Diagnosing Java code: Designing extensible applications, Part 4

Examine how S-expressions provide a lightweight form of black box extensibility

Level: Introductory

Eric Allen (eallen@cs.rice.edu), Ph.D. candidate, Java programming languages team, Rice University

11 Dec 2001

In this installment of *Diagnosing Java Code,* author Eric Allen illustrates how S-expressions -- syntactic representations of lists of elements delimited by parentheses -- can be used to provide a useful and lightweight form of black box extensibility. The advantages of using S-expressions are discussed in the context of a particular example. Also, the author details the limitations of S-expressions and notes when they may not be the best fit for an application. After reading this article, you'll know when to use S-expressions to create black box extensibility. Share your thoughts on this article with the author and other readers in the discussion forum by clicking **Discuss** at the top or bottom of the article.

In last month's column, you saw that availability of underlying code needn't be a problem if you learn:

- How to spot configuration scripts
- How to choose which configurations to allow
- To recognize which contexts call for black box extensibility
- To weigh the added complexity of building in the extensibility
- That when you provide for configuration scripts, you're *actually* building a language.

You also learned that using S-expressions can be an effective means to quickly set up a configuration language, allowing for black-box extensibility of an application. We'll delve further into S-expressions in this article, and provide an example of how they could be used to quickly and easily set up such a configuration language for a particular application.

A bit about S-expressions

Recall that S-expressions are syntactic representations of lists of elements, delimited by parentheses. They come in three flavors:

- An empty list of elements
- A non-empty list of elements
- A single atomic element (such as a word)

S-expressions are quite useful for configuration languages because they are easy to parse. A general S-expression parser can be used to read data into a program, which can then check that the expressions also meet more specific syntactic constraints. In this way, we get all the benefits of parsing input -- such as early detection of erroneous input and added security -- without the added effort and overhead of writing and maintaining a conventional parser. Also, unlike the parsers constructed by parser generators, the error message output can be precise and very helpful in tracking down the source of a

Don't miss the rest of the "Designing extensible applications" series
Part 1: "Black box, open box, or glass box: which is right and when?" (September 2001)

Part 2: "Examine when, where, and how glass box extensibility works best" (October 2001)

syntax error.

Part 3: "Examine when, where, and how black box extensibility works best" (November 2001)

The advantage of "S" over XML

As I mentioned in the last article, many of the same benefits from using S-expressions can be gained through the use of XML-based configuration languages. The main advantage an S-expression-based configuration language has over XML is that it's extremely lightweight and fast to set up.

Also, in many cases, it's easier to read and edit an S-expression-based configuration script than an equivalent XML-based script. As we discuss some examples of S-expression-based scripts below, consider what they would look like in XML notation.

Example: Adding macro support to an editor

Let's suppose that we would like to add simple macro support to a text editor, allowing users to define complex sequences of primitive operations. We might even want to throw in support for looping or recursive constructs.

Here is one example of what such a macro might look like:

```
(define (cutAndPasteAtEnd)
 (sequence
  (cut HIGHLIGHTED_TEXT)
  (move-to END_OF_DOCUMENT)
  (paste CLIPBOARD))
```

Even though we haven't discussed the syntax and semantics of our configuration language, you can probably guess the intended meaning of this script: a sequence of *cut*, *move*, and *paste* operations on text, something that a user of the application might ordinarily do himself.

Writing down examples of the kinds of scripts you want your language to capture is a good first step in building your language. In fact, these examples can help to form a good unit test suite over your interpreter.

Following is an example of a more complex macro:

```
(define (find-and-replace target replacement)
 (move-to BEGINNING_OF_DOCUMENT)
 (while (not AT_END)
  (cut NEXT_WORD)
  (if (equals CLIPBOARD target)
      (paste replacement)
      (paste target))
  (move-to NEXT_WORD)))
```

Again, you can probably guess that this script represents an implementation of *find-and-replace* within the scripting language of the application. For those not familiar with Scheme-style syntax, let me point out a few things:

- All operations use *prefix notation*. This style makes parsing even easier, but certain operations, such as

`equals`, which are normally written as infix operators, look rather odd to the uninitiated.

- The *if* statement, as in most languages, has three parts: a *condition clause*, a *consequent*, and an *alternative*, written in that order. But, again, to make parsing easier, I've left out an `else` keyword to mark the alternative.

This example illustrates that a sufficiently expressive scripting language added to an application is a great way to expand extensibility. It allows for all sorts of new functionality to be added without modifying or even viewing the original source code.

