The Java Syntactic Extender (JSE)

Jonathan Bachrach
Artificial Intelligence Laboratory
Massachussetts Institute of Technology
Cambridge, MA 02139
jrb@ai.mit.edu

Keith Playford
Functional Objects, Inc.
86 Chandler Street
Somerville, MA 02144
keith@functionalobjects.com

ABSTRACT

The ability to extend a language with new syntactic forms is a powerful tool. A sufficiently flexible macro system allows programmers to build from a common base towards a language designed specifically for their problem domain. However, macro facilities that are integrated, capable, and at the same time simple enough to be widely used have been limited to the Lisp family of languages to date. In this paper we introduce a macro facility, called the Java Syntactic Extender (JSE), with the superior power and ease of use of Lisp macro systems, but for Java, a language with a more conventional algebraic syntax. The design is based on the Dylan macro system, but exploits Java's compilation model to offer a full procedural macro engine. In other words, syntax expanders may be implemented in, and so use all the facilities of, the full Java language.

1. INTRODUCTION

A macro defines a syntactic extension to some core language and allows users to define the meaning of one construct in terms of other constructs. These declarations are called *macro definitions* and their uses are called *macro calls*.

Macros provide the power of abstraction where functional abstraction won't suffice, affording: clarity, concision, and implementation hiding. As an example consider a Java loop syntactic extension called `forEach` whose use would look like:

```
forEach(Task elt in tasks)
  elt.stop();
```

and whose expansion would be:

```
Iterator i = tasks.iterator();
while (i.hasNext()) {
  Task elt = (Task)i.next();
  elt.stop();
}
```

The use of `forEach` is more succinct than its expansion and hides implementation details that if changed would be difficult to update without syntactic abstraction.

Macros provide brevity which empowers users to *do the right thing* in situations that would otherwise be too taxing. Consider Sun's standard JavaTest framework which is very tedious from the point of view of test authoring. For example, it's very verbose to write robust tests and tests involving exceptions just go on and on. Furthermore, extra code must be written in order to be able to tell from a test log exactly where a failure occurred. In fact, because it's so long-winded to test for unexpected exceptions on a check-by-check basis like the first case, people don't tend to bother, causing test suites to bomb out at the first such failure. As an alternative consider a check macro whose uses would look like:

```
check foo.equals(bar);

check foo.equals(bar) throws NullPointerException;
```

and whose expansions would be respectively:

```
try {
  logCheck("foo.equals(bar)");
  checkAssertion(foo.equals(bar));
} catch (Throwable t) {
  unexpectedThrowFailure(t);
};

try {
  logCheck("foo.equals(bar) throws NullPointerException");
  foo.equals(bar);
  noThrowFailure();
} catch (NullPointerException e) {
  checkPassed();
} catch (Throwable t) {
  incorrectThrowFailure(t);
};
```

This macro makes the job of writing test suites much easier and encourages the test suite author to do the proper bookkeeping.

It is sometimes clumsy in Java to define data declaratively, because of the lack of optional arguments and restrictions about where array initializers can appear. For example, when defining a Lisp-style list object, it is very convenient to have a list function that would create a list with elements the same as the arguments. For example, it would be convenient to do:

```
f(list(x, y, z));
```

where `f` is called on a list with elements `x`, `y`, and `z`. More generally, allowing users to define a declarative syntax for defining data makes inputting, manipulating, and maintaining data much more manageable (see [15] for another example).

Macros also provide an effective vehicle for planned growth of a language (see [3]). In Guy Steele's OOPSLA-98 invited talk entitled, "Growing a Language" [11], he says "From now on, a main goal in designing a language should be to plan for growth." A large part of Lisp's success and longevity can be linked to its powerful macro facility, which, for example, allowed it to easily incorporate both Object-Oriented and Logic programming paradigms.

Beyond merely providing syntactic extension, allowing arbitrary computation during the construction of replacement phrases and more generally having syntax expanders generate and interoperate with Java code has a number of benefits:

- The analysis and rewriting possible in a syntax expander is no longer constrained by a limited pattern matching and substitution language.
- The pattern matching and rewrite rule engine does not have to be as complex and capable as it would otherwise have to be, since standard Java control and iteration constructs can be used along with it.
- It is possible to package and re-use syntax expansion utilities in the same way as any other useful Java code.
- Elements of the pattern matching engine are open to programmer extension.

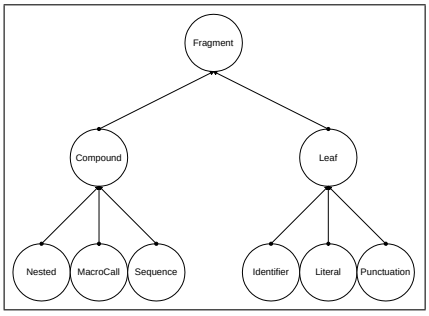
1.1 Overview

First, we describe a library which provides a code representation, called skeleton syntax tree (SST), source-level pattern matching and code construction utilities, and source code I/O. Next, we present parsing in the face of syntactic extension. Then, we discuss how the syntax expander mechanism can be layered on top of the previous building blocks and we discuss the syntactic extension execution model. Next, we discuss how our system ensures that variable references copied from a macro call and from a macro definition mean the same thing in an expansion. Then we discuss source level debugging in the face of macros. From there we discuss related syntactic extension systems. Next, we describe the current JSE implementation. Finally, we propose future directions.

2. SST AND FRAGMENTS

The fragments library provides a collection of classes suitable for representing fragments of source code in what we call a *skeleton syntax tree* (SST) form. Skeleton syntax trees may be constructed and pulled apart manually using the exported interface of these classes. Source level tools are provided for easier parsing and construction. I/O facilities permit the reading and writing of a textual representation of SST's.

A simplified fragment class hierarchy is shown below:

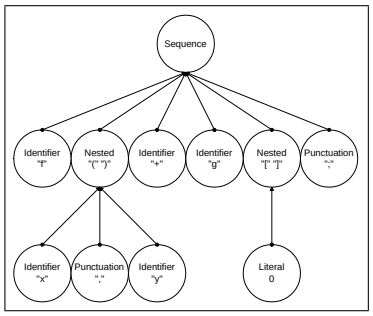


There are compound and leaf fragments to represent the distinction between nested and atomic syntactic elements. In general, a SST has fewer categories than a typical AST and instead represents the basic shapes and distinctions necessary for macro processing. One noteworthy compound fragment is the `macroCall` which earmarks a sequence of fragments for later macro expansion.

