# The Java Syntactic Extender

Jonathan Bachrach

MIT AI Lab

Keith Playford

Functional Objects, Inc.

# Macro

- Syntactic extension to core language
- Defines meaning of one construct in terms of other constructs

- Their declarations are called **macro definitions**
- Their uses are called **macro calls**
- **Macro expansion** is the process by which macro calls are recursively replaced by other constructs

# Macro Motivation

- Power of abstraction where functional abstraction won't suffice, affording:
  - Clarity
  - Concision
  - Implementation hiding
- People are lazy
  - do the right thing with less
- "From now on, a main goal in designing a language should be to plan for growth."
  - Guy Steele, "Growing a Language", OOPSLA-98, invited talk

# `forEach` Example

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

=>

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

# Goals and Nongoals

## Goals

- Convenient
- Powerful
- Simple

## Nongoals

- Arbitrary shapes
- Semantics-based

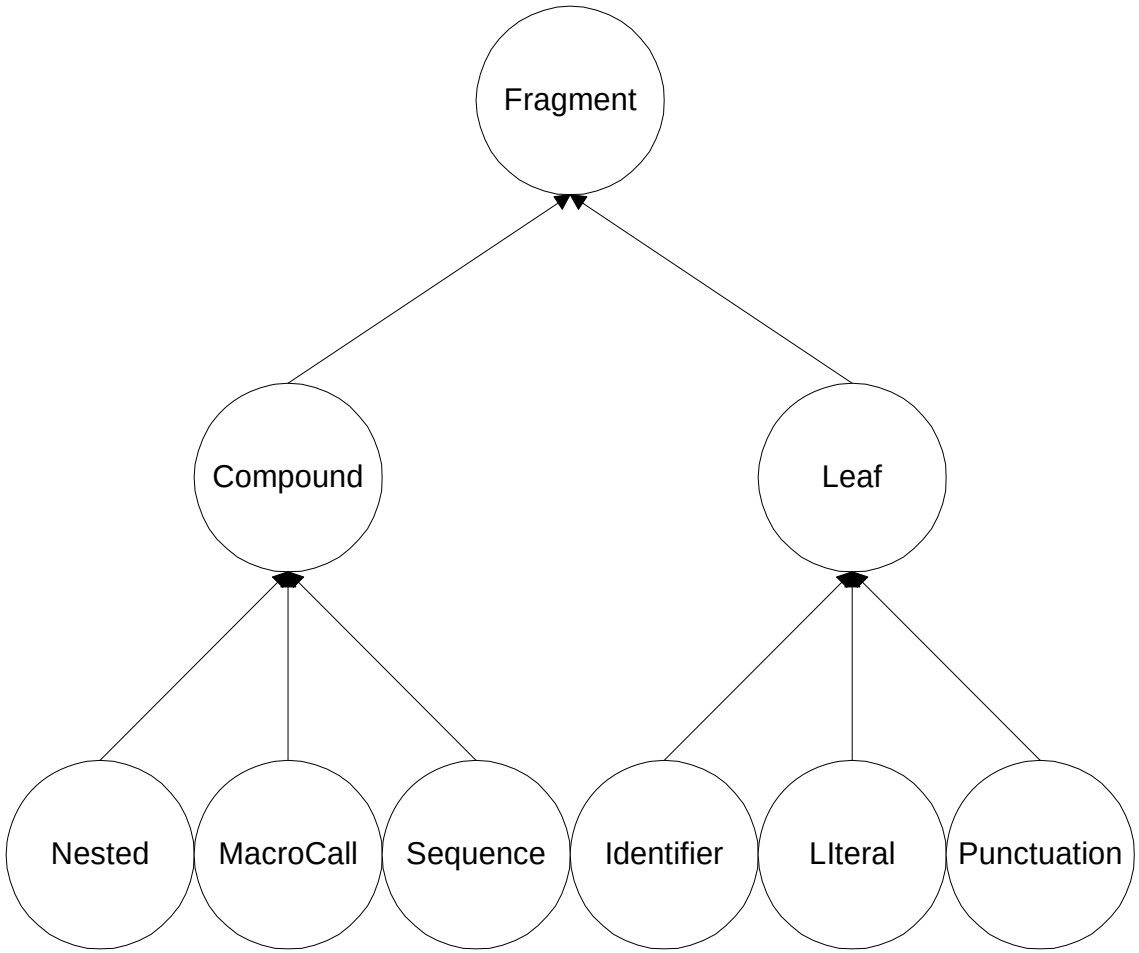
# Key Points

- WYSIWYG (convenient)
  - *Skeleton Syntax Tree* form is OO source representation
  - High-level WYSIWYG code fragment constructors and pattern matching facilities are provided
- Class-based and Procedural (powerful)
  - Macro definitions are represented as classes and
  - Their macro expanders are methods
- Name-based (simple)
  - Macros calls are identified by name and
  - Their macro definitions are found through `CLASSPATH`