Example: Determining the scripting language

To implement such a scripting language, it is first necessary to specify exactly what that language is. This means nailing down a syntax and semantics.

We can specify the syntax using BNF notation as follows:

```
<script> ::= (<definitions> <expression>)
<definitions> ::= <empty>
                | <definition> <definitions>
<definition> ::= (define (<var> <args>) <statement>)
<args> ::= <empty>
         | <var> args
<statement>   | (cut <name>)
              | (paste <name>)
              | (move-to <name>)
              | (sequence <statements>)
              | (if <expression> <statement> <statement>)
              | (while <expression> <statement> <statement>)
              | (return <expression>)
              | (<var> <args>)
<statements> ::= <empty>
               | <statement> <statements>
<name> ::= <var>
         | <constant>
<expression> ::= <name>
               | (not <expression>)
               | (or <expression> <expression>)
               | (and <expression> <expression>)
               | (equals <name> <name>)
               | (<var> <args>)
<var> ::= any word consisting of only letters and numbers
          (but starting with a letter).
<constant> ::= BEGINNING_OF_DOCUMENT
             | END_OF_DOCUMENT
             | AT_END
             | CLIPBOARD
```

A closer look

Notice that this language includes procedural definitions. They allow the user to abstract over a commonly used sequence of operations and then apply this abstraction at multiple points, passing in different arguments as values (akin to method definitions in the Java language). I'm defining a script as consisting of a sequence of these

BNF notation

BNF stands for *Backus Naur Form*, named for John Backus and Peter Naur who in 1959 first introduced a formal notation to describe the syntax of a given language (originally, it was for the

definitions followed by an expression.

A few things are missing in this simple language:

- *Let* expressions, used to bind variables inside a procedure; let expressions can be simulated with extra procedure definitions, but it's more convenient to include them as part of the language
- Assignment statements
- A static type system

In particular, adding a static type system would allow the interpreter to catch many errors before a script is run.

On the other hand, static types also make a language more verbose and, inevitably, they reject not just programs with real errors, but also some programs that would run just fine. For these reasons, you'll often see static types left out of scripting languages in which the programs tend to be short. We'll leave them out of the example, but there's no reason you can't add them in if you want.

description of the ALGOL 60 programming language).

BNF is not only used to describe syntax rules in notation, it is also commonly used by syntactic tools (with variants).

The meta-symbols BNF uses to describe syntax rules are as follows:

- The double colon (::) means "is defined as"
- The pipe (|) means "or"
- Angle brackets (< >) are used to surround category names.

The angle brackets distinguish syntax rules names (also called non-terminal symbols) from terminal symbols, which are written exactly as they are to be represented.

For more information on BNF, see Resources.

Crafting a three-phase parser

Once we are happy with our grammar for the language, we can start to write a parser for it. This is where the use of S-expressions is a big win. Instead of writing a conventional two-phase parser (tokenization and parsing), we can vastly simplify the whole process by adding an extra phase.

The extra phase occurs between tokenization and parsing into a syntax tree. It involves sectioning the input into internal representations of S-expressions. This sectioning process basically amounts to parenthesis matching, but doing so makes the parsing process much simpler.

Pulling representations from the stream

Let's suppose that we've used a tokenizer (like, for instance, `java.util.StreamTokenizer` or `StringTokenizer` for smaller inputs) to convert data into a stream of tokens in which each token is either a left parenthesis, a right parenthesis, or a word.

The most convenient way to manipulate this stream is as a stack. In Listing 5, I define an interface `StackI` that could be implemented with an adapter over any tokenizer we care to use. That way, we can concentrate on the structure of the program without worrying about the details of any particular tokenizer. We can then write methods to construct S-expression representations from this stream.