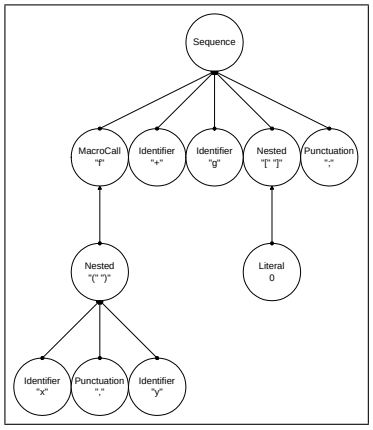
As an example of SST form consider the expression:

```
f(x, y) + g[0]
```

and a syntactic context in which none of the identifiers involved are associated with a macro category, the skeleton syntax tree is:



If, on the other hand, `f` were associated with a macro, then this would be the result:



2.1 Source Level Tools for Fragments

Working with Fragment classes directly is tedious and error-prone. Where possible, it is desirable for a programmers to be able to work in terms of the source code shapes they are already familiar with.

2.1.1 Code Quotes

An intuitive and accessible way of generating parameterized code is with *code quotes*. These allow a programmer to write a prototype of the form they want to generate, but with substitutions in place of the variable parts of the code. The evaluation of a code quote yields the skeleton form representation of code. Tokens within a code quote are substituted as themselves, apart from `?` which indicates a parameterized insertion. A `?` followed by a name substitutes the fragment bound to the local variable of that name (typically bound by pattern matching). A `?` followed by an expression in parens indicates the insertion of the fragment which is the result of evaluating the expression inside the parens. Evaluation of these substitutions occurs in the lexical environment in force where the code quote appears. Simple folding rules are applied to commas to make generating lists easier. As an example of code quotes and pattern variable evaluation consider:

```
Fragment test = #{ isOn() };
Fragment then = #{ turnOff(); };
return #{ if (?test) ?then };

==>

#{ if (isOn()) turnOff(); }
```

As example of code generation for code quotes, consider the expansion of the above code snippet:

```
Fragment test
   = Template.processTemplate
       (FragmentList.fnil
        .fpush(Template.substituteIdentifier("isOn"))
        .fpush(Template.processParens(FragmentList.fnil))
        .freverse());
Fragment then
   = Template.processTemplate
       (FragmentList.fnil
        .fpush(Template.substituteIdentifier("turnOff"))
        .fpush(Template.processParens(FragmentList.fnil))
        .fpush(Template.substituteSemicolon())
        .freverse());
return Template.processTemplate
  (FragmentList.fnil
   .fpush(Template.substituteIdentifier("if"))
   .fpush(Template.processParens
          (FragmentList.fnil.fpush(test)))
   .fpush(then)
   .freverse());
```

where `Fragment` is the most general code fragment class, and `FragmentList` is a Lisp-like list object used for containing sequences of fragments. `Template` objects represent code quotes and the `Template` class contains a number of facilities for their construction.

As an example of expression evaluation consider:

```
Fragment getter (IdentifierFragment id) {
  return new IdentifierFragment
    ("get".concat(id.asCapitalizedString()));
}

Fragment name = new IdentifierFragment("width");
return #{ x.?(getter(name))()}

==>

#{ x.getWidth() }
```

2.1.2 Nesting

Sometimes it is useful to be able to represent code quotes which when evaluated yield other code quotes. These nested code quotes can be used in situations such as macro-defining macros (see section 6.2 and [2]).

The evaluation of pattern variables and expressions is controlled through the use of multiple question marks: `??x`, `??(f(x))`, `???y`. Every `#{ }` wrapper introduces another level of nesting, and variables and expressions are evaluated when the number of question marks equals the current code quote nesting level. Otherwise the variable or expression is left unevaluated. For example,

```
return #{ #{ ?x } };                        ==> #{ ?x }
Fragment x = #{ a }; return #{ #{ ??x } }; ==> #{ a }
```

Multiple and delayed evaluation can be achieved using a combination of `?` and `()`'s. For example:

```
Fragment x = #{ y };
return #{ Fragment y = #{ a }; return #{ ?(??x) }; };

==>

Fragment y = #{ a }; return #{ ?y };

==>

#{ a }
```

Self Generating Code Quote

One amusing and instructive use of nested code quotes is in writing self generating programs. There is a long standing tradition of writing self generating code fragments [2], and nested code quotes can be used to express elegant solutions. Alan Bawden [2] presents a beautiful quasiquote solution written by Michael McMahon:

```
(let ((let ’‘(let ((let ’,let)) ,let)))
  ‘(let ((let ’,let)) ,let))
```

Following the same general format, a code quote solution would look like the following:

```
Fragment f = #{ #{ Fragment f = #{ ??f }; ?f; }; };
#{ Fragment f = #{ ??f }; ?f; };
```

We present the following stand-alone self generating Java program, because the usual self-generating goal is to write a complete stand-alone self generating program.

```
class selfish {
  static public void main (String args[]) {
    Fragment f
      = #{ #{ class selfish {
            static public void main (String args[]) {
            Fragment f = #{ ??f }; ?f.pprint();
            } } } };
    #{ class selfish {
         static public void main (String args[]) {
           Fragment f = #{ ??f }; ?f.pprint();
         } } }.pprint();
  } }
```

For comparison's sake, Klil Neori's stand-alone self generating Java program [14] is a typical Java solution:

```
class P{public static void main(String args[]){
String a="class P{public static void main
(String args[]){String a=;System.out.println
(a.substring(0,56)+((char)0x22)+a+((char)0x22)+a
.substring(56));}}"; System.out.println(a
.substring(0,56)+((char)0x22)+a+((char)0x22)+a
.substring(56));}}
```

where this is meant to be all on one line.

Although this isn't a typical Java application, we can still see how poorly suited vanilla Java is for manipulating program representations.

