How to Create a Java Metaobject Class in Eclipse

José de Oliveira Guimarães
`josedeoliveiraguimaraes@gmail.com, jose@ufscar.br`

Federal University of São Carlos — November 4, 2019

This text explains how to create a metaobject class in the Eclipse IDE and how to use it in a Cyan program. All the code used is in www.cyan-lang.org.

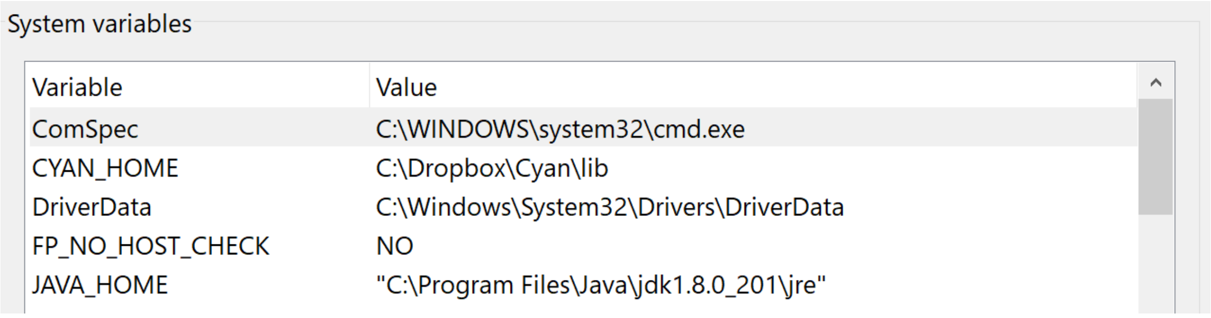
1 Setting up Windows

First of all, download lib.zip from www.cyan-lang.org. This file contains all that is necessary in order to create a metaobject class. We will suppose this file is uncompressed in a directory “`C:\Dropbox\Cyan\lib`”. Remember that, currently, the Cyan compiler only works in Windows.

Follow carefully the following steps in exactly the given order. Any misstep will cause difficult-to-discover errors.

Set the System environment variable `CYAN_HOME` to `"C:\Dropbox\Cyan\lib"`. Prudent readers should create a User variable `CYAN_HOME` too. Install both Java 8 (JDK 1.8) and Java 11 (JDK 11). Probably any version higher than 11 will work.

Set the System environment variable `JAVA_HOME` to the `jr` directory of Java 8 (JDK 1.8). See the image:



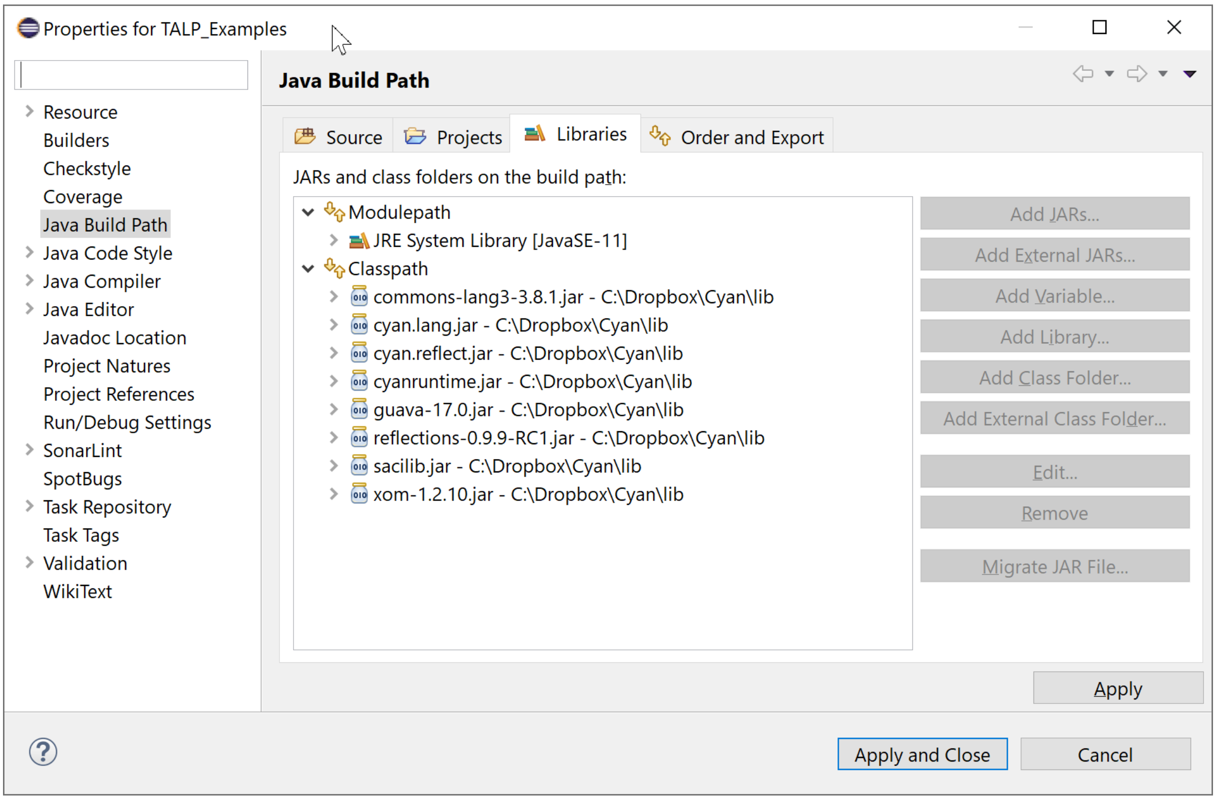
Variable	Value
ComSpec	C:\WINDOWS\system32\cmd.exe
CYAN_HOME	C:\Dropbox\Cyan\lib
DriverData	C:\Windows\System32\Drivers\DriverData
FP_NO_HOST_CHECK	NO
JAVA_HOME	"C:\Program Files\Java\jdk1.8.0_201\jre"

2 Setting up the Eclipse IDE

In Eclipse, create a new Java project called “MetaExamples” in directory
`"C:\Dropbox\Cyan\metaobjectClasses\metaExamples"`

If Eclipse creates a file “module-info.java”, delete it (this is very important).

Create a new package called “`metaobj`” and, inside it, a class called “`CyanMetaobjectOne`”. Click with the right mouse button on the project and choose “Build path/Configure build path”. Something like the image below will appear.

