Metaprogramming in Cyan

José de Oliveira Guimarães[1,]

[a]Department of Computer Science at Sorocaba, UFSCar, Rod. João Leme dos Santos, Km 110, Sorocaba, 18052-780, São Paulo, Brazil

Abstract

Certain languages allow a metaprogram to act as a compiler plugin and thus alter the compilation process. The metaprogram interacts with low-level details of the compiler, making its construction difficult and potentially leading to errors. Different parts of the metaprogram may have conflicting interactions, thus producing unintended outcomes. This article introduces metaprogramming in the prototype-based object-oriented language Cyan. This language provides the same core functionality as other metaprogramming systems, while additionally offering several features that facilitate interactions between the compiler and the metaprogram, as well as between different components of the metaprogram. Furthermore, Cyan incorporates security measures designed to circumvent typical issues encountered in metaprogramming.

Keywords: object-oriented languages, metaprogramming, metaobject, computational reflection, prototype-based languages, compilers

1. Introduction

Metaprogramming is the coding of programs, called *metaprograms*, that treat code as data (Lilis and Savidis, 2019). The *base program*, or simply the *program*, is the program that is treated as *data*. A *metaprogram* can generate new code, change existing programs, or do checks on them. Metaprogramming offers mechanisms for code reuse that go beyond those offered by traditional software libraries. It can generate families of related code, as in the case of C++ templates (Stroustrup, 2013); separate functional and nonfunctional concerns, as in AspectJ (Kiczales et al., 2001); generate code based on specifications, as ANTLR 4 does (Parr, 2013); support new syntax, such as Scala macros (Burmako, 2013); detect program bugs through static analyzers[1]; implement new type systems using a pluggable-type system (Bracha, 2004); run a program in multiple stages (Taha, 2007), each stage generating and running a new program; change the program at runtime (Redmond and Cahill, 2002; Kamin et al., 2003); and support embedded Domain-Specific Languages (Rompf et al., 2012; Biboudis et al., 2016).

In this paper, the focus is *language support* for Compile-Time Metaprogramming (CTMP), which is the handling of a program by a metaprogram at compile-time. To discuss specific characteristics of compile-time metaprogramming supported by programming languages, we will define some terms. The *program*, or *base code*, is the code that implements the desired

Email address: `josedeoliveiraguimaraes@gmail.com, jose@ufscar.br` (José de Oliveira Guimarães)

[1]https://github.com/google/error-prone, https://spotbugs.readthedocs.io/en/stable/

functionality for the application. A *metacode* is each of the pieces of code that compose the *metaprogram*. The *metacode* is loaded at the compilation time of the *base code* and works as an integral part of the compiler. Data is exchanged between the compiler and the metacode. The compiler calls metacode at specific points of compilation. Therefore, metacode can interact with the type checker, code generator, parser, and any other algorithm used by the compiler. They can also add, delete, or replace code in the *program*. In practice, languages restrict what metacode can do to a few things. The metaprogram is designed to help the program achieve the desired functionality.

Metacode interacts closely with the compiler. This interaction is precisely defined through a *protocol*, which specifies which part of each metacode is called in a given compilation phase and how data is passed from and to the compiler. For example, the protocol may specify that during the compilation phase *parsing*, a function[2] or method `duringParsing` may be called. The function or method may perform checks or add code to the program. The protocol would also specify which parameters the function takes, what it returns, and how the return changes the compilation.

Although every language with support for CTMP has a protocol for the interactions between the metacode and the compiler, some of them are said to have a Metaobject Protocol (MOP) by their designers. These are usually older languages and have some characteristics in common that are discussed throughout the paper. Let us evaluate languages supporting CTMP with or without a MOP to show their deficiencies and opportunities for improvement. Some mechanisms not directly related to the proposal of the paper, such as aspects (Kiczales et al., 2001), will only be discussed later. A typical Metaobject Protocol associates a metaclass to every class and, in general, a meta-S to every kind of declaration S. Developers can define their own metaclass for a class, which directs the compilation of the class. Using a metaclass, one can add code to the class, do additional checks on it, and change the semantics of the language for that class. For example, the metaclass can check if all class fields are read-only, intercept object creation, and change the meaning of inheritance. However, there are limitations. Usually, only one metaclass is associated with a class and the composition of behavior is difficult. Arguments cannot be passed to the metaclass, which makes its configuration impossible. There may be limitations on what can be changed, as in OpenC++ (Chiba, 1995) or OJ (Tatsubori et al., 2000). Or there may be very few limitations, as in CLOS, but the AST of the class associated with the metaclass is not available.This makes building the metaclass more difficult.