2.1.3 Pattern Matching

An intuitive and accessible way of expressing a parser is with patterns. This allows a programmer to write out the general shape of the expected input, but with pattern bindings in place of the variable parts of the code. The source-level parsing tools provided in the fragment library take this approach.

The primary parsing tool offered is `syntaxSwitch` (which is the moral equivalent of Lisp's `destructure-bind`). It implements pattern matching based on a simplified version of Dylan's constraint-directed rewrite rule system. It is like Java's `switch` statement and looks as follows:

```
syntaxSwitch (?:expression) { ?rules:* }
```

where rules look like:

```
case ?pattern:codeQuote : ?:statement
```

with each pattern looking like the construct to be matched. Patterns are augmented by pattern variables which match and lexically bind to appropriate parts of the construct. As an example consider:

```
Fragment input = #{ when (isOn()) turnOff(); };
syntaxSwitch (input) {
  case #{ when (?test:expression) ?body:statement }:
    return #{ if (?test) ?body };
}

==>

#{ if (isOn()) turnOff(); }
```

where the pattern variable `?test` binds to `isOn()` and `?body` binds to `turnOff();`.

The evaluation of a `syntaxSwitch` statement proceeds as follows. The input expression must evaluate to a valid fragment. During evaluation, this expression is tested against each rule's pattern in turn. If one of the patterns matches, its pattern variables are bound to local variables of the same name and the corresponding right hand side statement is run in the context of those bindings. If multiple patterns match, then the rule corresponding to the first pattern to match is chosen. No provision is made to flag rule ambiguities. If no patterns are found to match, then a `SyntaxMatchFailure` exception is thrown.

Pattern variables are denoted with `?` prefixing their names. Each pattern variable has a required constraint that restricts the syntactic type of fragments that it matches (e.g., `name`, `expression`, `body`). A constraint is denoted with a colon separated suffix (e.g., `?class:name`). A variable name defaults to the given constraint name (e.g., `?:type` is the same as `?type:type`). A wildcard constraint (`*`) matches anything and an ellipsis (`...`) is an abbreviation for a wildcard constrained pattern variable.

Pattern matching on a particular pattern proceeds from left to right. It employs a shortest first priority for wildcard variables. A simple backup and retry algorithm is used to try to find a match by binding the wildcard to more and more tokens. A largest first priority is used for matching non-wildcard variables with a similar backup and retry algorithm. Patterns match if and only if all of their subpatterns match. Non-pattern variable tokens appearing in a pattern match only the same token in the input. Simple folding rules are applied to commas to make matching lists easier.

To illustrate one possible code generation strategy for the simplest usage of `syntaxSwitch`, consider the following minimally cleaned up version of the generated code for the `syntaxSwitch` example from the above example:

```
Fragment __exp0 = input;
try {
  FragmentList __e1     = __exp0.getInsideFragments();
  SequenceFragment test = null;
  SequenceFragment body = null;

  FragmentList __fs2 = __e1.matchName("when");
  SplitList __tmp3   = __fs2.matchParens();
  FragmentList __ns4 = __tmp3.getInside();
  SplitList __tmp7   = __ns4.matchConstraint("expression");
  test               = new SequenceFragment(__tmp7.getBefore());
  FragmentList __fs6 = __tmp7.getAfter();
  __fs6.matchEmpty();
  FragmentList __fs5 = __tmp3.getAfter();
  SplitList __tmp9   = __fs5.matchConstraint("statement");
  body               = new SequenceFragment(__tmp9.getBefore());
  FragmentList __fs8 = __tmp9.getAfter();
  __fs8.matchEmpty();
  return Template.processTemplate
    (FragmentList.fnil
     .fpush(Template.substituteIdentifier("if"))
     .fpush(Template.processParens
            (FragmentList.fnil.fpush(test)))
     .fpush(body)
     .freverse());
} catch (SyntaxMatchFailure __e10) {
  throw new SyntaxMatchFailure();
}
```

where the code for creating `test` has been omitted. The basic strategy is to allocate all pattern variables up front and then step across the input matching against the pattern filling in pattern variables as the match proceeds. `SplitList` objects are used for pattern matching functions that need to return both the matching continuation point (`getAfter`) and the fragments matched (`getInside`). Failures are handled with Java's exception mechanism and potentially raised inside each pattern matcher utility. In the case of multiple `syntaxSwitch` rules, subsequent rules would be emitted in

a nested fashion inside nested `catch` code bodies. Wildcard pattern matching is handled with a `try/catch` inside a loop implementing the shortest first match policy.

2.1.4 Built-in Constraint Types

Several useful constraints corresponding to oft-used Java grammar terminals and non-terminals are provided as a starting point for users. The `name` constraint matches a single Java identifier. The `type` constraint matches a single well-formed type declaration. The `expression` constraint matches a single well-formed expression. The `statement` constraint matches either a single `{ }` block or a single semicolon terminated statement. In situations where the `{ }`'s are mandatory, it is necessary to put a `{ }` in the pattern so that it matches literally and to use the `body` constraint. The `body` constraint matches zero-or-more semicolon-terminated statements. Finally, the `switchBody` constraint matches a sequence of zero or more switch statement clauses.

2.1.5 User-defined Constraint Types

Beyond the builtin contraints, a mechanism is provided for users to define their own constraints. When a constraint is found in a pattern, the constraint implementation is resolved by looking up a class with the constraint name suffixed with `SyntaxConstraint`. By making a class implementing `SyntaxConstraint` available on the `CLASSPATH` and following the naming convention described above, programmers can introduce their own constraints. For example, the provided `statement` constraint is resolved to the class `statementSyntaxConstraint`. A canonical instance of a constraint class is consulted during pattern matching. For example, its admissibility predicate (implemented as a constraint class's `isAdmissable` method) is called during pattern matching with decreasing numbers of input fragments such that the constrained pattern variable matches the maximal number of input fragments.

A constraint definition itself minimally consists of a name to be used in pattern variable declarations and an admissibility predicate that takes a list of fragments and returns true if it matches exactly. A mechanism is also provided for making available Java grammar nonterminals as constraints.