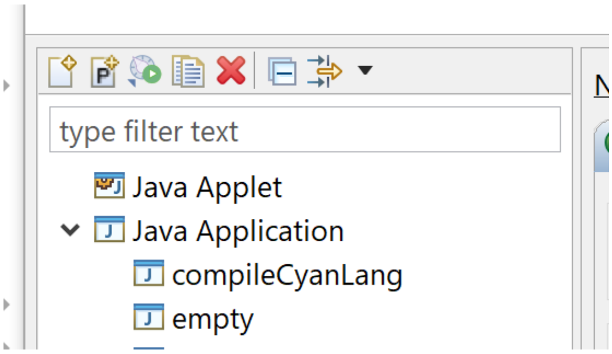


JARs and class folders on the build path:

- Modulepath
  - JRE System Library [JavaSE-11]
- Classpath
  - commons-lang3-3.8.1.jar - C:\Dropbox\Cyan\lib
  - cyan.lang.jar - C:\Dropbox\Cyan\lib
  - cyan.reflect.jar - C:\Dropbox\Cyan\lib
  - cyanruntime.jar - C:\Dropbox\Cyan\lib
  - guava-17.0.jar - C:\Dropbox\Cyan\lib
  - reflections-0.9.9-RC1.jar - C:\Dropbox\Cyan\lib
  - sacilib.jar - C:\Dropbox\Cyan\lib
  - xom-1.2.10.jar - C:\Dropbox\Cyan\lib

Click in “Classpath” and choose “Add External JARs”. Add all the listed jar files of directory `"C:\Dropbox\Cyan\lib"`. In “Modulepath”, make sure that the JRE version is at least 11. To change that, in the image above we would click in “JRE System Library [JavaSE-11]”, click in “Edit” and change the JRE.

Return to the main Eclipse workspace and click in tab “Run/Debug configurations”. In the left, choose “Java application” and then click in the leftmost icon in the upper part, that with a small +.



In the left, give the name “Mo examples” to the configuration. In tab “Main”, choose project “MetaExamples”. As the “Main class”, choose “saci.Saci”.

Click in tab “(x)= Arguments” and put `"C:\Dropbox\Cyan\examples\first"`. Make sure that, in tab “JRE”, the “Runtime JRE” is “JavaSE-11”. Otherwise, change “Execution environment” to Java 11.

click in “Apply” in the lower-left side of the window “Debug configurations”.

3 Creating the First Metaobject Class

Create directory `"C:\Dropbox\Cyan\examples\first"`. Inside it, create directory “`main`” with a file “`Program.cyan`” with the following contents:

```
package main

@one(0, "abc", def, [ 0, 1, 2 ], [. "name", 33 .] )
object Program
    func run {
        self.count = 5;
        count println;
    }
end
```

Return to the project. Make class `CyanMetaobjectOne` inherits from `CyanMetaobjectWithAt`. The constructor should be this:

```
        public CyanMetaobjectOne() {
            super("one", AnnotationArgumentsKind.ZeroOrMoreParameters,
                    new AttachedDeclarationKind[] {
                            AttachedDeclarationKind.PROTOTYPE_DEC
                    });
        }
```

It is expected that the metaobject class name is “CyanMetaobject” followed by the annotation name with the first letter in uppercase (in this case, “one”). The annotation is used in the Cyan prototype `Program`.

`AnnotationArgumentsKind` gives the allowed number of parameters of the annotation. In this case, zero or more. The third parameter to the super constructor is an array of `AttachedDeclarationKind`, the kind of declarations that the annotation can be attached to.

The complete code of class `CyanMetaobjectOne` is in Section A and in the site www.cyan-lang.org.

Without implementing any interface, a metaobject class does nothing. Then let us make `CyanMetaobjectOne` implement two interfaces:

```
     public class CyanMetaobjectDois extends CyanMetaobjectWithAt
          implements IAction_afti, ICheckDeclaration_afsa {
          ...
     }
```

Both interfaces and the superclass are in package `meta` of the file “`sacilib.jar`”.

Copy the source code attached to this text in www.cyan-lang.org. It is the code given in Section A.

We want to add a field `count` to the prototype attached to the annotation `one`. This is made by method `afti_codeToAdd`. See the description of this method in the text “The Cyan Metaobject Protocol” available in www.cyan-lang.org. The tuple returned consists of the full source code to be added and the “interfaces” of the fields and methods. What is an “interface” in this case? Consult the cited text.

Method `afsa_checkDeclaration` is called in phase afsa. In this phase, metaobjects cannot change the code anymore, which is ideal for doing checks. In order to implement any metaobject method, it is necessary information on the annotation parameters (if any) and on the source code in which the annotation is. There are many ways to get the necessary information. The object of the Abstract Syntax Tree that describes the annotation is retrieved using method `getMetaobjectAnnotation()` of the metaobject (which is who receive the message. For short, it is `this`). To retrieve the annotation parameters, use `getJavaParameterList()`.

```
    WrCyanMetaobjectWithAtAnnotation annot =
            this.getMetaobjectAnnotation();
    List<Object> paramList = annot.getJavaParameterList();
```

If annotation `one` is used as in the example,

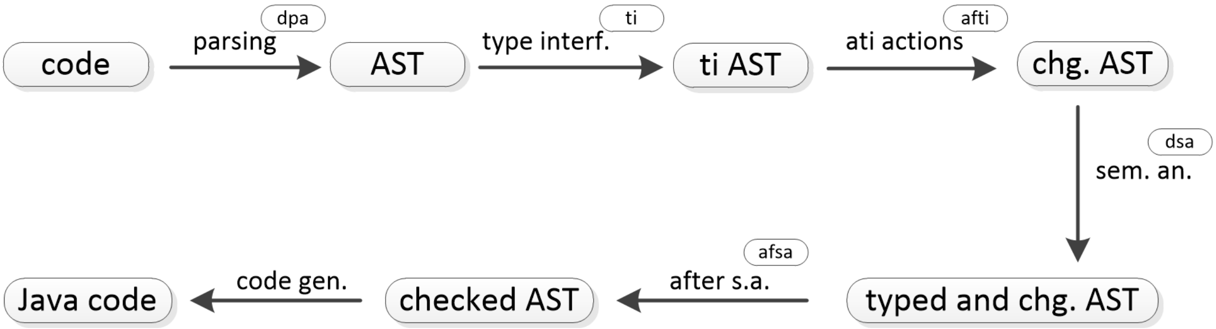


Figure 1: The compiler phases

```
@one(0, "abc", def, [ 0, 1, 2 ], [. "name", 33 .] )
```

Then variable `paramList` (see above) will contain Java objects of the following types:

```
Integer
String
String
Object []
Object []
```

Both Cyan literal arrays and Cyan tuples will be converted into Java arrays. Note that named tuples (like `[. name = "name", age = 33 .]`) are not allowed.