A language may support CTMP without a MOP such as Groovy (König, 2007), Scala[3], or Java[4]. In general, support for CTMP is at a much lower level than when using a MOP. The metacode developer has to interact with low-level compiler algorithms and data structures. It is easy for the metacode to invalidate compiler invariants or damage internal compiler structures, making the compiler crash.

There are some problems that occur with metaprogramming languages with or without a MOP, although mainly with the latter. Not all problems that follow occur in all languages, although most of them do occur in languages without a MOP. The following list is not exhaustive.

(a) A metacode associated with a source file can change the base code of another file. Hence, to know the final version of a file, the developer has to know the behavior of every metacode in the metaprogram.

[2]Function as in language C

[3]https://docs.scala-lang.org/scala3/reference/changed-features/compiler-plugins.html

[4]https://docs.oracle.com/en/java/javase/11/docs/api/jdk.compiler/com/sun/source/util/Plugin.html

As an example, suppose the metacode associated with files X and Y change file Z. To know the final code of Z, the developer has to know these associations *and* the runtime behavior of the metacode, which are run at compile time for the base code.

(b) A class may not have access to the private fields and methods of another class. But the metacode can because it has access to the compiler's data structures.

As an example, a metacode associated with class `A` creates a method

```
    String toJSON(B b)
```

The metacode checks whether all fields of `b` have get methods and uses these methods to return the JSON string of the argument. If class `B` changes only its private fields, `A` will not be recompiled. Therefore, the `toJson` method created by the metacode will be incorrect.

(c) The developer has to know many details about the compiler since the metacode interacts with low-level compiler algorithms and data structures. Therefore, metaprogramming is not only harder, but the metacode can bypass compiler checks, invalidate invariants, damage data structures, and crash the compiler.

As an example, the metacode may change the AST after the compiler type-checked the code. The changed AST may have type errors, which may crash the compiler during the following compilation phase.

(d) Metacode can insert new code into the base code. If the new code has errors, the compiler is unable to point out which metacode is responsible for adding the new code. This is because the metacode itself changes the compiler AST to add new AST objects. No trace is left of who did what.

As an example, suppose the metacode `M` inserts a new statement to method `search` for logging purposes:

```
    searchMethodStat.add( new MethodCall(...) );
```

The compiler does not record that the statement "`MethodCall`" was inserted by `M`. If it has errors, the compiler does not point out this metacode as its source.

(e) The order in which the metacode is called is not specified by the language. It can even change between compilations (with the same base code). By changing the metacode calling order, their view of the program can change if a metacode can view the code added by a metacode executed previously.

As an example, the metacode may be specified by a compiler option of a ficticious language Sumatra:

```
    compiler -processor P -processor Q A.sumatra
```

Metacode `P` adds a field `count` to the class `A` and `Q` creates get and set methods for all `A` fields. This works fine because `P` is executed before `Q` (assume this). If we reverse the `-processor` options, the get and set methods for `count` will not be created.

(f) A check made by one metacode may be invalidated by code added by another metacode.

As an example, a metacode P checks that all fields of a class `A` have get and set methods. After that, another metacode Q adds a field `ghost` but not get and set methods. The check that P made was invalidated by Q.

(g) Metacode may generate metacode that may generate metacode ad infinitum.

As an example, metacode `M` produces code containing references to `M` (as a macro `M` producing calls to `M`). If the compiler processes the newly created code, there will be an infinite creation of code.

(h) Metacode may not terminate its computation. The compiler will not terminate either.

No example is necessary for this problem.

(i) Metacode may unintentionally generate code that uses identifiers already in use. This is the hygiene problem of macros.

As an example, suppose a macro (Kohlbecker et al., 1986) `compTimeFactorial(r, num)` produces the following code.

```
r = 1;
for (int n = 1; n <= num; ++n) r = r*n;
```

If we use `compTimeFactorial(r, n)` in which `n` is a local variable, the `for` statement will run only once.

The goal of this article is to present the Metaobject Protocol (MOP) of the statically typed prototype-based object-oriented language Cyan (Guimarães, 2020). The metacode is composed of *metaobjects*, which are associated with *annotations* in the base code. *Metaobjects* can add code to the program, which includes new prototypes, fields and methods to prototypes, and statements and expressions to methods. Besides that, they can intercept message passing, field access, subprototyping, method overriding, etc. *Metaobjects* in Cyan have limited power; they cannot delete program code or replace any compiler algorithm as the type checker. Additional checks can be added to a program, but none can be bypassed.