3. PARSING AND SYNTAX EXPANSION

Now that we have a representation for code, and we have source level construction and parsing tools, we are ready to introduce syntactic extensions. First, we need to understand the context under which syntactic extensions are processed. We assume that compilation involves the following phases:

1. Parse source code into skeletal syntax tree
2. Recursively expand macros top-down
3. Create IR
4. Optimize IR
5. Emit code

Macro processing occurs during the first two phases from above. First, a surface parse of macro calls is performed to get a complete representation of calls in SST form. Second, this parsed form is handed to the macro expander for top-down pattern matching and rewriting. Each individual macro expansion replaces a given macro call with other constructs, which themselves can contain macro calls. This top-down process repeats until there are no macro calls remaining in the program.

3.1 Call Parsing

Macro calls are limited to a few contexts and shapes corresponding to existing Java syntactic contexts and shapes. Shapes serve to allow easy location of the end of a macro before handing it to the macro expander; shapes are a way to find "the closing bracket".

3.1.1 Call Macros

Call macros have function call syntax:

```
name(...)
```

and can occur in expression and statement position. The start of a function call macro is the function name and the end is the last matching argument parenthesis. One typical example is `assert`:

```
assert(a >= 0, "Underflow");
```

where an exception will be thrown unless `a >= 0`.

3.1.2 Statement Macros

Statement macros can occur in positions where Java statements occur and have the following basic shape:

```
... clause clause clause etc
```

where the leading ellipsis stands in for zero-or-more modifiers (like `public`, `private`, or `final`) and a clause is one of:

```
name ... terminator
```

where `name` is a clause name (with the first clause name being the name of the macro) and terminator is either a semicolon or a curly bracketed form. Examples of builtin Java statements are `class`, `interface`, `if`, `while`, `for`, `try`, `do`, and `switch`.

In order to parse a statement macro, JSE needs to know up front all of the clause names in order to determine where a macro ends, there being no common "end" marker. The beginning of the macro call is the first token past the last terminator and the end is the first terminator not followed by an associated clause name.

For example, to do the initial parse of `try`, the steps are something like:

1. Find the token `try` and resolve it to a syntax expander.
2. Get a list of clause words from the expander, in this case `exception` and `finally`.
3. Read the lead clause, `try { ... }`, in fragment form.
4. While the next identifier is a clause word, read the next clause.

Taking another example, `class` seems more complicated than `try`, yet from a form shape point of view, it's much simpler in that it has no clauses in this sense; all structure is internal to the lead clause, isolated within a `{ }` block.

3.1.3 Special Cases

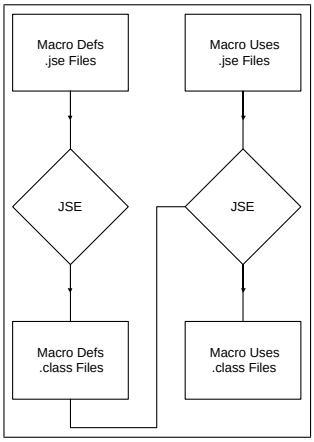
Java itself has a number of special cases that only work because they are tightly integrated into the parser. Method, field, and variable declarations in particular have no syntax words associated with them, and are disambiguated purely by context and construction. It's practically impossible to allow programmers to introduce constructs of similar status in a modular way, and no attempt is made to address this.

4. EXECUTION MODEL

Now that we have described our syntactic representation and our parsing model, we're now ready to actually describe our syntactic extension mechanism. Syntax expanders are classes that implement the `SyntaxExpander` interface and follow an associated naming convention. These classes coexist with the runtime Java source and class files of a program, but are actually dynamically loaded into the compiler or preprocessor on demand during a build.

Being implemented by classes, syntax expanders can either be defined at top level or live inside other classes. Syntax expanders defined at top level can be referred to unqualified. If defined within another class, the appropriate qualification must be used, as if referring directly to a nested static class.

The following figure depicts the way in which syntax extensions are defined and then used:



Macros are first defined and compiled by JSE producing `.class` files containing class representions of the defined macros. An application that uses these macros is then subsequently compiled by JSE augmented by demand-loaded class files corresponding to actually used macros. This then results in the final application class files.

It would be possible to manually write expander classes against the core code fragment manipulation API provided, but the idea is that a higher level `syntax` form be used (*see section 4.4.1*), which provides source-level pattern matching and code generation facilities and which expands into an appropriate class definition.

4.1 Package Scoped Syntax

Defining a top level syntax expander, then, allows new syntax to be defined with the same apparent status as built-in forms like `class`, `interface`, and `while`. For example, to get `forItems` syntax like:

```
forItems (ContactCard card in database) {
  // ...
}

==>

{
  Iterator _it = database.items();
    while (_it.hasNext()) {
      ContactCard card = (ContactCard)_it.next();
      // ...
    }
}
```

a top level class `forItemsSyntaxExpander` must have been defined and made visible through the `.java` file's import declarations.

When JSE sees an identifier in a context where syntax expansion may apply, it attempts to resolve that identifier, suffixed with `SyntaxExpander`, to a class according to the containing file's import spec and target program `CLASSPATH`. If found, the class is loaded into the language processor and the syntax following the leading name is parsed and expanded according to its definition. If no definition is found, the identifier is parsed as it would be in standard Java.

4.2 Class Scoped Syntax

Defining a nested syntax expander allows new class-specific syntax to be defined without polluting the top level package namespace. For example, to define utility syntax like the abstraction of the `open` operation for a `ContactDatabase` class one does the following:

```
ContactDatabase.withOpen
   (db = "/Documents/Contacts") {
 // ...
}

==>

{
  ContactDatabase db
    = ContactDatabase.open("/Documents/Contacts");
  try {
    // ...
  } finally {
    db.close();
  }
}
```

where a visible nested static class of `ContactDatabase` must have been defined called `withOpenSyntaxExpander`.

Processing is as above, except that identifier resolution only kicks in after the prefix `ContactDatabase` has been seen.

4.3 Execution Model Issues

Note that the execution model described so far precludes the use of a nested syntax definition within the code of its enclosing class. This restriction could possibly be removed

by having JSE extract and separately compile syntax definitions as they're encountered, but it is hard to get the scoping right in the resulting code without cooperation from the compiler.

