receives the compiled Adenine code.

The top level of an Adenine file is used for data and method declarations and cannot contain executable code. This is because Adenine is in essence an alternate syntax for RDF/XML. Within method declarations, however, is code that is compiled into RDF; hence, methods are like syntactic sugar for the equivalent Adenine RDF "bytecode".

6. Applications

About a quarter of the Haystack source base is written in Adenine. A significant portion of this source is user interface code, where screen layouts are completely described in RDF. Adenine is used to generate screen layouts dynamically, in event handlers for responding to user input, and for writing agents that process information, as most information in Haystack is described in RDF. Development on Adenine is ongoing, and Adenine is being used as a platform for testing new ideas on writing RDF-manipulating agents.

References

Huynh, D., Karger, D., and Quan, D. (2002). Haystack: A Platform for Creating, Organizing and Visualizing Information Using RDF. *Semantic Web Workshop, The Eleventh World Wide Web Conference 2002 (WWW2002)*. Honolulu, HI. http://haystack.lcs.mit.edu/papers/sww02.pdf.

Resource Description Framework (RDF) Model and Syntax Specification. (1999). http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

Acknowledgements

This work was supported by the MIT-NTT collaboration, the MIT Oxygen project, a Packard Foundation fellowship, and IBM.