The contribution of this article is to show how the Cyan MOP achieves a good balance between power, security, and ease of use. It combines a full MOP, like that of CLOS, with metaprogramming features of more recent languages, such as Groovy and BSJ. The Cyan MOP is powerful enough to allow the implementation of the great majority of metacodes or metaobjects we have found in the literature. It solves, at least partially, all of the problems listed previously with CTMP, except those related to nondeterminism and hygiene. Finally, a Metaobject Protocol that is sufficiently powerful cannot be simple as it must interact with a complex software component, namely the compiler. But this article tries to argue that some characteristics of the Cyan MOP make it simpler than most CTMPs with similar power. These are the *declarative* way of building metaobject classes in which the compiler calls the metacode and not the opposite (as usual in CTMP without a MOP), a simplified read-only AST and other compiler data structures, code generation using strings (instead of the more complex AST objects), and security measures that prevent the common problems with CTMP.

The Cyan MOP has two main goals. The first one is to build general-use metaobjects such as `property` to create get and set methods for a field, `annot` to associate information to declarations, `checkStyle` to verify the style of code, several metaobjects that give information on the source code (as the line number of the annotation), `eval` to evaluate code at compile-time, `deprecated` to issue a warning if a deprecated declaration is used, and many others.

The second goal of the Cyan MOP is to build metaobjects associated with developers' packages (libraries). These metaobjects would generate code and check the use of the packages. As examples of code generation, package `cyan.lang` uses metaobjects to generate code for several generic prototypes, such as `Tuple` and `Array`. Based on the real argument, such as `Int` in `Array<Int>`, metaobjects add some methods to `Array<Int>`.

The semantics of methods, which are usually written in the documentation, can be put in metaobjects that do checks at compile-time. An example of that is method `printf` of prototype `Out`. Similar to the function with the same name in language C, this method accepts a format string as the first argument. A metaobject checks at compile-time if the arguments match the first argument. The semantics of inheritance can also be checked by metaobjects. Annotation

`@overrideToo("hashCode")` attached to method == of the top Cyan prototype, `Any`, demands that, whenever == is overridden in a subprototype, method `hashCode` should be overridden too. A metaobject whose annotation is attached to a method may request that this method be called in every method that overrides it in a subprototype. These examples show that packages' semantics can be checked by metaobjects.

The paper's organization is as follow. Section 2 is a brief introduction to the Cyan language. The Metaobject Protocol of Cyan is explained in Section 3. Section 4 compares the metaprogramming systems of other languages with the Cyan MOP. The last section concludes.

Additional information on the Cyan compiler and the language is available at `cyan-lang.org`.

2. The Cyan Language

Cyan is a statically typed prototype-based object-oriented language. A *prototype* is a template from which other objects may be created, the same role is played by *classes* in Java (Gosling et al., 2014), C++ (Stroustrup, 2013), C♯[5], and Smalltalk (Goldberg and Robson, 1983). The difference is that the prototype itself is an object like any other.[6]

2.1. Basic Elements of the Language

The look and feel of Cyan is that of a class-based language. The compiler translates Cyan into non-legible Java code. Thus, many language constructs are directly translated into Java, such as packages, inheritance, method overriding, message passing, assignment, and prototype declaration (each prototype is translated to a Java class). Cyan code can import Java packages and classes. The compiler does all the necessary conversions between names and values of the basic types.

Listing 1 shows the declaration of prototype `Student` of the package `university`. Cyan employs a syntax for method declaration and message passing that is in some ways similar to Smalltalk. A *unary message passing* is made up of a *receiver* and an identifier, which should be the name of the *unary method*:

```
aStudent getName
```

`aStudent` is the *message receiver*. A *keyword message passing* is composed of a *receiver* and one or more *message keywords*, or just *keywords* with their parameters:

```
var Array<String> as = Array<String> new;  // creates an object
 // 'add:' is a message keyword, a method 'add:' is called
as add: "first";
 // 'at:' and 'put:' are keywords, method 'at:put:' is called
as at: 0 put: "zero";
```

Both *method keywords* and *message keywords* are called *keywords*. To avoid confusion, *Cyan keyword* is used for reserved words in the language.

2.2. Examples of Using Annotations

This Subsection gives a general view of how annotations are used in Cyan code and what metaobjects can do. An *annotation*, or *metaobject annotation*, is the syntax element that links the program to a metacode. There are several kinds of annotations in Cyan. This Subsection

[5]https://learn.microsoft.com/en-us/dotnet/csharp

[6]There are exceptions to this rule, but this is not important to this paper.