Nesting a compile-time class within a run-time class has further potential for problems because, with a single set of imports, compile-time and run-time dependencies are conflated. Most of the time it is unlikely to be a problem, but if the enclosing class makes numerous static references, or worse if reference is made to something that can't be loaded at compile-time (e.g. if cross-compiling and JNI is involved), this could become an issue.

Finally, using a single class path for both looking up run-time classes and compile-time macro expanders could be generalized in order to better separate the two worlds. The simplest fix would be to provide a separate class path and compilation area for syntax expanders to avoid the above mentioned problems.

4.4 Syntax Expander Classes

A syntax expander definition consists of:

- Access specifier
- Macro name
- List of clause names
- Main rule set
- Extra class declarations

whose basic structure is defined in the `SyntaxExpander` interface and whose details are specified in a class definition implementing the `SyntaxExpander` interface. For example, the following is a class definition for the `forEach` syntax expander:

```
public class forEachSyntaxExpander
    implements SyntaxExpander {
  private static String[] clauseNames = {};
  public String[] getClauseNames() {
    return clauseNames;
  }
  public Expansion expand
      (Fragment fragments) throws SyntaxMatchFailure {
    syntaxSwitch (fragments) {
    case #{ forEach (?:type ?elt:name in ?:expression)
              ?:statement }:
      return #{ Iterator i = ?expression.iterator();
                while (i.hasNext()) {
                  ?elt = (?type)i.next();
                  ?statement
                } };
    } } }
```

where clause names are accessed with `getClauseNames` and the main rule is invoked with the `expand` method.

4.4.1 Syntactic Extension Definitions

In order to make it more convenient for defining syntactic extension we provide a syntax expander named `syntax` having the following form:

```
#{ ?modifiers:* syntax ?:name ?clauseNames:* {
     ?mainRules:switchBody
     ?definitions:*
   } }
```

It permits the `forEach` syntactic extension to be more succinctly written as:

```
public syntax forEach {
   case #{ forEach (?:type ?elt:name in ?:expression)
             ?:statement }:
     return #{ Iterator i = ?expression.iterator();
               while (i.hasNext()) {
                 ?type ?elt = (?type)i.next();
                 ?statement
               } };
}
```

5. AUTOMATIC HYGIENE

Two desirable properties of a Syntactic Extension system are *hygiene* and *referential transparency* [17] [16] [13] [5] which both roughly mean that variable references copied from a macro call and from a macro definition mean the same thing in an expansion. The mechanism attempts to avoid accidental collisions between macro bindings and program bindings of the same name.

In order to support these properties, each template name records its original name, lexical context, and specific macro call context. A named value reference and a binding connect if and only if the original name and the specific macro call occurrences are both the same. A new hygiene context is dynamically bound during expansion, but for more precise control, hygiene contexts can also be manually established and dynamically bound.

References to global bindings should mean the same thing as they did if looked up from within the originating macro definition. Unfortunately, this is hard to do in Java without violating security. We choose to force users to manually export macro uses.

Getting this to work well in the absence of compiler or language support is hard in places in Java, and simply impossible in others. When implemented as a preprocessor, avoiding shadowing local variables requires renaming and a detailed code walk.

Ensuring that any methods or other bindings referenced by code generated by a macro refer to the right things at a call point is difficult in Java. Again a full code walk and insertion of fully-qualified names is required. However, this only works if the things referenced are accessible at the call point: private or package access elements accessed from a public macro at a call point outside their scope present a problem. This was solved for inner classes in Java by having the compiler implicitly liberalize the access declarations of anything private used by the inner class. It's possible something similar could be done here, but then we're starting to substantially modify non-macro code in such a way as could violate security boundaries.

We may have to compromise here and guarantee correct resolution of references, but not necessarily their accessibility.

5.1 Circumventing Hygiene

Sometimes it is necessary to circumvent hygiene in order to permit macros to introduce variables accessible by the lexical context from which the macro was called. For example, imagine writing an if-like macro which executes the "then" statement when its test expression evaluates to a non-null value. Furthermore, as a convenience, it binds a special variable named `it` to the result of the test. The following is an example `nif` usage:

```
nif (moby()) return it; else return false;
```

and the following is the definition of `nif`:

```
public syntax nif {
  case #{ nif ?test:expression ?then:statement ?else:statement }:
    return #{ Object ?=it = ?test;
              if (?=it == null) ?else else ?then };
}
```

where `?=` defeats automatic hygiene and makes the prefixed variable available by way of variable references in the macro call.

6. A FEW EXAMPLES

In this section we present several JSE examples.

6.1 Parallel Iteration

This example expands on the previous `forEach` expander (in section 4.4.1) to include parallel iteration over multiple collections.

```
public syntax forEach {
  case #{ forEach (?clauses:*) ?:statement }:
          Fragment inits = #{ };
          Fragment preds = #{ true };
          Fragment nexts = #{ };
          return
            #{ ?(loop(clauses, statement,
                      inits, preds, nexts))};

  private Fragment loop
      (Fragment clauses, Fragment statement,
       Fragment inits, Fragment preds, Fragment nexts)
    throws SyntaxMatchFailure {
    syntaxSwitch (clauses) {
      case #{ }:
        return
          #{ ?inits while (?preds) { ?nexts ?statement } };
      case #{ ?:type ?:name in ?c:expression, ... }:
        Fragment newInits
          = #{ ?inits Iterator i = ?c.iterator(); };
        Fragment newPreds
          = #{ ?preds & i.hasNext() };
        Fragment newNexts
          = #{ ?nexts ?type ?name = (?type)i.next(); };
        return
          #{ ?(loop(..., statement,
                    newInits, newPreds, newNexts)) };
    }
  }
}
```

The basic strategy is to create an iterator state variable for each collection over which to iterate, to create a predicate that determines when any one of the collections is exhausted, and then to bind each given element name to subsequent collection values. For example, the following macro call would expand as follows:

```
forEach (Point e1 in c1, Color e2 in c2) f(e1, e2);

==>

Iterator i1 = c1.iterator;
Iterator i2 = c2.iterator;
while (true & i1.hasNext() & i2.hasNext()) {
  e1 = (Point)i1.next();
  e2 = (Point)i2.next();
  f(e1, e2);
}
```

The values of the expander variables are constructed through iterating over each `forEach` clause and when no more clauses remain, returning a code quote including the final expander variable values.