The second parameter will contains quotes and the third will not. As strings, their values are

```
"\"abc\""
"def"
```

To remove the quotes of the second parameter, do this:

```
String p2 = (String ) paramList.get(1);
String p2_wo_Quotes = CyanMetaobject.removeQuotes(p2);
```

4 The Compilaton Phases

The compilation phases are shown in Figure 1. The parsing of the source code is made in phase dpa. Everything outside method bodies is typed in phase ti. Phase afti allows one to add fields and methods to prototypes and rename methods. Phase dsa is the semantic analysis of statements and expressions *inside* methods. Metaobjects can produce code after their corresponding annotations in this phase. Checks should be made in phase afsa. In it, the code cannot be changed anymore. There are metaobject interfaces for phases dpa, afti, dsa, and afsa. Choose the appropriate one for the goals of the metaobject. If it is necessary to add fields and methods, implement interface `IAction_afti`. It the metaobject should just do checks, implement some interface of phase afti. If the annotation should generate and expression inside a method, implement interface `IAction_dsa`. If the annotation takes an attached DSL code, implement interfaces

```
IParseWithoutCyanCompiler
IParseWithCyanCompiler
```

5 Where to Get Information

Other information can be got from the method parameters. In this case, `afsa_checkDeclaration` has only one parameter, `compiler`. From it, one can get the *environment* of the compilation, which is an object with data on the current prototype, current method, compilation phase, and so on.

```
WrEnv env = compiler.getEnv();
```

Variable `env` keeps data on the current prototype. It is passed as parameter to some methods that request AST objects. The compiler will not allow one prototype to retrieve private data of another prototype. It checks that using the environment of the requesting prototype which is passed as parameter to the second one.

Annotations are usually attached to declarations. In this case, `one` can be attached to prototypes only. The declaration to which the annotation is attached to is retrieved with "`this.getAttachedDeclaration()`". The return has type `IDeclaration`.

```
IDeclaration dec = this.getAttachedDeclaration();
WrProgramUnit pu = (WrProgramUnit ) dec;
```

In the example, `one` can only be attached to prototypes, so it is safe to cast `dec` to a `WrProgramUnit`, which is the AST class used to represent prototypes. And if we want to allow `one` to be attached to methods too? We need to change the metaobject class constructor:

```
public CyanMetaobjectOne() {
    super("one", AnnotationArgumentsKind.ZeroOrMoreParameters,
            new AttachedDeclarationKind[] {
            AttachedDeclarationKind.PROTOTYPE_DEC,
            AttachedDeclarationKind.METHOD_DEC
    });
}
```

And now we need to cast `dec` to the appropriate type:

```
IDeclaration dec = this.getAttachedDeclaration();
if ( dec instanceof WrProgramUnit ) {
    WrProgramUnit proto = (WrProgramUnit ) dec;
    // do something
}
else if ( dec instanceof WrMethodDec ) {
    WrMethodDec method = (WrMethodDec ) dec;
    // do something
}
```

Objects of the AST that represent the program can be got using `env`, `compiler`, or other AST objects. For example, one can get the current compilation unit through `env`:

WrCompilationUnit compUnit = env.getCurrentCompilationUnit();

The compilation unit is the full file in which the annotation is. It contains the package declaration, the import declarations, and the sole prototype.

The full text of the compilation unit is available:

```
char []text = compUnit.getOriginalText();
```

The complete list of symbols of the compilation unit, including the comments, is also at one's disposal:

```
WrSymbol []symbolList = compUnit.getSymbolList();
int sizeSymbolList = compUnit.getSizeSymbolList();
for (int i = 0; i < sizeSymbolList; ++i) {
    System.out.print(symbolList[i] + " ");
}
```

Use `getSizeSymbolList()` to discover the size of the array `symbolList` instead of `symbolList.length`. Using this array, a metaobject can, for example, check the style of the source code that depends on the indentation: there is a white space before and after "+" and "*"? Comments are aligned with the declaration they comment?

The current method, if there is one, and the current prototype, if there is one, can be got too:

```
WrMethodDec currentMethod = env.getCurrentMethod();
WrProgramUnit currentPrototype = env.getCurrentProgramUnit();
```

6 Visiting the Abstract Syntax Tree

All AST objects support method `accept` that implements the Design Pattern *Visitor*. To discover what is visited, see class `WrASTVisitor`. As an example of the use of the pattern, we can print, at compile-time, the name of all methods using `accept`:

```
IDeclaration dec = this.getAttachedDeclaration();
WrProgramUnit pu = (WrProgramUnit ) dec;
pu.accept( new WrASTVisitor() {
    @Override
    public void visit(WrMethodDec node, WrEnv env2) {
        System.out.println(node.getName());
    }

}, env);
```

We assume this code was added to method `afsa_checkDeclaration`. It would work in method `afti_codeToAdd` too. This method is declared in interface `IAction_afti` which is implemented by `CyanMetaobjectOne`.

7 How to Issue Compilation Errors?

Compiler errors are issued with methods `addError` of the metaobject (`this`):

```
WrCyanMetaobjectWithAtAnnotation annot =
        this.getMetaobjectAnnotation();

this.addError("This is the error message");
this.addError(annot.getFirstSymbol(), "This is the error message");
this.addError(pu.getFirstSymbol(env), "This is the error message");
this.addError(pu.getMethodDecList(env).get(0).getFirstSymbol(env),
        "This is the error message");
```

In the first call to `addError`, just one parameter, the compiler will point the error in the annotation `one`. In the other lines, the symbol used for pointing the error is given as the first parameter. The second call to `addError` is identical to the first one. In the third call, the error is pointed in the symbol “`object`” of the prototype to which the annotation is attached. In the last call, the error is pointed in the first prototype method (prototype to which the annotation is attached to).

Suppose we want to sign an error if the prototype attached to annotation `one` uses statement “`break`”. The code for that is:

```
CyanMetaobjectWithAt metaobject = this;
pu.accept( new WrASTVisitor() {
    @Override
    public void visit(WrStatementBreak node, WrEnv env2) {
        metaobject.addError(node.getFirstSymbol(),
                "’break’ are not allowed in prototype "
                + pu.getFullName());
    }

}, env);
```

Note that we need to assign `this` to a variable `metaobject` since `this` inside the annonymous object has another meaning. Note that the first `addError` parameter is `node.getFirstSymbol()`. Without this parameter, the error would be pointed in keyword “`object`” of the prototype.