Essentially, the process involves parsing the first S-expression in the stream and then determining that nothing comes after this S-expression (since the entire program is just one big S-expression). Parsing an S-expression can be defined recursively, because the elements of a complex S-expression are themselves just simpler S-expressions:

```
import java.util.LinkedList;
import java.util.*;

class SExpParseException extends Exception {
  public SExpParseException(String msg) {
    super(msg);
  }
}



interface StackI {
  public Object peek();
  public Object pop();
  public Object push(Object o);
  public boolean empty();
}

abstract class SExp {

  public static final String LEFT_PAREN = "(";
  public static final String RIGHT_PAREN = ")";

  public static SExp parseSExp(StackI tokens) throws SExpParseException {
    SExp nextSExp = parseNextSExp(tokens);
    if (tokens.empty()) {
      // The stack of tokens consisted of a single S-expression
      // (with possible subexpressions), as expected.
      return nextSExp;
    }
    else {
      throw new SExpParseException("Extraneous material " +
                                   "at end of stream.");
    }
  }

  public static SExp parseNextSExp(StackI tokens) throws SExpParseException {
    if (tokens.empty()) {
      throw new SExpParseException("Unexpected end of token stream.");
    }
    else { // tokens.pop() succeeds
      Object next = tokens.pop();

      if (next.equals(LEFT_PAREN)) {
        // The S-expression is a list. Accumulate the subexpressions
        // this list contains, and return the result.
        SList result = new SEmpty();

        while (! tokens.empty()) { // tokens.pop() succeeds
          next = tokens.peek();

          if (next.equals(RIGHT_PAREN)) {
            // We've reached the end of the list. We need only
            // pop off the ending right parenthesis before returning.
            // Since subexpressions were accumulated in the front
            // of the list, we must return the reverse of the list
            // to reflect the proper structure of the S-expression.
            tokens.pop();
            return result.reverse();
          }
          else {
            // Recursively parse the next subexpression and
            // add it to result.
            result = new SCons(parseNextSExp(tokens), result);
          }
        }
        // If we haven't yet returned, then we've reached the end
        // of the token stream without finding the matching right
        // paren.
        throw new SExpParseException("Unmatched left parenthesis.");
```

```
      }
      else if (next.equals(RIGHT_PAREN)) {
        // A right parenthesis was encountered at the beginning of
        // the S-expression!
        throw new SExpParseException("Unmatched right parenthesis.");
      }
      else {
        // The next S-expression is an atom.
        return new Atom(next);
      }
    }
  }
}

abstract class SList extends SExp {
  abstract SList reverse();
}

class SEmpty extends SList {
  public String toString() {
    return "( )";
  }

  SList reverse() {
    return this;
  }
}

class SCons extends SList {
  public SExp first;
  public SList rest;

  public SCons(SExp _first, SList _rest) {
    this.first = _first;
    this.rest = _rest;
  }

  SList reverse() {
    SList result = new SEmpty();
    SList elements = this;
    while (! (elements instanceof SEmpty)) {
      result = new SCons(((SCons)elements).first, result);
      elements = ((SCons)elements).rest;
    }
    return result;
  }
}

class Atom extends SExp {
  public Object value;

  public Atom(Object _value) {
    this.value = _value;
  }
}
```

Defining classes for syntax-tree parsing

Now, compared to parsing a raw token stream, parsing an S-expression into a syntax tree is a breeze. But in order to do so, we'll want to have separate classes defined for each syntactic construct in our grammar:

```
import java.util.LinkedList;
```

```
abstract class SyntaxTree {
  public abstract Object accept(SyntaxTreeVisitor that);
}

class Script extends SyntaxTree {
  LinkedList definitions;
  Expression body;

  public Script(LinkedList _definitions, Expression _body) {
   this.definitions = _definitions;
   this.body = _body;
  }

  public Object accept(SyntaxTreeVisitor that) {
    return that.forScript(this);
  }
}

abstract class Statement extends SyntaxTree {}

class CutStatement extends Statement {
  Name name;

  public CutStatement(Name _name) {
    this.name = _name;
  }

  public Name getName() {return this.name;}

  public Object accept(SyntaxTreeVisitor that) {
    return that.forCutStatement(this);
  }
}
...

abstract class Expression extends SyntaxTree {}
...

abstract class Name extends SyntaxTree {}
...

abstract class SyntaxTreeVisitor {
  public abstract Object forScript(Script that);
  public abstract Object forCutStatement(CutStatement that);
  ...
}
```

And so on. We need an abstract class for every non-terminal symbol in our grammar, and a concrete class for every form of that non-terminal. We'll also want to define visitors over this hierarchy of classes.

I've provided only a handful of the code that needs to be written to give you the idea. Extending it is straightforward. Code like this is a great candidate for automated code generation.

Recursively defining parse methods

After we've defined all these classes, we can recursively define parse methods for each syntactic construct: `parseStatement`, `parseExpression`, for example.

Each method will take in an S-expression. Its body will consist of a large `if-then-else` statement that checks the first element of an `SExp` and determines to which syntactic construct it corresponds. At that point, we simply check that the form of the `SExp` corresponds to a valid form for that construct (for example, that an `if`

statement has three components: an expression and two statements) and call the appropriate constructor, parsing the subcomponents recursively.