# Overview

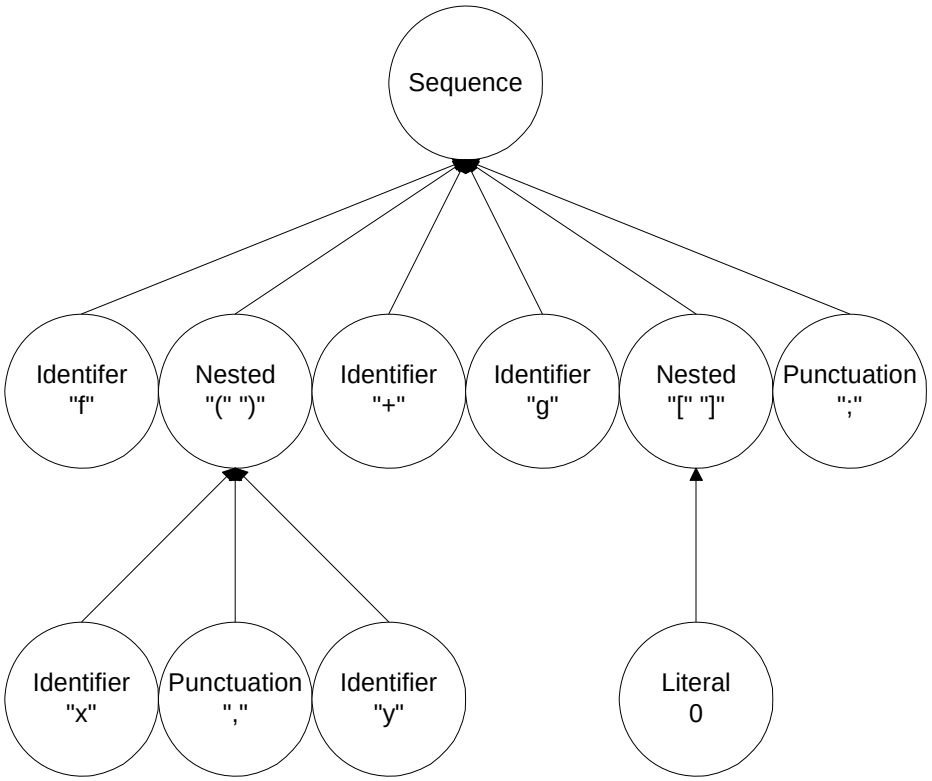
- Fragments
- CodeQuotes
- Pattern Matching
- Parsing / Expansion
- Execution Model
- Examples
- Comparisons
- Extra Paper Topics
- Implementation
- Future Work