This code works if it is put in method `afsa_checkDeclaration` or in any method of interfaces associated to the compiler phases afsa or dsa. It does not work in phase afti. Then this code does not work if it is in method `afti_codeToAdd` (see the full metaobject class in Section A). Why?

In phase afti the compiler did not analyzed method statements. It has typed fields and method signatures but not method statements. Therefore, the compiler will not call `visit` methods whose parameters, like `WrStatementBreak`, represent statements or anything that can be inside a method.

8 Getting Data from Visitor Classes

Only final or effectively final local variables can be used inside an anonymous class. Then the code below results in a compilation error:

```
int breakCount = 0;
pu.accept( new WrASTVisitor() {
    @Override
    public void visit(WrStatementBreak node, WrEnv env2) {
        ++breakCount; // compilation error
    }
}, env);
```

This problem can be solved in two ways. One is to use a wrapper for the integer. Package `cyanruntime` of `sacilib` already provides one:

```
cyanruntime.Ref<Integer> breakCount = new cyanruntime.Ref<>();
breakCount.elem = 0;
pu.accept( new WrASTVisitor() {
    @Override
    public void visit(WrStatementBreak node, WrEnv env2) {
        ++breakCount.elem;
    }
}, env);
if ( breakCount.elem != 0 )
    addError("’break’ statement is not allowed");
```

The other option is to use an external class. Since its use is local, it can be declared in the same source file it is used.

```
class BreakCountASTVisitor extends WrASTVisitor {
    public int breakCount = 0;

    @Override
    public void visit(WrStatementBreak node, WrEnv env) {
        ++breakCount;
    }
}
```

This class would be used as shown:

```
BreakCountASTVisitor bcount = new BreakCountASTVisitor();
pu.accept(bcount, env);
if ( bcount.breakCount != 0 )
    addError("’break’ statement is not allowed");
```

This code is put in class `CyanMetaobjectTwo` (Section B) that implements `IAction_afti` and `ICheckDeclaration_afsa`

9 Using Annotation Parameters

Parameters to an annotation may direct the workings of a metaobject. To exemplify that, metaobject `two`, defined in class `CyanMetaobjectTwo`, will add a field defined by its two first parameters and, if there is a third parameter and it is `break`, no breaks will be allowed in the prototype. `one` can only be attached to prototypes. The first two parameters should be strings and they should be the values returned, as a tuple, by method `afti_codeToAdd`. Then `two` could be used as

```
@two( "var Int count = 0;",
      "var Int count",
      break )
```

```
object Two
    func run {
        self.count = 5;
        count println;
    }
end
```

The code of `afti_codeToAdd` is trivial. It just checks if there are at least two parameters and if they are strings — see the source code. Method `afsa_checkDeclaration` checks if there is a third parameter, if it is a string, and if the string is `"break"`. Note that there are two ways of given the third parameter:

```
@two( "...", "...", break )
@two( "...", "...", "break" )
```

In the last one, the parameter will have quotes that need to be removed. Method `CyanMetaobject.removeQuotes` remove quotes if they exist in the parameter:

```
boolean checkBreak = false;
if ( paramList.size() == 3 ) {
    if ( !(paramList.get(2) instanceof String) ) {
        this.addError("The third parameter to this annotation "
                + "should be a string");
    }
    String strBreak = (String ) paramList.get(2);
    strBreak = CyanMetaobject.removeQuotes(strBreak);
    if ( strBreak.equalsIgnoreCase("break") ) {
        checkBreak = true;
    }
}
if ( checkBreak ) {
    // use the AST to verify if there are any 'break's
}
```

One is tempted to put `checkBreak` as a field of `CyanMetaobjectTwo` and initialize it in phase afti, in method `afti_codeToAdd`. Then field `checkBreak` would be used in method `afsa_checkDeclaration` of phase afsa. That does not work currently. Fields of metaobject classes keep their values between methods of interfaces

```
IParseWithoutCyanCompiler
IParseWithCyanCompiler
```

to methods of interfaces of afti, dsa, and afsa. Then, a field initialized in a method of phase afti is not seen in phase dsa, for example. What happens is that, after phase afti, the compiler parses the source file again and a new metaobject is created. The value put in the metaobject field in phase afti is lost because the metaobject is discarded. This will change in a not-near future.

Note that there is a value put in a metaobject field in phase `IAction_dpa`, method `dpa_codeToAdd`, is not seen in phases dsa or afsa.

10 Annotations with Attached DSL Code

An annotation can have an attached DSL as `doc`:

```
@doc{*
    This prototype is used for testing the code
*}
object Test ... end
```

We will create a metaobject `three` whose annotations must have a code that follows the syntax of this grammar:

```
    Start :: = Stat { "," Stat }
    Stat  :: = "if:" N | "while:" N | "repeat:" N
               | no break
```

Anything between { and } can be repeated zero or more times.

Annotation `three` controls the number of `if`, `while`, and `break` statements in overridden subprototype methods. The annotation should be attached to a method. An example is:

```
package main

open
object Three
    func run {
    }

    @three{*
        if: 1,
        while: 1
    *}
    func unaryMethod {
        // ok, the checks are in the overridden method
        if 0 < 0 { }
        if 1 < 1 { }
        if false < true { }
    }

    @three{*
        if: 1,
        while: 0,
        no break
    *}
    func at: Int n with: String s {
    }
end
```

A subprototype of `Three` that overrides `unaryMethod` should use at most one `if` and one `while` statement.

To implement metaobject `three`, we need to parse the DSL code. You can implement interface

`IParseWithoutCyanCompiler_dpa`

or just parse the DSL code inside any other metaobject method. In this case, you will need the text of the attached DSL:

```
    WrCyanMetaobjectWithAtAnnotation annot = this.getMetaobjectAnnotation();
    char []text = annot.getTextAttachedDSL();
```

With `text`, do the parsing as usual.