Listing 1: Prototype `Student`

```
package university

object Student
    // fields name and number
   var String name
   var Int    number
    // this is a constructor. Use 'Student new: name, number'
    // or 'Student(name, number)' to create an object of Student
   func init: String name, Int number {
      self.name = name;
      self.number = number;
   }
   func getName −> String     = name;
   func setName: String name  { self.name = name }
   func getNumber −> Int      { return number }
   func setNumber: Int number { self.number = number }
end
```

Listing 2: Prototype `Student` with annotations

```
package university

@init(name, number)
object Student
   @property var String name
   @property var Int number
end
```

will only give examples of the most general of them all, that which starts with "`@`", as shown in Listing 2. At compile-time, each annotation is linked to a single *metaobject* (and vice versa). The metaobject is able to do checks, create new prototypes, and add code to the current prototype. In this example, the metaobject associated with `init` will add a constructor to the prototype `Student` to initialize the newly created object with the student's name and number. The two metaobjects associated with the two `property` annotations will add get and set methods to `Student`. The resulting prototype is identical to the one in Listing 1. We say that the two `property` annotations are *attached* to the declarations of `name` and `number`. The annotation `init` is *attached* to the prototype `Student`. Basic values (`3`, `3.14`, `'A'`), identifiers, literal arrays, literal tuples, literal maps, and any combination of these can be parameters for annotations. Annotation `init` takes two identifiers as arguments, `name` and `number`, which are treated as strings by the metaobject.

The *metaobjects associated with a prototype* `P`, or *metaobjects of* `P`, are the metaobjects associated with annotations of prototype `P`. There are three metaobjects associated with `Student` of Listing 2. We will use "*metaobject* `init`" when no confusion may arise. If there are two `init` annotations in a code, "*metaobject* `init`" will be ambiguous because it may refer to metaobjects associated with both annotations.

Annotations with an Attached DSL

The annotation of the following *ficticious*[7] example takes as arguments an identifier, `build`, and a multiline literal string (delimited by three `"`).

```
@buildGUI(build, """
  button {
     text "Next page",
     size (40, 100),
     onPressed goNextPage
  }
  """)
object MyButton ... end
```

Assume that the metaobject `buildGUI` adds a method to the prototype `MyButton` whose name is the first argument (`build` in this example). This method builds the graphical elements specified by the multiline string, which is the code of a DSL for building GUI (Graphical User Interfaces). Annotations whose last argument is a string with the code of a DSL are very common. There are so common that a special syntax was invented for them: the DSL code is put after the annotation arguments, between `{*` and `*}`, called "text delimiters".[8] Hence, the above example could be written as

```
@buildGUI(build){*
  button {
     text "Next page",
     size (40, 100),
     onPressed goNextPage
  }
*}
object MyButton ... end
```

The text between delimiteres will be called *attached text* or *attached Domain-Specific Language (DSL) code*.

A metaobject uses the *attached text* as any other string argument. Usually, annotations with the same name use the same DSL. So, all annotations with the name `buildGUI` would use the same DSL for building GUIs. One can imagine the annotation as the name of the DSL compiler that is called at compile-time to create code (as in the above example) or do checks. Different names mean different compilers and DSLs.

An annotation could support different DSLs. An example, not yet implemented, would be a flexible `flexDoc` annotation for documentation that should be attached to a prototype, method, and so on.

```
@flexDoc(html){*      /* documentation in HTML     */  *}
@flexDoc(markdown){*  /* documentation in Markdown */  *}
```

The annotation argument defines the DSL to be used with the attached text. This is unusual.

[7]This is just an example, there is no such metaobject.

[8]There are many possible variations of symbols, as described by Guimarães (Guimarães, 2022), so that the DSL and the delimiters do not clash.

Listing 3: Annotation `insertCode` adds methods to the current prototype

```
1  @insertCode{*
2    // adds to the prototype functions like
3    //    func power_num: Int n -> Int = n*n ... *n;
4    //  $ is the string interpolation, "= $s;" is equal to "= " ++ s ++ ";" in Java
5    for num in 2..10 {
6       var sig = "func power_$num: Int n -> Int ";
7       var s = "n";
8       //  ++ is concatenation of strings
9       for p in 2..num { s = s ++ "*n" }
10      insert: sig, sig ++ "= $s;"
11   }
12 *}
13 object Program
14   func run {
15     assert power_5: 2 == 32;   // 2^5 == 32
16     assert twice: 11 == 22;
17   }
18   @doc{* returns the double of the argument *}
19   @replaceCallBy(once){*  2*n  *}
20   func twice: Int n -> Int = n + n;
21 end
```

Annotation `insertCode` of Listing 3 takes an attached DSL code that is in the language *Myan*, which is a dynamically typed simplified version of Cyan. The DSL of annotation `replaceCallBy` (line 19) is a subset of Cyan in which only expressions are legal. Any text is valid for the `doc` annotation (line 18). It is intended as the documentation for the declaration to which the annotation is attached (`twice:` in this case).