# Fragment Classes (for SST)



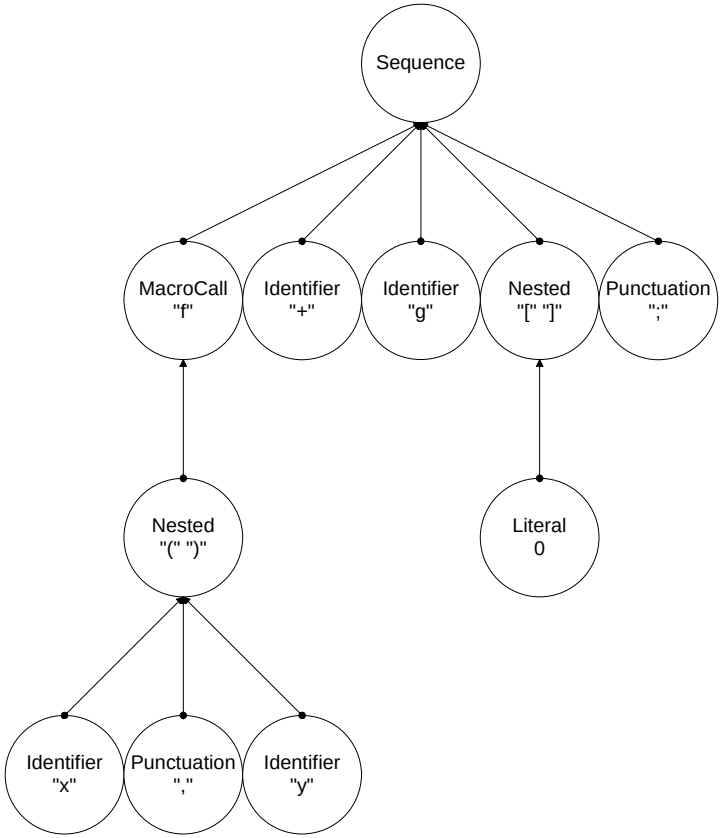
# Skeleton Syntax Tree Example 1

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



# Skeleton Syntax Tree Example 2

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



# CodeQuote

- Like Lisp's quasiquote (QQ)
- WYSIWYG / concrete representation of code within `#{}`'s
  - `#{ if (!isOff()) turnOff(); }`
- Evaluation of codeQuote yields SST form of code
- Quoting is turned off with `?` (like QQ's comma)
  - Variables: `?x`
  - Expressions: `?(f(x))`

# CodeQuote Example One

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

=>

```
#{ if (!isOff()) turnOff(); }
```

# CodeQuote Example Two

```
Fragment getterName (IdentifierFragment id) {
  return new IdentifierFragment
    (“get”.concat(id.asCapitalizedString()));
}
Fragment name = new IdentifierFragment(“width”);
return #{ x.?(getterName(name))() }
```

=>

```
#{ x.getWidth() }
```

# Pattern Matching

- `syntaxSwitch`: like `switch` statement
  - `syntaxSwitch (?:expression) { rules:* }`
- Rule:
  - `case ?pattern:codeQuote : ?:body`
- Pattern:
  - CodeQuote that looks like the construct to be matched
  - Augmented by pattern variables which match and lexically bind to appropriate parts of the construct

# Pattern Variables

- Denoted with `?` prefixing their names
- Have constraints that restrict the syntactic type of fragments that they match
  - Examples: `name`, `expression`, `body`, …
  - Constraint denoted with a colon separated suffix (e.g., `?class:name`)
  - Variable name defaults to constraint name (e.g., `?:type` is the same as `?type:type`)
  - Wildcard constraint (`*`) matches anything
  - Ellipsis (`...`) is an abbreviation for wildcard

# `syntaxSwitch` Example

```
syntaxSwitch (#{ unless (isOff()) turnOff(); }) {
  case #{ unless (?test:expression)
            ?then:statement }:
    return #{ if (! ?test) ?then };
}
```

=>

```
#{ if (!isOff()) turnOff(); }
```

# `syntaxSwitch` Evaluation

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

- 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 body is run in the context of those bindings
- If no patterns are found to match, then an exception is thrown

# Pattern Matching Execution

- Left to right processing
- Shortest first priority for wildcard variables
- Largest first priority for non-wildcard variables
- Patterns match if and only if all of their subpatterns match

# Compilation

- Parse into SST earmarking macro calls
- Recursively expand macro calls top-down
- Create IR from SST
- Optimize IR
- Emit code

# Parsing Macros

- Macros can occur at certain known context
  - Declarations
  - Statements
  - Expressions
- While parsing in macro context lookup macro name
  - Load macro definition class for information
- Find macro call extent
  - Find start and end points
  - Tokens in between serve as macro call arguments

# Macro Shapes

- **Call**:
  - `assert(?:expression, ?message:expression)`
- **Statement**:
  - `try`, `while`, `...`
  - Have optional continuation words (e.g., `catch`)
- Special:
  - Methods, infix operators, fields, etc

# Parsing Function Call Macros

- Start is function name
- End is last matching argument parenthesis
- Example

```
#{ f(assert(x < 0, “bad ” + x), g(y)) }
```

`assert`’s Macro arguments would be:

```
#{ assert(x < 0, “bad ” + x) }
```

# Parsing Statement Macros

- Start is first token past last terminator
- End is first terminator not followed by a continuation word
- Example

```
#{ p(); try { f(); } catch (Exception e) { g(); } n(); }
```