6.2 The Accessible Macro

It is recommended to create a functional interface to fields so as to hide their implementation as fields to allow for implementation changes without affecting client code. The following example shows how to add a functional interface to fields using a syntax expander called `accessible`. The following is an example usage of the `accessible` macro:

```
public class RepeatRule {
  public accessible Date startDate;
  public accessible Date endDate;
  public accessible int  repeatCount = 0;
}
```

and the macro itself is defined as follows:

```
public syntax accessible {
  case #{ ?mods:* accessible ?:type ?:name ?init:*; }: {
    Fragment getterName
      = new IdentifierFragment
          ("get".asCapitalizedString());
    Fragment setterName
      = new IdentifierFragment
          ("set".asCapitalizedString());
    return #{
      private ?type ?name ?init;
      ?mods ?type ?getterName() { return ?name; }
      ?mods ?type ?setterName(?type newValue) {
        ?name = newValue;
      }
    };
  } }
```

Note the creation of getter and setter identifiers using an IdentifierFragment constructor.

6.3 A Macro Defining Macro

We now present an example of a macro which defines other macros. Suppose we find it useful to create aliases for particular methods. Instead of defining a macro for each alias, we could simplify our task by instead defining an alias defining macro which itself defines a macro for each alias.

```
syntax defineAlias {
  case #{ defineAlias ?new:name = ?old:name }:
    return #{
      syntax ?new {
        case #{ ??new ... }:
          return #{ ??old ... };
      } };
}
```

7. TRACING AND DEBUGGING

Without tracing and debugging mechanisms, macros can be difficult to use. We provide a number of facilities which make it much easier to understand macros and when they're not working to figure out why. JSE provides a macro expand facility which permits it to input a string containing a macro call and to output the resulting macro expansion. A macro call can either be fully expanded or expanded one level at a time. This can be used in smart editors to selectively macro expand marked regions of program source.

Even with this selective macro expansion support, it still can often be daunting to understand exactly why macros are failing. Often it is necessary to understand exactly what patterns are matching and how pattern variables are then subsequently bound. JSE provides both a global and per macro tracing flag which controls this output.

Finally, when compiler errors occur, JSE maintains source locations through macro expansion such that the original macro call source can be printed instead of the corresponding macro expansion. This gives programmers compiler feedback based on the code they actually wrote. This can also be used during source level debugging. Unfortunately, because JSE supports procedural macros, it is possible that macros can arbitrarily rewrite original program source and thus in uses of more complicated macros, source level stepping, for example, could be a bit Disorienting. In practice though, we have found that simple source location propagation gives reasonable results.

A number of facilities are made to gracefully handle the case of macro expanders that crash or otherwise fail during expansion. Top level macro expansion exceptions raised during macro expansion are handled and reported through a compiler error message along with appropriate context information. Expanders that fail to terminate will hang the compiler. One possible solution would be to provide a user specified timeout (implemented with a parallel timer thread) which, if reached, would again report non-termination failures through an expansion-time error message.

8. COMPARISONS

8.1 Dylan Macros

Dylan macros were the main inspiration for JSE, but unlike Dylan's rewrite-rule only macro system, JSE exploits Java's compilation model to offer a full procedural macro engine. In other words, syntax expanders may be implemented in, and so use all the facilities of, the full Java language. Because of this, JSE has a simpler pattern matching and rewrite rule engine utilizing standard Java control and iteration constructs for instance. JSE includes a more complicated mechanism for finding the extent of a macro call, because the basic Java syntax does not include an easy mechanism for finding the end bracket.

8.2 Lisp Macros

Lisp's destructuring and quasiquote facilities inspired Dylan's macro system design. Their advent played a big part in popularizing macros and Lisp itself [2]. We maintain that our patterns and templates are as natural to use as Lisp's destructuring and quasiquote. In fact, we feel that our splicing operator (`?`) is an easier to use unification of quasiquote's unquote (`,`) and splicing (`,@`) operators. The reason this is possible in JSE is because pattern variables can be bound to the actual elements of a sequence and not the sequence itself. For example, in Lisp, in order to splice in elements to the end of a parameter list, one uses the splicing operator on a whole sequence as follows:

```
(let ((more '(c d e)))
  `(a b ,@more))
```

while in JSE, one could do the same with the following:

```
Fragment more = #{c, d, e};
return #{a, b, ?more};
```

without the need for a different splicing operator.

In order to better relate JSE's code quotes to Lisp's quasiquote, we present the following table showing a loose correspondence between the two:

Name	**L**isp	**J**SE
quasiquote	`` `() ``	`#{}`
unquote	`,`	`?`
splicing	`,@`	`?`
quote	`'`	N/A
unquote-quote-unquote	`,',`	`??`
unquote-unquote	`,,`	`?(??)`

We feel that for the typical nested code quote situation, JSE's multiple question mark is much more intuitive to use.

Unfortunately, several limitations restrict Lisp macros' ease of use. First, variable capture is a real problem and leads to difficult to debug macros. Second, macro calls are difficult to debug as Lisp macros do not offer a mechanism for correlating between macro expanded code and a user's original source code.

Several other researchers have reported on systems that generalize the quasiquote mechanism to infix syntaxes. Weise and Crew [21] are discussed below. Engler, Hsieh, and Kaashoek [8] employ a version of quasiquote to support a form of partial evaluation in C.

8.3 Scheme Macros

Macro systems for Scheme (e.g., `syntax-rules`) come the closest to offering the power and ease of use of JSE. The major restriction with Scheme macro systems is that they are restricted to languages with sexpr-based syntax. Ignoring this major restriction, we feel that JSE provides a much more cohesive whole that gracefully progresses from a self-contained pattern language to the full power of procedural macros.

In this section we will briefly compare JSE to R5RS's [12] `syntax-rules` rewrite-only and the `syntax-case` [7] procedural macro systems . It is worth mentioning that other

Scheme systems exist (e.g., [5]), but they are beyond the scope of this paper. `syntax-rules` is restricted to a rewrite-only system and thus arbitrary Scheme code can not be utilized during macro expansion. The `syntax-case` extension lifts this restriction. In JSE, the same basic pattern matching language naturally incorporates the full Java language when writing procedural macros.