The metaobject associated with `insertCode` interprets the Myan code between `{*` and `*}` at compile-time. The `insert:` method called in line 10 asks the compiler to insert methods into the current prototype. Its first parameter is the method signature (without the body), and the second is the complete method. For example, in the first `for` step (`num` is `2`), the call will be:

```
insert: "func power_2: Int n -> Int ", "func power_2: Int n -> Int = n*n;"
```

The metaobject associated with the annotation `replaceCallBy` intercepts all calls to the method `twice:` and replaces them with the expression between `{*` and `*}`. The method argument is only evaluated once because of the `once` argument to `replaceCallBy`. Annotations can be used inside expressions if they were designed with this goal.

```
var fatorial_of_10 = @eval("cyan.lang", "Int"){*
  var r = 2;
  for n in 3..10 { r = r*n }
  return r
*};
```

In this example, `eval` interprets the Myan code and returns `10!` at compile-time. Java packages

Listing 4: Annotation `onOverride` whose code is interpreted whenever the attached method is overridden

```
@onOverride{*
  if (method getStatementList: env) getStatementList size < 10 {
    metaobject addError: "method test should have at least 10 statements"
  }
*}
func test { }
```

Listing 5: The generic prototype with varying number of parameters `Tuple`

```
package cyan.lang
@createTuple
object Tuple<T+>
end
```

can be imported and used within Cyan and Myan code. Therefore, one could read a file or open a web connection and create, based on it, a bunch of methods or fields in the current prototype.

Myan code has access to several compiler objects through the variables `method`, `metaobject`, and `env`. Listing 4 shows an example where the Myan code attached to the annotation `onOverride` is interpreted whenever the attached method (`test`) is overridden in a subprototype. It issues an error if the method has fewer than `10` statements.

Generic Prototypes and Metaobjects

Generic prototypes in Cyan take one or more parameters.

object GroupList<T> ... **end**

`T` is a *generic parameter* used inside `GroupList` in any place a type is expected: as the type of variables, parameters, fields, the return type of methods, and inside expressions. A generic prototype is *instantiated* when real arguments are supplied to it.

var GroupList<GroupElem> groupList;

Assume that `GroupElem` is a prototype. *Instantiation* is the process of creating a new prototype by replacing, textually, the *generic parameters* with the real arguments. In this example, `T` is textually replaced by `GroupElem`. A new source file is created and compiled. Therefore, the semantics of Cyan *generic prototypes* is similar to that of C++ *class templates*; a new prototype is created for each set of real arguments.

A generic prototype with a varying number of generic parameters has just one parameter followed by `+`, as shown in Listing 5. This is the real code of the prototype `Tuple` that is used as a superprototype of literal tuples that can have any number of elements of different types. Parameter `T` cannot be used inside the prototype using the Cyan syntax because it represents the list of types passed as arguments, and there is no syntax to access such a list. But the metaobject associated with the annotation `createTuple` has access to the list of real arguments to the instantiation and uses it to generate code. For example, for the instantiation `Tuple<Int, Char>`, the metaobject adds to the prototype methods `f1 -> Int` and `f2 -> Char`.

Concepts (Stroustrup, 2003) are constraints on real arguments for generic classes, functions, and prototypes, an idea that originated in CLU (Liskov et al., 1977). A concept checks whether a real-type argument to a generic prototype, for example, is valid before instantiating the prototype with that type. For example, the `GroupList` prototype assumes that its type argument has a method `unit`. If `GroupList` is instantiated with a type that does not support this method, such as `Int`, the compiler will create a new prototype `GroupList<Int>`, compile it, and only then will it issue an error. However, a concept can check whether the type argument, `Int` in this case, has a method `unit` before the instantiation.

Concepts were devised to help the compiler issue clearer error messages during the instantiation of a template class in C++.[9] Concepts are so important that they are supported by a plethora of languages, including G (Siek and Lumsdaine, 2011), JavaGI (Wehr and Thiemann, 2011), Java (Gosling et al., 2014), Scala (Odersky et al., 2021) (Pelenitsyn, 2015), C♯^{cpt} (Belyakova and Mikhalkovich, 2015), Haskell (Wadler and Blott, 1989), Rust (Klabnik and Nichols, 2022), C♯ (Csh, 2023), Swift[10], and Genus (Zhang et al., 2015).