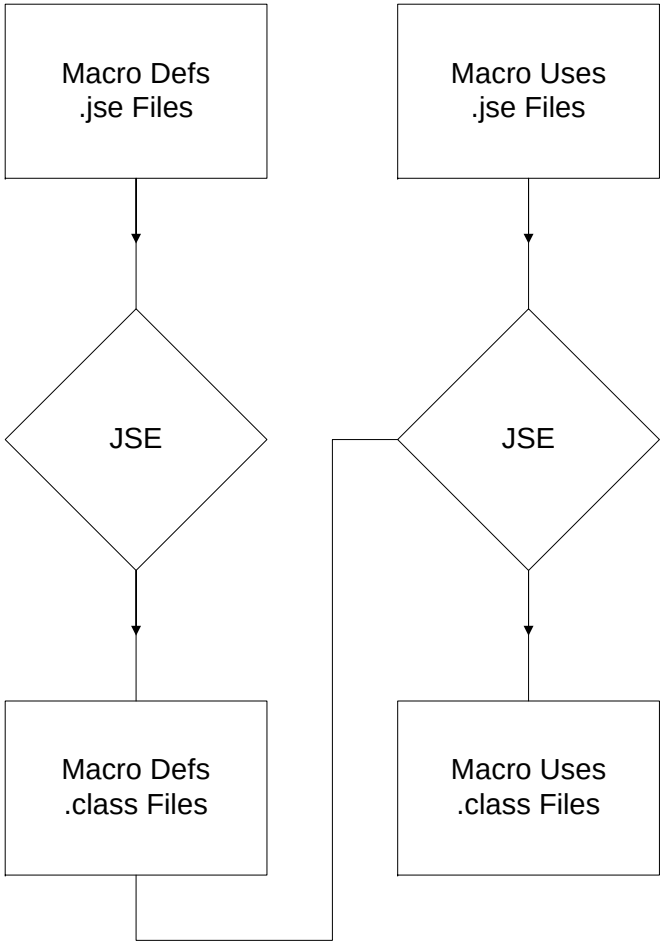
`try`'s macro arguments would be:

```
#{ try { f(); } catch (Exception e) { g(); } }
```

# Execution Model

- Macro Expanders are implemented as classes whose names are the name of the macro with a `"SyntaxExpander"` suffix
  - `forEach`'s macro class would be named `forEachSyntaxExpander`
- Coexist with runtime Java source and class files
- Dynamically loaded into compiler on demand during build
- Macros are looked up using the usual Java class lookup strategy:
  - Package scoped
  - Class scoped

# Execution Model Picture



# Macro Class

- Contains
  - Access specifier
  - Name
  - Shape
  - Continuation words
  - Expand method
  - Extra class declarations

# `forEach` Macro

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

# `syntax` Macro

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

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

# `accessible` Macro

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

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

# Parallel Iteration

```
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, ?rest:* }:
        Fragment newInits = #{ ?inits Iterator i = ?c.iterator(); };
        Fragment newPreds = #{ ?preds & i.hasNext() };
        Fragment newNexts = #{ ?nexts ?name = (?type)i.next(); };
        return loop(rest, statement, newInits, newPreds, newNexts));
    } } }
```

# Comparisons

- Limited to conventionally syntaxed systems

- Dylan
- Grammar Extensions
- MOP-based

# Dylan Macros

- More complicated pattern matching
- More limited shapes
- Not procedural
  - Pattern matching rewrite rule only
  - Authors have proposed a procedural extension

# Grammar Extension Macros

"Programmable Syntax Macros"

by Weise and Crew and

"Growing Languages with Metamorphic Syntax Macros"

by Brabrand and Schwartzbach

- Challenging to understand rules and their interactions
- Tedious to write complicated macros
+ More call shapes possibles
+ Type checkable

# MOP-based Approaches

- EPP and MPC++
  - Users extend recursive descent parser using non-terminal parsing mixins.
  - No guarantees about interactions between mixins
  - EPP has code quotes but not pattern matching
- OpenJava and OpenC++
  - Minimal syntactic extension (e.g., class adjectives)
  - Focuses instead on semantic extension

# Other Features

- Automatic hygiene support
  - Avoids accidental name clashes etc
- Debugging support
  - Tracing and source locations
- Extensible pattern matching
  - User-defined constraints

# Implementation

- Preprocessor
  - Takes `.jse` files
  - Produces `.java` files preserving line numbers
  - Optionally calls Java Compiler
- Uses standard ANTLR lexer and parser
- Tracing, Error Trailing and Hygiene not implemented
- Available by early November, 2001
  - `www.ai.mit.edu/~jrb/jse`
  - `jrb@ai.mit.edu`

# Future Work

- Type checking
- Staged compilation
- More shapes
- Other languages (e.g., Scheme, C, …)

# JSE

- Convenient, powerful, and simple macros for conventionally syntaxed languages
- Open source
  - www.ai.mit.edu/~jrb/jse
- Thanks to:
  - Howie Shrobe for funding
  - Greg Sullivan for support
  - MIT Dynamic Languages Group

---

# Tracing

- Either globally or locally
- Print when and to what pattern variables match
- Print when patterns do or do not match

