A Simple, Lazy, Expression Evaluator

Andrew Cooke*

August 2006

Abstract

I explain the motivation for, design of, and experience using a simple domain-specific language in Java.

Contents

1 The Problem

1.1 The Remediation Service

The NOAO DMaSS (Data Management and Science Support) platform includes a Remediation Service. Separate instances of this service can be individually configured to perform fairly complex 'processing' of textual data. For example, one might be used to construct values for a general science data model given data from instrument-specific FITS headers[1] for different instruments.

The Tiny Workflow[1] provides a suitable framework for configuring remediation. Continuing with the example above, it would allow different 'flows' to be defined for each instrument, selecting the appropriate flow depending on the science data received.

Data within the remediation workflow are represented as name/value pairs. A simple library (similar to Java's `Properties` class) supports this abstraction, adding immutable values (both per-value, to protect system data, and per-collection, to allow more efficient processing).

Given this framework, the work of implementing the remediation service was reduced to writing various 'actions' that, when appropriately configured, modified the workflow data.

1.2 Remediation Actions

Initial analysis suggested that a sufficiently flexible system could be constructed from a set of very simple actions:

- Require that a given set of properties (names) is present.
- Rename a set of properties.
- Provide default property values.

*andrew@acooke.org

[1] FITS headers contain astronomy image metadata in a simple name/value format.

- Rewrite property values using regular expressions.

These different actions could be composed within the workflow (which included the ability to test for values by evaluating XPath expressions against an XML representation of the properties) to construct progressively more complex procedures.

Although these simple actions were easy to implement and configure individually, two problems soon became clear.

First, the Spring workflow configuration exploded into a mass of unsightly XML. I consider myself XML–tolerant, but felt that the verbosity involved in configuring non-trivial processes was excessive.

Second, I had forgotten the need to concatenate values. While this could be fixed this with another action, I was concerned that the set of actions was going to continue to expand, making configuration even more complex and opaque.

1.3 Configuration via Programs

One solution to the problems described above is to provide a single, more powerful 'general action' for processing data — ie. provide access to a programming language. The configuration for the action then becomes a small program in its own right.

Various languages are available to provide 'scripting' in Java (e.g: Groovy; Javascript in the next major JDK release). However, these might cause further problems.

Introducing a general language provides a lot of (unnecessary) functionality at one point in a layered architecture. There is a temptation to use it to solve unrelated problems. For easy maintenance/operation I would like to restrict the functionality available. For example, high–level control should remain in the workflow rather than being implemented in the same script used to manipulate values[2]

There is also a cost involved in using external packages. This is typically dismissed as unimportant compared to the 'obvious expense' of implementing a language from scratch. However, I felt otherwise.

[2]I later realised that the workflow is closely related to information in the Vocabulary Service[2]; this separation helped us exploit that commonality.

2 The Solution

2.1 Domain Specific Languages

In recent years there has been an explosion of interest in Embedded / Little / Domain Specific Languages (DSLs), particularly in the functional language programming community. This 'movement' has recognised that:

- Implementing a simple language is not difficult or expensive.
- Languages targeted at particular problems make good user interfaces.
- Careful language design can make the interface easy to control (it can be as restrictive or extensible as needed).

Although much of the literature emphasises the (undeniable) advantages of functional programming languages in supporting this paradigm, I felt that a similar approach should also be possible in Java.

2.2 Design Choices

A typical DSL builds on existing language support (e.g. combinator libraries, macros) or uses a dedicated recursive descent parser. Unfortunately, neither of these approaches would work in this case: Java does not have simple runtime compilation or the kind of features (e.g. high order functions) necessary for embedding; a typical hand-written recursive descent parser would be tediously verbose.

I considered using a parser library (GI[3] is very nice) but instead decided that it would be simpler to use a prefix syntax (like Lisp / Scheme) whose parsing is trivial.

Initial requirements involved only function evaluation. Ignoring function and variable definition removed the need for scoping or mutable data. Nor

did I need support for lists (at first this seems irrelevant, given the syntax, but it simplifies parsing by eliminating quotation).

However, some control flow was necessary. Learning from the behaviour of '?:' (taken from C language syntax) in some IRAF[4] tasks, which evaluates *both* sides of the branch, it seemed simplest to avoid such problems by using lazy evaluation (terms are calculated only as they are required).

I also chose untyped semantics (a.k.a. 'dynamic' or 'tagged' types), assumed arbitrary dynamic casts, and did not consider operator overloading.

Examples of the resulting syntax can be seen in section 2.6.

2.3 Parsing

The design described above is so simple it hardly warrants being called a 'language' - it is almost identical to the calculator example (terms and expressions) given in programming textbooks.

Program text is lexed using a simple state machine (a single Java class with a case statement over an `Enum` state). Single character look–ahead is sufficient; I used `nio.CharBuffer`'s 'mark' mechanism to provide this. There is a token for 'open', 'close', 'eof' and each atomic type (including 'name' for unquoted text).