Concepts are implemented in Cyan using the metaobject `concept`, without any help from the language itself. The DSL code attached to the annotation specifies the restrictions that the generic parameters should obey. In the example that follows, `T` is required to define three methods: `unit`, `*`, and `inverse`, with the given signatures.

```
@concept{*
   T has [ func unit -> T, "Type T must supply a 'unit' method"
         func * T -> T
         func inverse -> T  ]
*}
object GroupList<T> ...  end
```

The DSL of the code attached to the `concept` annotation has statements for requiring that a prototype inherits another, a prototype implements another interface, that a parameter is an interface or a non-interface, a prototype declares a set of methods (used in the above example), a prototype belongs to a set of prototypes, and the negation of every of these statements. In the above example, there is a string after "`func unit -> T`". This string is a tailored error message. If we use `GroupList<MyGroup>` and `MyGroup` does not have a method `unit`, the metaobject will issue this error message.

3. The Cyan Metaobject Protocol

The Cyan Metaobject Protocol (MOP) describes the *interactions* between the Cyan code being compiled, the compiler, the MOP library, the metaprogram, and annotations in the Cyan code. The metaprogram in Cyan is composed of Java classes, Cyan prototypes, or a mixture of both. The compiler is implemented in Java making it convenient to use Java classes as the metaprogram. But since the compiler translates each Cyan prototype into a Java class, Cyan can also be used as the metaprogramming language.

Before presenting the MOP in detail, Subsection 3.1 shows how to build a metaobject class and how to use metaobject annotations. The interactions between the metacode and the compiler are presented in Subsection 3.2. Subsection 3.3 explains the phases of the Cyan compiler. The important features of the Cyan MOP are summarized in Subsection 3.4. Other kinds of annotation

[9]https://www.iso.org/standard/79358.html
[10]https://docs.swift.org/swift-book

Listing 6: Annotation `get` attached to field `name`

```
1 package art
2 import cyanPack
3 object Painting
4   func init: String name { self.name = name }
5   @get String name
6 end
```

are introduced in Subsection 3.5. Annotations can be attached to types too, as explained in Subsection 3.6.

3.1. An Example of a Metaobject

This Subsection explains how to build a metaobject `get` whose annotation should be attached to a prototype field. The metaobject adds to the prototype a method `get_field` that just returns the attached field. Listing 6 shows annotation `get` attached to field `name` in line 5 and, therefore, the following method is created and added to `Painting`.

```
func get_name -> String = name;
```

This method can be called as if it were defined in the prototype.

```
var p = Painting("Kandinsky");
p get_name println;
```

We will give step-by-step instructions on how to build metaobject `get`. First, create a Java file that declares a class `CyanMetaobjectGet` that inherits from `CyanMetaobjectAtAnnot`. Next, select an interface to implement based on the goals of the metaobject. Use Table 1 (end of the text). This table relates goals and the interfaces the metaobject class should implement to achieve them. As will be explained in detail later, the interfaces implemented by a metaobject class direct the compilation. They tell the compiler when (compilation phase) and which metaobject methods should be called.

The goal of metaobject `get` is to add a method. Table 1 says class `CyanMetaobjectGet` should implement interface `IAction_afterResTypes`. Based on the documentation of this interface, we choose to override the method `afterResTypes_codeToAdd` that is used for adding fields and methods to the current prototype. The complete metaobject class is in Listing 7 (without the imports), which will be explained.

The annotation name is `"get"`, it does not take parameters, and it should be attached to a field. This is all communicated in the call to `super` in lines 5-6. To code method `afterResTypes_codeToAdd` starting at line 9, we again remember the goals of the metaobject, which is to create a method whose name is based on the attached field. So, we need the name of the field attached to the annotation. Method `getAnnotation` in line 13 returns an object with information on the annotation. From it, the code can retrieve the attached declaration using method `getDeclaration` (a field in this case). Since, by line 6, we know that the attached declaration is always a field, the code can cast the object returned to class `WrFieldDec`. This is the AST class for Cyan prototype fields. If the annotation `get` took arguments, these could be retrieved using another method of the object returned by `getAnnotation`. The attached field name is got in line 14 and the method source code, as a string, is created in lines 15-16. The return