# Debugging

- Maintain source locations
- If integrated into compiler can also maintain macro expansion context to support error trailing through macro expansion

# Hygiene and Referential Transparency

- Variable references copied from a macro call and definition mean the same thing in an expansion
- Avoids the need for
  - gensym to avoid accidental name clashes and
  - manually exporting names used in macro definitions

# Hygiene Design

- 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
  - Hygiene context is dynamically bound during expansion
  - Hygiene contexts can also be manually established and dynamically bound
- References to global bindings should mean the same thing as they did in macro definition
  - Hard to do in Java without violating security
  - Forces user to manually export macro uses

# Procedural Macro Motivation

- Analysis and rewriting no longer constrained
- Simplified pattern matching and rewrite rule engine
- Can package and re-use syntax expansion utilities
- Pattern matching engine is extensible

# Nested CodeQuotes

- Introduce nested pattern variables and expressions:
  - `??x, ??(f(x)), ???y`
- Evaluate when var/expr's nesting level equals codeQuote nesting level otherwise regenerate:
  - `#{ #{ ?x } }` `=> #{ ?x }`
  - `Fragment x = #{ a }; #{ #{ ??x } } => #{ a }`
- Can keep `?`'s using `!`
  - `Fragment x = #{ y }; Fragment y = #{ a }; #{ #{ ?!?x } }`
  - `=> #{ ?y }`

# Macro Defining Macros

```
syntax defineFieldDefiner {
  case #{ defineFieldDefiner ?type:name ; }:
    Fragment newName
      = new IdentifierFragment
          (type.getName() + “Field");
    return #{
      syntax ?newName {
       case #{ ??newName ?:name = ?:expression ; }:
         return #{ ??type ?name = ?expression ; }
      } };
}
=>
defineFieldDefiner int;
=>
syntax intField {
  case #{ intField ?:name = ?:expression ; }:
    return #{ int ?name = ?expression ; }
}
```

# Self Generating Code Quote

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

Based on Mike McMahon's self generating quasiquote solution published in Alan Bawden's quasiquote paper.

```
(let ((let '`(let ((let ',let)) ,let)))
  `(let ((let ',let)) ,let))
```

# Self Generating Java 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();
  } }
```

## Java Solution by Klil Neori

```
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));}}
```

# Java Syntax

- Java forms fit the pattern

  `… clause clause clause etc`

- where clauses are:

  `thing …;`

  `thing … { }`

- Also expressions

# Credits

- Dave Moon -- Dr. Dylan Macros
- Alan Bawden -- Dr. Quasiquote
- Thomas Mack -- UROP
- Howie & Bob -- Drs. Funding
- Benefitted from discussions with
  - Greg Sullivan
  - Scott McKay
  - Andrew Blumberg

# Declarative Data Examples

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

```
Studies {
  course Math101 {
    title “Mathematics 101”;
    2 points fall term;
  } …
  exclusions {
    Math101 <> MathA;
    Math102 <> MathB;
  }
  prerequisites {
    (Math101, Math102) < (Math201, Math202, Math204);
    (CS101,CS102) < (CS201, CS203);
  }
  …
}
```

# Overview

- Definitions
- Parsing
- Execution Model
- Fragments
- CodeQuotes
- Pattern Matching
- Hygiene
- Nested CodeQuotes
- Debugging
- Comparisons
- Implementation
- Future Work

# Say What?

- Lisp-style macro power and simplicity for Java
- Debt to Dylan and Lisp is great
- Seamlessly procedural
- WYSIWYG
- Mostly hygienic
- Source level debuggable

# User Defined Constraints

- Based on class
  - Whose name is constraintName + “SyntaxConstraint”
  - Loaded on demand using standard Java class loading mechanism
  - That implements the SyntaxConstraint protocol
    - `String getName()`
    - `boolean isAdmissable(SequenceFragment frags)`

# TestSuite Macros

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

=>

```
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);
};
```

# Lisp Macros

- Quasiquote is a bit more complicated (but potentially more powerful):
  - No quote
  - No unquote-splicing
  - `,',x` versus `??x` and `,,x` versus `?!?x`
- Variable capture is a problem
- Macro calls are difficult to debug

# R5RS Macros

- `syntax-rules` is not procedural
- Two environments
- … is cute but brittle
- Pattern variables are defaults
- No hygiene escapes
- Local syntax
- Other Scheme macro systems exist

# Fragments

- Fragments library provides a collection of classes suitable for representing fragments of source code in skeleton syntax tree 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

# Macro Expansion

- Replaces macro call with another construct, which itself can contain macro calls.
- This process repeats until there are no macro calls remaining in the program