However, if the DSL is similar to Cyan itself, the easiest way of doing the parsing is implementing interface `IParseWithCyanCompiler_dpa` and defining its method `dpa_parse`.

```
    @Override
    public void dpa_parse(ICompiler_dpa compiler_dpa) {
        compiler_dpa.next();
        while ( compiler_dpa.getSymbol().token != Token.EOLO ) {
            ...
        }
    }
```

The first statement should be

`compiler_dpa.next();`

because it eats the `{*` (in the above example) that follows the annotation name. The last `*}` composite symbol is translated to token `EOLO`. The current symbol is got by

```
compiler_dpa.getSymbol()
```

To compare it with a Cyan symbol, use field `token` and class `Token`:

```
if ( compiler_dpa.getSymbol().token != Token.COMMA ) {
    break;
}
```

The token for an identifier is `Token.IDENT` and for identifier followed by a "`:`" is `Token.IDENTCOLON`. The string for any symbol is got by calling method `getSymbolString()`:

```
String id = compiler_dpa.getSymbol().getSymbolString();
```

Errors can be signaled both using `this.addError` or method `error` of parameter `compiler_dpa`. After the parsing, method `dpa_parse` should always check that the end of the code was reached:

```
if ( compiler_dpa.getSymbol().token != Token.EOLO ) {
    compiler_dpa.error(compiler_dpa.getSymbol(), "End of file expected");
}
```

See the source code of `CyanMetaobjectThree` in Section C.

The number of `if`, `while`, and `break` statements are checked in method `afsa_checkOverride` of interface `ICheckOverride_afsa`. This method is called whenever the method attached to the annotation `three` is overridden in a subprototype. In this example, `afsa_checkOverride` uses method `accept` of the subprototype method in order to count the number of `if`, `while`, and `break` statements. It then compares each number with the allowed number of each of them (collected from the DSL code of the annotation). Note that this is possible because fields of `CyanMetaobjectThree` initialized in the method of interface

```
IParseWithoutCyanCompiler
```

can be used in phase `afti`.

There is one missing point: the metaobject class of an annotation that accepts an attached DSL code must define a `shouldTakeText` method that returns `true`:

```
@Override
public boolean shouldTakeText() { return true; }
```

11 Annotations that are Expressions

Annotations can be expressions in Cyan. Metaobject `lineNumber` of package `cyan.lang` returns the current line number and `eval` evaluates a Cyan code at compile-time:

```
var Int lineNumber = @lineNumber;
assert @lineNumber == lineNumber + 1;
var Int sumFrom_1_to_10 = @eval("cyan.lang", "Int"){*
    var Int count = 0;
    for n in 1..10 {
        count = count + n
    }
    return count;
*};
```

A metaobject class whose annotation returns an expression should implement interface `IAction_dsa`[1] and define at least the following three methods.

```
@Override
public String getPackageOfType() {
    return "cyan.lang";
}
@Override
public String getPrototypeOfType() {
    return "Int";
```

---

[1] The class of `lineNumber` implements interface `IAction_dpa`. This interface will be removed in a near future.

```
    }

    @Override
    public boolean isExpression() {
        return true;
    }
```

The expression should be returned by method `dsa_codeToAdd` of interface `IAction_dsa`. The type of the expression should be a prototype or Java class whose package and name are returned by `getPackageOfType()` and `getPrototypeOfType()`. The strings returned by these two methods need not to be constants as the example. They may be calculated before phase dsa. Method `isExpression()` should return `true`.

Annotation `four` generates a value based on its sole parameter.

```
    package main

    object Four
        func run {
            run: 0, "0";
        }
        func run: Int n, String p {
            var Int methodNumber = @four("method");
            printexpr methodNumber;
            assert @four("method") == 4;
            assert @four(field) == 1;
            @four(statement) println;
            assert @four(parameter) == 2;
            assert @four(interface) == 0;
        }

        var String four = "quatro";
    end
```

If the parameter is `method` or `"method"`, the annotation generates the number of methods in the current prototype. In the example, the number is 4 because the compiler adds some methods to the prototype. The other parameters produce similar values.

The class of metaobject `four` is in Section D. Its constructors define that the kind of attached declaration is “none”. That is, the annotations should not be attached to anything.

12 Literal Strings and Numbers as Annotations

User-defined literal strings and numbers can be annotations:

```
package main

object Five
    func run {
           // ’a’ followed by any number of character
           // and ending with a ’b’
        var regexpr = r"a.*b";
        assert "a#xcyb" ~= regexpr;
        assert 10bin == 2;
    }
end
```

Here, `r"a.*b"` is a literal string and `10bin` is a number, both are metaobjects. The metaobjects are defined in package `cyan.lang`.

A metaobject class of a user-defined literal string should inherit from class

```
    CyanMetaobjectLiteralString
```

and implement interface
`IParseWithoutCyanCompiler_dpa`
The class that implementes literal string `r` is
`CyanMetaobjectLiteralStringRegExpr`
Its constructor should call `super` passing the identifiers that can start the literal string:

```
public CyanMetaobjectLiteralStringRegExpr() {
    super(new String[] { "r", "R" });
}
```

More then one option is valid, each one can have several letters and numbers (starting with a letter). Then we can also use `R"a.*b"`.

Method `dpa_parse` of the metaobject class of literal string `r` (the regular expression of the example) is

```
@Override
public void dpa_parse(ICompilerAction_dpa compilerAction, String code) {

    //String t = code.substring(2, code.length() - 1);
    try {
        java.util.regex.Pattern.compile(code);
    }
    catch (java.util.regex.PatternSyntaxException e) {
        addError("Pattern is not well defined");
    }
    codeToGenerate = new StringBuffer( "RegExpr(\"" + code + "\")");
}
```

Parameter `code` is the literal string, `"a.*b"` in the above example. Java libraries are used to validate the string. The generated code, in method `dsa_codetoAdd` of `IAction_dsa`, just output field `codeToGenerate`:

```
@Override
public StringBuffer dsa_codeToAdd(ICompiler_dsa compiler_dsa) {
    return this.codeToGenerate;
}
```

Since the literal string is an expression, methods `getPackageOfType()` and `getPrototypeOfType()` should be defined:

```
@Override
public String getPackageOfType() {
    return "cyan.lang";
}
@Override
public String getPrototypeOfType() {
    return "RegExpr";
}
```

Class `CyanMetaobjectLiteralString` already implements interface `IAction_dsa` so it does not to be implemented by
`CyanMetaobjectLiteralStringRegExpr`

User-defined literal numbers, like `10bin`, are also metaobjects whose classes inherit from
`CyanMetaobjectNumber`
Interface `IParseWithoutCyanCompiler_dpa` is implemented by this class.