Even without considering this major limitation, we feel that JSE makes typical macro writing more natural. For example, in `syntax-case`, a programmer is required to introduce local pattern variables using `with-syntax` whereas, in JSE, a programmer merely introduces them with the usual local variable declaration syntax. For example in `syntax-case` one must write the following:

```
(lambda (f)
  (with-syntax ((stuff f))
    (syntax stuff)))
```

whereas in JSE, one could do the equivalent with the following:

```
Fragment doit (Fragment f) { return #{ ?f }; }
```

without having to explicitly introduce `f` as a pattern variable. Furthermore, both `syntax-rules` and `syntax-case` require users to specify reserved intermediate words up front; otherwise names occurring within a pattern or template are interpreted as pattern variables. In JSE, pattern variables have a special notation and thus reserved intermediate words do not need to be declared ahead of time.

`syntax-case` and `syntax-rules` provide nice solutions to hygiene that automatically avoid most variable capture errors. JSE improves on this by providing an intuitive notation (i.e., `?=`) for circumventing hygiene. Consider the `syntax-case` version of the following JSE `nif` macro defined above:

```
(define-syntax nif
  (lambda (x)
    (syntax-case x ()
      ((nif test then else)
       (with-syntax
         ((it (datum->syntax-object (syntax nif) 'it))
          (syntax (let ((it test))
                    (if test then else)))))))))
```

Notice how in `syntax-case`, one must use a long-winded call and a `with-syntax` binding to produce the desired `it` variable. Note that R5RS's `syntax-rules` provides no way to disable hygiene.

One advantage Scheme macros have over JSE is that one can limit a macro's visibility to a lexically local region of code using `let-syntax` and `letrec-syntax`. Although not as precise, JSE does provide class-scoped macros.

8.4 Grammar Extension Macros

Grammar extension macros allow a programmer to make incremental changes to a grammar in order to extend the syntax of the base language. JSE is less ambitious in that it provides a convenient and powerful mechanism for extending the syntax in limited ways. In particular, it provides only a limited number of shapes and requires that macros must always commence with a name. We feel that this tradeoff is justified because it makes the vast majority of macros easier to write.

8.4.1 Programmable Syntax Macros

Weise and Crew [21] describe a macro system for infix syntax languages such as C. Their macro system is programmable in an extended form of C, guarantees syntactic correctness of macro produced code, and provides a template substitution mechanism based on Lisp's quasiquote mechanism. Their system lacks support for hygiene, but instead requires programmer intervention to avoid variable capture errors.

Unfortunately, their system is restrictive. Their macro syntax is constrained to that describable by what is, essentially, a weak regular expression. In contrast, in our system, within the liberal shapes of an SST, just about anything goes, and further, the fragments can be parsed using any appropriate technique. Finally, because templates are eagerly parsed, it's not clear that forward references within expanders (and so mutual recursion, say) works in their system. It could be made to work, however (by forward declaring the input syntax apart from the transformer), but this would be awkward to use.

Their system requires knowledge of formal parsing intricacies. They say:

> The pattern parser used to parse macro invocations requires that detecting the end of a repetition or the presence of an optional element require only one token lookahead. It will report an error in the specification of a pattern if the end of a repetition cannot be uniquely determined by one token lookahead.

We think that it is difficult for programmers to understand and solve static grammar ambiguities like this.

We believe that as far as possible, programmers should only have to know the concrete syntax, not the abstract syntax. Unfortunately, their system requires programmers to use syntax accessors to extract syntactic elements, whereas our system allows access through pattern matching concrete syntax.

Their system requires templates to always be consistent. We feel that it's often useful to be able to generate incomplete templates containing macro parameters for instance, that are spliced together at the end of macro processing to form something recognizable. Weise and Crew's insistence that the result of evaluating a template should always be a recognizable syntactic entity defeats that mechanism. Being able to work easily with intermediate part-expressions that have no corresponding full-AST class (e.g. a disembodied pair of function arguments) is a win for the SST approach.

8.4.2 Metamorphic Syntax Macros

Braband and Schwartzbach [3] introduce a macro system that improves on Weise and Crew's system but is still laden with similar restrictions. In particular, it is tedious to write complicated macros because their system is based solely on rewrite rule extensions to a base grammar. Although, their system can express almost all macros expressible in JSE, having full access to Java control and data structures makes writing demanding macros in JSE much easier. Their advantage is that they can within their system prove at macro definition time that macros will always produce admissible expansions.

8.4.3 Camlp4

Camlp4 [6] is a pre-processor-pretty-printer for Objective Caml. Grammar tools and offers the ability to modify the concrete syntax of the language including new quotations and syntactic extensions. It uses a recursive descent parser and users extend the syntax by adding new rewrite rules. Users can specify levels of precedence so as to help order rules. Rewrite rules are written in terms of pattern matching with replacement patterns and similar restrictions apply. Finally, they offer no guarantees about parseability and/or correctness.

8.5 Java Macro Systems

8.5.1 Jakarta Tool Set

The Jakarta Tool Set [1] (JTS) provides a set of tools for creating domain specific languages. The first of those tools, named Jak, adds support for meta-programming to Java. In particular, Jak supports the definitions of AST constructors and offers hygiene facilities. AST's are created using typed code quotes, and are manipulated using a tree walk. This is in contrast to our procedurally extensible rewrite-rule system which mostly shields the user from the details of the underlying AST.

The second JTS tool, named Bali, is essentially a parser generator, addressing the need for creating syntactic extensions in a more familiar BNF style with regular-expression repetitions. The Bali productions are associated with the class of objects created when a production fires. The result of parsing is an AST which can be further modified through a tree walk. This system is similar to the above mentioned grammar extension macro systems.

8.5.2 EPP

EPP [9] is a Java system for extending a recursive descent parser and lexical analyzer using the composition of mixins. The parser consists of functions which parse each of the non-terminals returning corresponding AST's. Users can define parser mixins in order to extend these functions by defining their own non-terminal parsers which can augment and/or override inherited functions. No guarantees are made about the interactions between these mixins. Similar concerns about ease of use apply to their system. Users are asked to understand the idiosyncrasies of a grammar and the interactions of grammar extension combinations. Although more limited, we consider the manipulation of surface syntax (as in JSE) to be a much less daunting enterprise.