For example, Listing 6 shows how we could parse an `if` statement:

```
parseStatement(SExp sExp) {
  ...
  }
  else if (sExp.nth(0).equals("if") && sExp.length() == 4) {
    return new IfStatement(parseExp(sExp.nth(1)),
                           parseStatement(sExp.nth(2)),
                           parseStatement(sExp.nth(3)));
  }
  ...
}
```

At the end of this `if-then-else` statement is an `else` clause corresponding to the case where the S-expression matched none of the valid forms of the syntactic construct. In this case, a `SyntaxError` is thrown along with an appropriate error message.

The next step: Evaluation

After a script has been parsed into this form, we can easily implement other phases of the interpretation process. If our language included a static type system, this would be the place to include the type checker.

Also, this would be the place to include checkers for any other language constraints. For example, if our scripting language included a class hierarchy, we would want to check that the hierarchy contained no cycles.

A nice way to implement these various phases is by using a visitor over our syntax trees for each one. That way, all of the code for a particular phase is contained in one place. Furthermore, it is easy to add in extra phases -- we just write another visitor and include it in the sequence. None of the other classes need to be modified in any way.

But in our example language, there are no such added constraints, and we can move on to the final phase of interpretation: evaluation. Like the other phases after parsing, this phase can also be implemented as a visitor over syntax trees, and I heartily recommend doing so.

Each `for` clause in the visitor will describe how to evaluate program constructs of a particular form. Evaluation of the primitive operations in our language will correspond to method invocations on the supporting application:

```
class Evaluator extends SyntaxTreeVisitor {
  Application app;

  public Object forCutStatement(CutStatement that) {
    app.cut(that.getName());

    // A VoidObject is returned as the result of evaluating
    statements, to meet the signature of the for methods.
    return new VoidObject();
  }
  ...
}
```

As for the more complex operations, we can rely on the underlying program constructs in the Java language to easily implement these operations. For example, here is how we could implement the `if` and `while` constructs:

```
  public Object forWhileStatement(WhileStatement that) {
    while (that.getTest().accept(this).equals(new Boolean(true))) {
      that.getBody().accept(this);
    }
    return new VoidObject();
  }

  public Object forIfStatement(IfStatement that) {
    if (that.getTest().accept(this).equals(new Boolean(true))) {
      that.getConsequent.accept(this);
    }
    else {
      that.getAlternative.accept(this);
    }
    return new VoidObject();
  }
}
```

Read the fine print

Now we're at a point where interpretation of a script in our language will involve simply reading in the file (or other stream) containing the script, then processing it through the phases described in this article.

Both users and developers of our application will be able to extend the application in all sorts of ways without ever touching the source code. So there you have it: black box extensibility via S-expression-based languages.

This is the last article in this four-part mini-series on adding extensibility to an application. I should stress again that these techniques are like sharp knives -- they cut both ways. They can be a powerful means of efficiently reusing code, but they are also quite dangerous if you use them too indiscriminately and quickly add extensibility -- the complexity of your application can balloon out of control. Be careful out there!

Resources

- Participate in the discussion forum.
- The World Wide Web Consortium (W3C) offers an exhaustive source of information on XML.
- The *developerWorks* XML zone is an award-winning source of technical content for XML developers.
- The W3C offers eight notational conventions on using BNF notation.
- A short history and introduction to BNF notation can be found here.

- Ronald Rivest, professor of electrical engineering and computer science in MIT, discusses S-expressions and their use in various security applications, particularly in use in the SDSI (Simple Distributed Security Infrastructure), a new design for a public-key infrastructure.

- The JUnit Web site provides links to many interesting articles from a multitude of sources that discuss program testing methods.

- Christoph Czernohous's two-part series "Bank on it: Introduction to J/XFS" (*developerWorks*, August 2001) introduces and addresses integrating Extensions for Financial Services for the Java platform (J/XFS) into existing systems.

- Read all of Eric's *Diagnosing Java code* articles.

- Find more Java technology resources on the *developerWorks* Java technology zone.

About the author

Eric Allen has a bachelor's degree in computer science and mathematics from Cornell University and is a PhD candidate in the Java programming languages team at Rice University. Before returning to Rice to finish his degree, Eric was the lead Java software developer at Cycorp, Inc. He has also moderated the Java Beginner discussion forum at JavaWorld. His research concerns the development of semantic models and static analysis tools for the Java language, both at the source and bytecode levels. Eric has also helped in the development of Rice's compiler for the NextGen programming language, an extension of the Java language with generic run-time types. Contact Eric at eallen@cs.rice.edu.