A CyanMetaobjectOne

```
package metaobj;

import java.util.List;
import meta.AnnotationArgumentsKind;
import meta.AttachedDeclarationKind;
import meta.CyanMetaobject;
import meta.CyanMetaobjectWithAt;
import meta.IAction_afti;
import meta.ICheckDeclaration_afsa;
import meta.ICompiler_afti;
import meta.ICompiler_dsa;
import meta.IDeclaration;
import meta.ISlotInterface;
import meta.Tuple2;
import meta.WrASTVisitor;
import meta.WrCyanMetaobjectAnnotation;
import meta.WrCyanMetaobjectWithAtAnnotation;
import meta.WrEnv;
import meta.WrMethodDec;
import meta.WrProgramUnit;
import meta.WrStatementBreak;
import meta.WrStatementRepeat;
import meta.WrStatementWhile;

public class CyanMetaobjectOne extends CyanMetaobjectWithAt
implements IAction_afti, ICheckDeclaration_afsa {

  public CyanMetaobjectOne() {
    super("one", AnnotationArgumentsKind.ZeroOrMoreParameters,
        new AttachedDeclarationKind[] {
        AttachedDeclarationKind.PROTOTYPE_DEC
    });
  }

  @Override
  public Tuple2<StringBuffer, String> afti_codeToAdd(
      ICompiler_afti compiler,
      List<Tuple2<WrCyanMetaobjectAnnotation,
      List<ISlotInterface>>> infoList) {


    IDeclaration dec = this.getAttachedDeclaration();
    WrProgramUnit pu = (WrProgramUnit ) dec;
    WrEnv environment = compiler.getEnv();
    WrCyanMetaobjectWithAtAnnotation annot =
        this.getMetaobjectAnnotation();
    for ( Object obj : annot.getJavaParameterList() ) {
      if ( obj instanceof Integer ) {

      }
    }

    return new Tuple2<StringBuffer, String>(
        new StringBuffer("var␣Int␣count␣=␣0;"),
        "var␣Int␣count");
  }
  @Override
  public void afsa_checkDeclaration(ICompiler_dsa compiler) {

    IDeclaration dec = this.getAttachedDeclaration();
    WrProgramUnit pu = (WrProgramUnit ) dec;
    WrEnv env = compiler.getEnv();
    WrCyanMetaobjectWithAtAnnotation annot =
        this.getMetaobjectAnnotation();
    List<Object> paramList = annot.getJavaParameterList();
    //char []textDSL = annot.getTextAttachedDSL();
//    String p2 = (String ) paramList.get(1);
//    String p2_wo_Quotes = CyanMetaobject.removeQuotes(p2);
//    WrCompilationUnit compUnit = env.getCurrentCompilationUnit();
//    char []text = compUnit.getOriginalText();
//    WrSymbol []symbolList = compUnit.getSymbolList();
//    int sizeSymbolList = compUnit.getSizeSymbolList();
//
//    for (int i = 0; i < sizeSymbolList; ++i) {
//      System.out.print(symbolList[i] + " ");
//    }
//
//
//    env.getCurrentMethod();
//    env.getCurrentProgramUnit();

    for ( Object obj : paramList ) {
      if ( obj instanceof Integer ) {
        System.out.print("parameter␣is␣an␣Int:␣" + obj);
      }
      else if ( obj instanceof String ) {
        String s = (String ) obj;
        String sWithoutQuotes = CyanMetaobject.removeQuotes(s);
        if ( !s.equals(sWithoutQuotes) ) {
          System.out.print("parameter␣is␣string,␣it␣was␣given␣with␣quotes:␣" +
                sWithoutQuotes);
        }
        else {
          System.out.println("parameter␣is␣an␣identifer:␣" + s);
        }

      }
      else if ( obj instanceof Object[] ) {
        /*
         * literal arrays and tuples are all converted to a Java array
         * of type Object []
         *
         */
        Object []array = (Object []) obj;
        System.out.print("␣[␣");
        int size = array.length;
        for (Object elem : array ) {
          System.out.print(elem);
          if ( --size > 0 ) System.out.print(",␣");
        }
        System.out.println("␣]␣");
      }
    }
//    this.addError("This is the error message");
//    this.addError(annot.getFirstSymbol(), "This is the error message");
//    this.addError(pu.getFirstSymbol(env), "This is the error message");
//    this.addError(pu.getMethodDecList(env).get(0).getFirstSymbol(env),
//        "This is the error message");

    CyanMetaobjectWithAt metaobject = this;
      cyanruntime.Ref<Integer> breakCount = new cyanruntime.Ref<>();
      breakCount.elem = 0;

    pu.accept( new WrASTVisitor() {
        @Override
      public void visit(WrStatementBreak node, WrEnv env2) {
          ++breakCount.elem;
          metaobject.addError(node.getFirstSymbol(),
              "'break'␣are␣not␣allowed␣in␣prototype␣"
              + pu.getFullName());
      }

    }, env);


    MyASTVisitor visitor = new MyASTVisitor();
    pu.accept(visitor, env);
    if ( visitor.ifCount > 3 ) {
      this.addError("This␣prototype␣should␣not␣have␣more␣than␣2␣'if'␣statements");
    }

    pu.accept( new WrASTVisitor() {
        @Override
      public void visit(WrMethodDec node, WrEnv env2) {
          System.out.println(node.getName());
      }

    }, env);



    if ( pu.getMethodDecList(env).size() > 0 ) {
      pu.getMethodDecList(env).get(0).accept(
          new WrASTVisitor() {
            @Override
          public void visit(WrStatementRepeat node, WrEnv env2) {
              metaobject.addError(node.getFirstSymbol(),
              "'repeat-until'␣statements␣are␣not␣allowed␣in␣"
              + "the␣first␣method");
          }

        }, env);

    }
  }


}
```

```
class MyASTVisitor extends WrASTVisitor {
  public CyanMetaobjectWithAt metaobject;
  public int ifCount = 0;
  public int whileCount = 0;

  @Override
  public void visit(WrStatementWhile node, WrEnv env) {
    metaobject.addError(node.getFirstSymbol(), "’while’␣statements␣are␣not␣allowed")
  }
}
```