Listing 7: The `CyanMetaobjectGet` class in Java

```
1  public class CyanMetaobjectGet extends CyanMetaobjectAtAnnot
2           implements IAction_afterResTypes {
3
4   public CyanMetaobjectGet() {
5    super("get", AnnotationArgumentsKind.ZeroParameters,
6     new AttachedDeclarationKind[] { AttachedDeclarationKind.FIELD_DEC });
7   }
8   @Override
9   public Tuple2<StringBuffer, String> afterResTypes_codeToAdd(
10        ICompiler_afterResTypes compiler,
11        List<Tuple2<WrAnnotation, List<ISlotSignature>>> infoList) {
12
13   final WrFieldDec field = (WrFieldDec) this.getAnnotation().getDeclaration();
14   final String name = field.getName();
15   String methodSig = "func get_" + name + " -> " + field.getType().getFullName();
16   String method = methodSig + " = " + name + ";";
17   return new Tuple2<StringBuffer, String>(new StringBuffer(method), methodSig);
18  }
19 }
```

type of `afterResTypes_codeToAdd` is a tuple consisting of the source code of the fields and methods to be inserted in the current prototype (`Painting` in the example) and the signatures of these declarations. The signature of a method is the method without its body, and the signature of a field is its declaration without any initialization (`= expr`). If the compiler were smarter, the return value could be just the code to be inserted.

Class `CyanMetaobjectGet` should be compiled with the file `saci.jar` (available on the website `www.cyan-lang.org`). The resulting `.class` file should be put in a directory "`cyanPack\--meta\javaPack`" if class `CyanMetaobjectGet` is in the Java package `javaPack`. Now, any Cyan code that imports package `cyanPack` can use annotation `get`.

Let us study what happens in the compilation of `Painting` of Listing 6. At the start of the compilation, the compiler imports package `cyan.lang` (automatically) and `cyanPack` (line 2 of this example) and loads dynamically the classes of directories `--meta` of these packages. It then creates one object for each class. Using method `getName`, it has access to the annotation name of each metaobject, which was given in the call to `super` (line 5 of Listing 7). During parsing, when the compiler finds an annotation `get`, it looks for a metaobject called `"get"` in the imported packages. `CyanMetaobjectGet` is found, and the compiler creates an object of this class, which is associated with the annotation of line 5 of Listing 6.

In the compilation phase `afterResTypes` (to be seen later), the compiler calls the method `afterResTypes_codeToAdd`. If the returned value is not `null`, the returned code is inserted in the source code of the prototype, which needs to be compiled again. Only the text of the prototype in the compiler memory is changed. In the following compilation phases, the compiler will find a `get_name` method in prototype `Painting`.

A metaobject class seems difficult to build because it demands the knowledge of a lot of cyan MOP classes, which includes the AST. However, this is only partially true because the MOP and the IDE guide the developer throughout the process. Let us see why.

A metaobject class has a standard superclass, and its constructor should call `super` with the annotation name, information on the number of annotation arguments, and a list of allowed attached declarations. The Java compiler will demand that the developer pass this information to the superclass. Based on the goals of the metaobject, the developer uses Table 1 to discover which interfaces the metaobject class should implement. The documentation of each interface tells which methods should be overridden. The coding of a metaobject class method should be based on information on the annotation or the source code (the current prototype or method). Information on the annotation (attached declaration, arguments, or attached text between `{*` and `*}`) is obtained from the method `getAnnotation`. Information on the source code is obtained from an interface method parameter called `compiler` or `env`. For example, the parameter `compiler` of method `afterResTypes_codeToAdd` (line 10 of Listing 7). As an example, suppose we want to check if there is a method in the prototype with the same name as the method the metaobject will add. This can be done with the following code, which should be put after line 14.

```
for (WrMethodDec m : compiler.getMethodDecList()) {
  if ( m.getName().equals("get_" + name) ) {
    return null; // no method will be added
  }
}
```

"`compiler.getMethodDecList()`" returns a list of AST objects of class `WrMethodDec`, one for each method of the prototype in which the annotation is. There is no need to learn all MOP classes and methods. Using the "code completion" feature of the IDE (after typing "`param.`"), the developer has access to methods and, from them, to the classes she or he needs.

3.2. Interactions with the Compiler

The *metacode of a Cyan program* (code) is composed of all Java classes or Cyan prototypes of the metaobjects associated with the annotations used by the program. To support the metacode, there are two versions of a *MOP library* in the `saci.jar` file: one in Java and another in Cyan (one is a mirror of the other). The Cyan compiler knows both MOP libraries, and some of their classes and prototypes are superclasses or superprototypes of the classes and prototypes of the metacode. The class `CyanMetaobjectGet` of Listing 7 is part of the metacode of the Cyan program that has class `Painting`. This Java class imports its superclass from the MOP library that is in the `saci.jar` file, which is also part of the metacode.