The stream of tokens is assembled into an AST using a simple recursive descent parser that consumes tokens and returns AST nodes. This was implemented as a single class (the productions are private methods). Since the JVM does not have tail call optimisation I needed to worry about stack use; it is proportional to the maximum nesting depth in an expression which, for the intended use, is not a problem.

2.4 Evaluation

Evaluation is equivalent to a traversal of the AST, selecting (evaluating) only those nodes required by the semantics. This is not explicit in the implementation (there is no tree walker, for example). Instead, sub nodes are evaluated via the `Node.evaluate(Namespace)` method. Each node implementation is responsible for deciding which child nodes to evaluate in turn.

Types are identified ('tagged') by the subclasses of the AST `Node` class. The common superclass guarantees that future extension to include first class functions and lists will be painless (both already exist as `Node` subclasses, but the current operational semantics do not expose their dynamic creation to the user).

Functions and variables are provided via a `Namespace` interface. Typically, variables are read from the `Properties` interface described earlier, while functions include a standard collection that provides basic functionality. There is also an adapter for eager functions (the user programs to an interface that is provided with pre-evaluated arguments) so that simple functions for specific applications can be added without understanding how lazy arguments are implemented.

2.5 Generics, Reflection

I was pleasantly surprised at the level of integration possible between the language being implemented and the Java host. Figure 1 shows almost all the code necessary to implement literal values, including the base classes, support for dynamic casts and type errors, and the implementation class for string literals. Generics and reflection help us write code that can be used for all (literal) types while also helping leverage the support for these types in the host language.

The dynamic cast method (`as`) is a good example[3]. It is used in the example in figure 2.

For those not familiar with Java, the important aspects of figure 1 are that (1) most of the logic for literal types (everything except type conversions) is in a single type–safe generic class, `Literal`; (2) the code for dynamic type conversions is statically checked and type–safe; (3) the runtime check for type errors (in

[3] Although this implementation makes the addition of later 'third party' types difficult, since casts are encoded in each source class; a 'from' approach would be better, but with an uglier syntax, if this kind of extension were an important requirement.

```
public abstract class BaseNode implements Node {
    public abstract Node evaluate(Namespace namespace) throws Exception;
    public <Cast extends Node> Cast as(Class<Cast> clazz) throws TypeException {
        if (clazz.equals(StringLiteral.class)) {
            return TypeException.assertType(new StringLiteral(toString()), clazz);
        } else {
            return TypeException.assertType(this, clazz);
        }
    }
}

public abstract class Literal<Internal> extends BaseNode {
    private Internal value;
    public Literal(Internal value) {this.value = value;}
    public Node evaluate(Namespace namespace) throws Exception {return this;}
    public Internal getValue() {return value;}
    public String toString() {return getValue().toString();}
}

public class StringLiteral extends Literal<String> {
    public StringLiteral(String value) {super(value);}
    public <Cast extends Node> Cast as(Class<Cast> clazz) throws TypeException {
        if (clazz.equals(BooleanLiteral.class)) {
            return TypeException.assertType(new BooleanLiteral(getValue()), clazz);
        } else if ( ... ) {
            ...
        } else {
            return super.as(clazz);
        }
    }
}

public class TypeException extends EvaluationException {
    public TypeException(String msg) {super(msg);}
    public static <Type extends Node> Type assertType(Node node, Class<Type> clazz)
    throws TypeException {
        if (! clazz.isAssignableFrom(node.getClass())) {
            throw new TypeException("Expected " + clazz.getSimpleName()
                    + ", but got " + node.getClass().getSimpleName());
        }
        return clazz.cast(node);
    }
}
```

Figure 1: Generic annotations and reflection simplify integration between language and Java host.

the 'new' language, not Java) is made in a generic class (a static member of `TypeException`) using reflection. The ease with which a 'dynamic' language can be implemented in 'static' Java is surprising[4].

2.6 Examples

2.6.1 The 'If' Function

Figure 2 shows the implementation of a function that is equivalent to an 'if statement' in eager languages. For example

```
(if false (/ 1 0) (+ 1 2))
```

will return `3` — there will be no division by zero error because the `if` function does not evaluate the second argument if the first is false (`true` and `false` are parsed as literal boolean values).

The first argument is evaluated, cast to a boolean, and used to select the first of second argument. The `evaluate` method would be called from the s–expression, which itself implements `Node.evaluate` by evaluating the list head (typically a `Name` whose `Node.evaluate` retrieves the function from the `Namespace`), casting to a `LazyFunction`, and passing the list tail as arguments.

2.6.2 Basic Actions

Section 1.2 listed basic remediation actions. Here I show how the function evaluator can reproduce that functionality.

Exceptions in the evaluator are implemented as Java exceptions in the implementation. The evaluator is structured so that the evaluator automatically inherits the correct semantics.