B CyanMetaobjectTwo

```
package metaobj;

import java.util.List;
import meta.AnnotationArgumentsKind;
import meta.AttachedDeclarationKind;
import meta.CyanMetaobject;
import meta.CyanMetaobjectWithAt;
import meta.IAction_afti;
import meta.ICheckDeclaration_afsa;
import meta.ICompiler_afti;
import meta.ICompiler_dsa;
import meta.IDeclaration;
import meta.ISlotInterface;
import meta.Tuple2;
import meta.WrASTVisitor;
import meta.WrCyanMetaobjectAnnotation;
import meta.WrCyanMetaobjectWithAtAnnotation;
import meta.WrEnv;
import meta.WrProgramUnit;
import meta.WrStatementBreak;

public class CyanMetaobjectTwo extends CyanMetaobjectWithAt
  implements IAction_afti, ICheckDeclaration_afsa {

    public CyanMetaobjectTwo() {
        super("two", AnnotationArgumentsKind.ZeroOrMoreParameters,
                new AttachedDeclarationKind[] {
                AttachedDeclarationKind.PROTOTYPE_DEC
        });
    }

  @Override
  public Tuple2<StringBuffer, String> afti_codeToAdd(
      ICompiler_afti compiler,
      List<Tuple2<WrCyanMetaobjectAnnotation,
      List<ISlotInterface>>> infoList) {

    WrCyanMetaobjectWithAtAnnotation annot =
        this.getMetaobjectAnnotation();
    List<Object> paramList = annot.getJavaParameterList();
    if ( paramList.size() < 2 || paramList.size() > 3 ) {
      this.addError("This␣annotation␣should␣have␣two␣or␣"
          + "three␣parameters");
    }
    if ( !(paramList.get(0) instanceof String) ||
        !(paramList.get(1) instanceof String) ) {
      this.addError("The␣two␣first␣parameters␣to␣this␣annotation␣"
          + "should␣be␣strings");
    }
    return new Tuple2<>(
        new StringBuffer( CyanMetaobject.removeQuotes( (String ) paramList.get(0)) )
        CyanMetaobject.removeQuotes( (String ) paramList.get(1)) );
  }


    @Override
    public void afsa_checkDeclaration(ICompiler_dsa compiler) {

        IDeclaration dec = this.getAttachedDeclaration();
        WrProgramUnit pu = (WrProgramUnit ) dec;
        WrEnv env = compiler.getEnv();


    WrCyanMetaobjectWithAtAnnotation annot =
        this.getMetaobjectAnnotation();
    List<Object> paramList = annot.getJavaParameterList();

    boolean checkBreak = false;

    if ( paramList.size() == 3 ) {
      if ( !(paramList.get(2) instanceof String) ) {
        this.addError("The␣third␣parameter␣to␣this␣annotation␣"
            + "should␣be␣a␣string");
      }
      String strBreak = (String ) paramList.get(2);
      /*
       * the parameter may be give as an identifer, without
       * quotes, or with quotes. Both are considered strings.
             @two( "var Int count = 0;",
                   "var Int count",
                   break )   // no quotes

             @two( "var Int count = 0;",
                   "var Int count",
                   "break" )    // with quotes

         So, if there are quotes, strBreak, in this case,
         would be
              "\"break\""
         So let us remove the quotes, if they exist
       */
      strBreak = CyanMetaobject.removeQuotes(strBreak);
      if ( strBreak.equalsIgnoreCase("break") ) {
        checkBreak = true;
      }
    }

    if ( checkBreak ) {
      /*
       * we show TWO ways of checking if there is a 'break' statement
```

```
       */

          cyanruntime.Ref<Integer> breakCount = new cyanruntime.Ref<>();
        breakCount.elem = 0;

      pu.accept( new WrASTVisitor() {
          @Override
        public void visit(WrStatementBreak node, WrEnv env2) {
            ++breakCount.elem;
        }

      }, env);
      if ( breakCount.elem != 0 ) this.addError("'break'␣statement␣is␣not␣allowed");

      BreakCountASTVisitor bcount = new BreakCountASTVisitor();
      pu.accept(bcount, env);
      if ( bcount.breakCount != 0 )
        addError("'break'␣statement␣is␣not␣allowed");

      /*
       * Of course, if checkBreak is true and there is a break,
       * TWO error messages will be issued
       */
    }

    }
}

class BreakCountASTVisitor extends WrASTVisitor {
    public int breakCount = 0;

    @Override
    public void visit(WrStatementBreak node, WrEnv env) {
        ++breakCount;
    }
}
```