8.5.3 JPP

JPP [18] implements a fixed set of language extensions rather than a means for Java programmers to add their own. It supports `cpp`-style conditional compilation, but not substitution macros: "Macros like the standard C preprocessor has are the cause of many bugs, and because of this I have no plans to support them".

8.5.4 Open Java

OpenJava [20] is a compiler supporting a compile-time meta object protocol (MOP) for Java. A number of protocols for adding or modifying features of the Java language are provided, including a limited syntax extension mechanism. Syntactic extension is limited to only a few certain places in class definitions (e.g., class adjectives) and their uses (e.g., after class names in callers). This allows the system to parse programmer input without potential grammar conflicts. Although an impressive system, its main focus is on semantic rather than syntactic extension. While limited in the amount of potential surface syntax extension, their system allows for more powerful extensions that can depend on logical or contextual information such as types.

A number of other MOP-based systems are worth mentioning. In particular, Chiba introduces OpenC++ [4] which is very similar to OpenJava and shares most of its relavent strengths and weaknesses.

MPC++ [10] is a powerful system which defines a compile-time metalevel architecture including a mechanism for extending limited parts of the C++ grammar. Syntactic extensions are defined by writing what amount to non-terminal mixins similar to EPP. Code quotes are introduced, but generally the user is required to construct code fragments using object constructors. Furthermore, MPC++ does not offer any sort of high level pattern matching facilities nor automatic hygiene support.

9. IMPLEMENTATION STATUS

JSE is currently implemented as a preprocessor taking `.jse` files and producing `.java` files. Hygiene is currently unimplemented, although we have confidence in the the described design as it is based on our Dylan procedural macro system used in Functional Objects' production compiler. A full SST library is implemented and documented. JSE is available as open source software from `www.ai.mit.edu/~jrb/jse`.

10. FUTURE WORK

Many important future directions remain. We would like to see JSE provide more guarantees about macro definitions always producing admissible expansions. We would like to extend JSE to allow for symbol macros and to support generalized variables such as CommonLisp's `setf`. Furthermore, JSE seems like a very nice substrate for staged compilation as exhibited in `'C` (see also [19]). Finally, we think that our approach could be fruitfully applied to other languages such as C and Scheme.

11. CREDITS

A large amount of inspiration for JSE came from the Dylan macro system. Much of the initial design work on Dylan

macros was done at Apple. The syntax of macro definitions, patterns, constraints, and templates used in Dylan's standard macro system was developed by Mike Kahl. Dylan's very first "loose grammar" was due to David Moon, who also designed a pattern matching and constraint parsing model suitable for such a grammar. Earlier, Moon had proposed a model of compile-time evaluation for Dylan in the context of a prefix-syntax macro system from which ours takes some terminology.

Generalizations of Dylan's loose grammar enabling it to describe all Dylan's syntactic forms, along with the first implementation of the macro system, were developed by the authors while at Harlequin Ltd. The procedural macro system was initially designed and implemented as a component of Harlequin's Dylan compiler, now owned by Functional Objects, Inc. Many other "Dylan Partners" made contributions to the design of Dylan macros, particularly the Gwydion team at CMU.

This paper benefitted from helpful discussions with Alan Bawden, Andrew Blumberg, Tony Mann, Scott McKay, Dave Moon, and Greg Sullivan. Finally, we would like to thank the anonymous OOPSLA reviewers for their many helpful suggestions.

12. REFERENCES

[1] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 143–153. IEEE Computer Society Press, 1998.

[2] Alan Bawden. Quasiquotation in Lisp. In *SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12. ACM, January 1999.

[3] Braband and Schwartzbach. Growing languages with metamorphic syntax macros. available on their web site, 2000.

[4] Shigeru Chiba. A metaobject protocol for C++. In *Proceedings of OOPSLA '95*, pages 285–299, October 1995.

[5] William Clinger. Hygienic macros through explicit renaming. *ACM LISP Pointers*, 4(4):25–28, 1991.

[6] Daniel de Rauglaudre. Camlp4. `http://caml.inria.fr/camlp4/`, 2001.

[7] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[8] E.R. Engler, W.C. Hsieh, and M.F. Kaashoek. `‘c`: A language for fast, efficient, high-level dynamic code generation. In *Proceedings of Symposium on Principles of Programming Languages*, January 1996.

[9] Yuuji Ichisugi. Modular and extensible parser implementation. `http://www.etl.go.jp/~epp/edoc/epp-parser.pdf`, 2000.

[10] Yutaka Ishikawa, Atsushi Hori, Mitsuhisa Sato, Motohiko Matsuda, Jorg Nolte, Hiroshi Tezuka, and Hiroki Konaka. Design and implementation of metalevel architecture in C++ – MPC++ approach –. In *Proceedings: Reflection'96*, 1996.

[11] Guy Lewis Steele Jr. Growing a language. *Lisp and Symbolic Computation*, 1998.

[12] R. Kelsey, W. Clinger, and J. Rees. Revised^5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[13] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygenic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161. ACM, ACM, August 1986.

[14] Klil Neori. Self generating java program. available on a web site, 1999.

[15] Peter Norvig. DEFTABLE: A macro for implementing tables. *ACM LISP Pointers*, 5(4):32–38, 1992.

[16] C. Queinnec. *Lisp In Small Pieces*. University Press, Cambridge, 1994.

[17] A. Shalit. *The Dylan Reference Manual*. Addison Wesley, 1996.

[18] Nik Shaylor. Java preprocessor (JPP). `http://www.geocities.com/CapeCanaveral/Hangar/4040/jpp.html`, 1996.

[19] Walid Taha and Tim Sheard. Multi-stage programming with explicit annotations. In *SIGPLAN workshop on partial evaluation and semantics-based program manipulation*, pages 203–217, 1997.

[20] Michiaki Tatsubori. Open Java. `http://www.hlla.is.tsukuba.ac.jp/~mich/openjava/`, 2000.

[21] Daniel Weise and Roger Crew. Programmable syntax macros. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 156–165, June 1993.