Required name

```
(require name1 name2 ...)
```

The `require` function throws an exception if there is no binding for any of its arguments. If all arguments are bound it returns `()` (the empty list).

[4] I believe similar code would be possible in C#

Rename values

```
new-name = old-name
```

The simple evaluator does not bind values, so the syntax above, and the associated responsibility for assigning a new value, belong to the calling package.

Default values

```
name = (default name "value")
```

The `default` function evaluates and returns the first argument. If an exception is thrown during evaluation and the function has a further argument, the exception is discarded and the next argument evaluated.

Note that only the expression enclosed by `()` evaluated by the library described here. The final assignment to a value is the responsibility of the caller.

Regular expressions

```
(regexp firstword "^\\s*(\\w+).*" "$1")
(regexp lastword ".*?(\\w+)\\s*$" "$1")
(regexp firstletters "(\w)\w*\s*" "$1,")
(regexp dropcommas "(.*),$" "$1")
firstname = (firstword DTPI)
lastname = (lastword DTPI)
initials = (dropcomma (firstletters DTPI))
```

Regular expressions are not supported by the evaluator, but they show how easy it is for the caller to add extensions. Here `regexp` is a function, provided by the caller, that defines a Perl 5 regular expression in the caller's state as a side-effect. The caller then supplies the compiled expression via `Namespace` for the next call to the evaluator.

As before, assignment is the responsibility of the caller. The caller is also free to schedule evaluation so that, for example, the regular expressions are defined only once, when the system starts.

3 Conclusions

3.1 Experience

The initial implementation of this 'language' (lexer; parser; interpreter; unit tests) took about 20 hours

```
public class If extends LazyFunction {
    public Node evaluate(NodeList arguments, Namespace namespace)
            throws Exception {
        Iterator<Node> args = arguments.iterator();
        if (! args.next().evaluate(namespace).as(BooleanLiteral.class).getValue()) {
            args.next();
        }
        return args.next().evaluate(namespace);
    }
}
```

Figure 2: Implementing 'if' as a lazy function.

(two 10 hour days). As a result, the configuration for remediation was simplified significantly, using a tool whose power is limited, as required, but which could be easily extended in the future.

A careful choice of syntax and semantics allowed us to exploit techniques developed (or rediscovered) with significantly more powerful tools. Implementation time was short, allowing easy integration into our iterative (agile) development process. I believe that many of the approaches emerging from 'academic' functional programming (an emphasis on declarative approaches; little languages; a realisation that powerful modern languages make some heavyweight libraries obsolete) fit well with the agile approach.

Embedding a 'dynamic' language in 'static' Java was surprisingly easy (section 2.5). Similar ideas can be seen in the elegant generic form handling within the Spring MVC framework. In my opinion the power of generics and reflection are too often overlooked by Java programmers.

The main drawback to this approach is raw execution speed. A third party language that compiles to bytecode would be many orders of magnitude faster. For this application — evaluating simple expressions during remediation — this is not an issue; if it becomes important later then this solution is, of course, carefully encapsulated within a small number of interfaces, and easily replaced.

3.2 Optimisation

As an experiment, I modified the system to cache known results if they were constant. Since I make no assumptions about purity (in particular, functions could be implemented to have side effects and variables mutable) this was not completely trivial; however, it was simpler than I expected (one evening's work).

The modification consisted of: adding `Thunk` with the same interface as `Node`, but the ability to cache a value; extending `NodeList` to wrap `Node` instances in a `Thunk`; changing the return type of `evaluate` to return both a `Node` and an indication whether it could be cached (a signal to the surrounding `Thunk`); extending functions to include the necessary logic for propagating 'constness'.

In addition, to complete the work, I would have needed to make a distinction between pure and impure functions, and fixed and mutable variables (not difficult; this distinction is already present in the Properties library, but the code there would need to be made more generic, to handle functions as well as strings).

This work was discarded, however, because it seemed (i) too complex for any expected gain (in particular, intermediate objects were generated and discarded for each evaluation) and (ii) I could not see how to make the changes either optional or cost–free when used in an impure context.

3.3 Philosophy

Finally, I also feel that there is a hidden value in work like this: it gives real pleasure in what is otherwise an increasingly mechanical and uninvolving profession[5]. Happy programmers are better programmers.

References

[1] A. Cooke 2006; A Tiny Workflow in Spring: http://www.acooke.org/andrew/papers

[2] A. Cooke, A. Egaña, S. Lowry 2006; FITS Files and Regular Grammars: A DMaSS Design Case Study: http://www.acooke.org/andrew/papers

[3] GI (Generic Interpreter): http://www.csupomona.edu/ carich/gi/

[4] IRAF (Image Reduction and Analysis Facility): http://iraf.net

[5] Populated by technicians who use 'never re–invent the wheel' to justify over–complex solutions with fragile dependencies on rapidly evolving third–party systems; the consequent lack of knowledge about basic programming techniques makes their warning a self–fulfilling prophecy.