C CyanMetaobjectThree

```
package metaobj;

import meta.AnnotationArgumentsKind;
import meta.AttachedDeclarationKind;
import meta.CyanMetaobjectWithAt;
import meta.ICheckOverride_afsa;
import meta.ICompiler_dpa;
import meta.ICompiler_dsa;
import meta.IParseWithCyanCompiler_dpa;
import meta.Token;
import meta.WrASTVisitor;
import meta.WrEnv;
import meta.WrMethodDec;
import meta.WrStatementBreak;
import meta.WrStatementIf;
import meta.WrStatementWhile;

/**
```

```
 * The annotation of this metaobject should be attached to a method.
 * When it is overridden in a subprototype, the metaobject checks
 * the code of the subprototype method based on the code of the attached
 * DSL. The annotation accepts an attached DSL with the syntax:<br>
 * <code><br>
 *     Start :: = Stat { "," Stat }<br>
 *     Stat :: = "if:" N | "while:" N | "repeat:" N <br>
 *                 | no break<br>
 *  </code><br>
 *
 * N is an integer number. The semantics is: if the DSL code has
 * stat: N, then  'stat' can only appear in the code N times.
 * If 'no break' is present, the code should have no 'break'
 * statement. Then, if the annotation is<br>
 * </code<br>
         {@literal @}three<*<br>
          if: 2, <br>
          no break<br>
         *><br>
   <code>
   the overridden method can have any number of while and repeat statements but
   only two if statements. And no breaks.


   @author jose
 */
public class CyanMetaobjectThree extends CyanMetaobjectWithAt
    implements IParseWithCyanCompiler_dpa, ICheckOverride_afsa {

    public CyanMetaobjectThree() {
        super("three", AnnotationArgumentsKind.ZeroOrMoreParameters,
                new AttachedDeclarationKind[] {
                AttachedDeclarationKind.METHOD_DEC
        });
    }

    private int numberIFsAllowed = -1;
    private int numberWHILEsAllowed = -1;
    private boolean allowBreaks = true;


    @Override
  public boolean shouldTakeText() { return true; }

  @Override
  public void dpa_parse(ICompiler_dpa compiler_dpa) {
    compiler_dpa.next();
    while ( compiler_dpa.getSymbol().token != Token.EOLO ) {

      switch ( compiler_dpa.getSymbol().token ) {
      case IDENT:
        String id = compiler_dpa.getSymbol().getSymbolString();
        if ( !id.equalsIgnoreCase("no") ) {
          compiler_dpa.error(compiler_dpa.getSymbol(),
              "'no'␣expected");
        }
        compiler_dpa.next();
        if ( compiler_dpa.getSymbol().token != Token.BREAK ) {
          compiler_dpa.error(compiler_dpa.getSymbol(),
              "'break'␣expected");
        }
//        id = compiler_dpa.getSymbol().getSymbolString();
//        if ( !id.equalsIgnoreCase("break") ) {
//          compiler_dpa.error(compiler_dpa.getSymbol(),
//              "'break' expected");
//        }
        allowBreaks = false;
        compiler_dpa.next();
        break;
      case IDENTCOLON:
        switch ( compiler_dpa.getSymbol().getSymbolString() ) {
        case "if:" :
          compiler_dpa.next();
          if ( compiler_dpa.getSymbol().token != Token.INTLITERAL ) {
            compiler_dpa.error(compiler_dpa.getSymbol(),
                "Literal␣int␣expected");
          }
          this.numberIFsAllowed = Integer.parseInt(
              compiler_dpa.getSymbol().getSymbolString());
          compiler_dpa.next();
          break;
        case "while:" :
          compiler_dpa.next();
          if ( compiler_dpa.getSymbol().token != Token.INTLITERAL ) {
            compiler_dpa.error(compiler_dpa.getSymbol(),
                "Literal␣int␣expected");
          }
          this.numberWHILEsAllowed = Integer.parseInt(
              compiler_dpa.getSymbol().getSymbolString());
          compiler_dpa.next();
          break;
        default:
          compiler_dpa.error(compiler_dpa.getSymbol(),
            "'if:'␣or␣'while:'␣expected.␣Found␣'" +
             compiler_dpa.getSymbol().getSymbolString() + "'");
        }
        break;
      default:
        compiler_dpa.error(compiler_dpa.getSymbol(),
            "'if:',␣'while:'␣or␣'no'␣expected");
      }
      if ( compiler_dpa.getSymbol().token != Token.COMMA ) {
        break;
      }
      else {
        compiler_dpa.next();
      }
    }
    if ( compiler_dpa.getSymbol().token != Token.EOLO ) {
      compiler_dpa.error(compiler_dpa.getSymbol(), "End␣of␣file␣expected");
    }
  }
  @Override
  public void afsa_checkOverride(ICompiler_dsa compiler_dsa,
      WrMethodDec method) {
    WrEnv env = compiler_dsa.getEnv();
    StatCountASTVisitor visitor = new StatCountASTVisitor();
    method.accept(visitor, env);
    if ( this.numberIFsAllowed >= 0 && visitor.ifCount > this.numberIFsAllowed ) {
      this.addError(method.getFirstSymbol(env),
          "This␣method␣has␣" + visitor.ifCount + "␣'if'␣statements."
              + "␣The␣number␣allowed␣is␣" + this.numberIFsAllowed);
    }

    if ( this.numberWHILEsAllowed >= 0 && visitor.whileCount > this.numberWHILEsAllo
      this.addError(method.getFirstSymbol(env),
          "This␣method␣has␣" + visitor.whileCount + "␣'while'␣statements."
              + "␣The␣number␣allowed␣is␣" + this.numberWHILEsAllowed);
    }

    if ( ! this.allowBreaks && visitor.breakCount > 0) {
      this.addError(method.getFirstSymbol(env),
          "This␣method␣has␣a␣'break'␣statement␣but␣none␣is␣allowed");
    }
  }

}


class StatCountASTVisitor extends WrASTVisitor {
    public CyanMetaobjectWithAt metaobject;
    public int ifCount = 0;
    public int whileCount = 0;
    public int breakCount = 0;

    @Override
    public void visit(WrStatementWhile node, WrEnv env) {
        ++whileCount;
    }

    @Override
    public void visit(WrStatementIf node, WrEnv env) {
        ++ifCount;
    }

    @Override
    public void visit(WrStatementBreak node, WrEnv env) {
        ++breakCount;
    }

}
```

D CyanMetaobjectFour

```
package metaobj;

import java.util.List;
import meta.AnnotationArgumentsKind;
import meta.AttachedDeclarationKind;
import meta.CyanMetaobject;
import meta.CyanMetaobjectLiteralObject;
import meta.CyanMetaobjectWithAt;
import meta.IAction_dsa;
import meta.ICompiler_dsa;
import meta.WrCyanMetaobjectWithAtAnnotation;
import meta.WrEnv;
import meta.WrMethodDec;
import meta.WrProgramUnit;

public class CyanMetaobjectFour extends CyanMetaobjectWithAt
    implements IAction_dsa {

  public CyanMetaobjectFour() {
    super("four", AnnotationArgumentsKind.OneParameter,
        new AttachedDeclarationKind[] {
            AttachedDeclarationKind.NONE_DEC
    });
  }


  @Override
  public String getPackageOfType() { return "cyan.lang"; }
  /**
   * If the metaobject annotation has type <code>packageName.prototypeName</code>, t
   * <code>prototypeName</code>.  See {@link CyanMetaobjectLiteralObject#getPackageO
     @return
   */

  @Override
  public String getPrototypeOfType() {
    return "Int";
  }

  @Override
  public boolean isExpression() {
    return true;
  }

  @Override
  public StringBuffer dsa_codeToAdd(ICompiler_dsa compiler_dsa) {

    int value = 0;
    WrEnv env = compiler_dsa.getEnv();
    WrProgramUnit pu = env.getCurrentProgramUnit();
    WrMethodDec currentMethod = env.getCurrentMethod();

    WrCyanMetaobjectWithAtAnnotation annot = this.getMetaobjectAnnotation();
    List<Object> paramList = annot.getJavaParameterList();
    Object obj = paramList.get(0);

    if ( !(obj instanceof String) ) {
      this.addError("The␣parameter␣to␣this␣annotation␣should␣be␣a␣string");
    }
    String s = (String ) obj;
    s = CyanMetaobject.removeQuotes(s);
    switch ( s ) {
    case "method":
      value = pu.getMethodDecList(env).size();
      break;
    case "field":
      value = pu.getFieldList(env).size();
      break;
    case "statement":
      value = currentMethod.getStatementList(env).getStatementList().size();
      break;
    case "parameter":
      value = currentMethod.getMethodSignature().getParameterList().size();
      break;
    case "interface":
      value = pu.getInterfaceList(env).size();
      break;
    default:
      this.addError("Illegal␣parameter:␣’" + s + "’" );
    }
    return new StringBuffer("" + value);
  }


}
```