When parsing source code, the Cyan compiler creates, for each annotation, an object of the AST internal to the compiler, which is represented by the round rectangle "annot. AST object" on the left side of Figure 1. This object represents the annotation as a syntactical element. Based on the annotation name, the compiler discovers the metaobject class associated with it, `CyanMetaobjectGet` in this case, and creates a metaobject, which is represented by the round rectangle on the left-bottom side of the Figure. The metaobject gets the annotation data, as well as its arguments and attached declaration, from a wrapper AST object. In the Figure, this wrapper AST object is represented by a round rectangle enveloping the rectangle "annot. AST object". Both objects will be called *AST objects*. Figure 1 also shows that there are one-to-one relationships between metaobjects, wrapper AST objects, annotation AST objects, and annotations (the syntactic element).

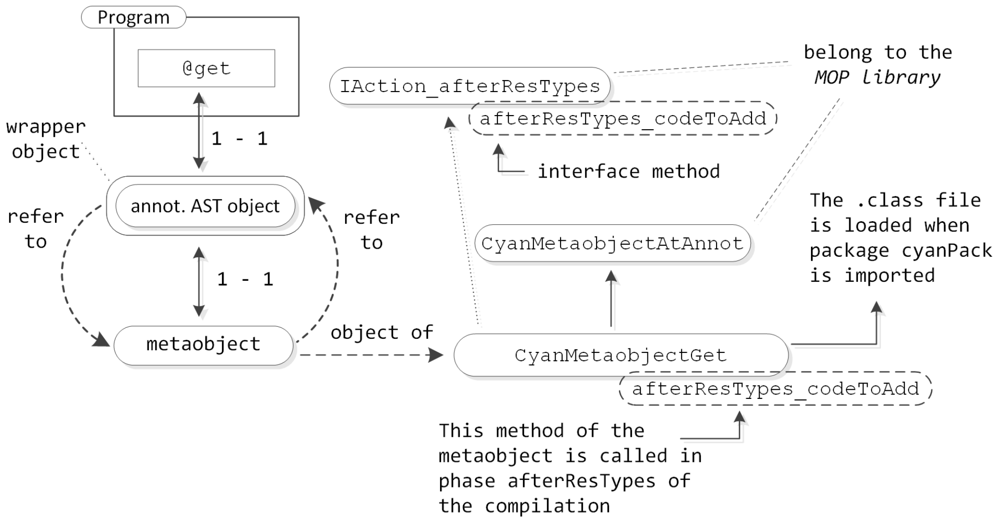

Figure 1: Relationship between metaobjects, annotations, metaobject classes, MOP interfaces, and compilation phases

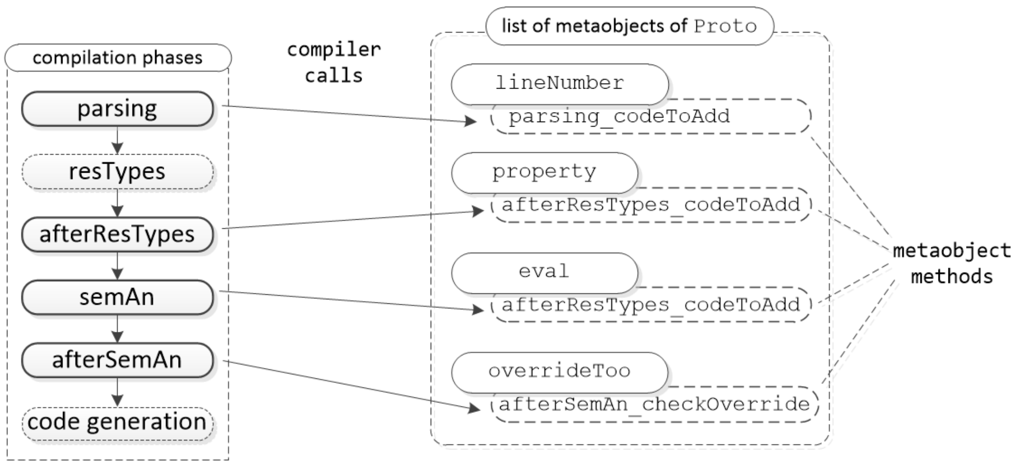

Figure 2: The compilation phases and their links to methods of metaobjects at compile time

3.3. Phases of Compilation

The Cyan compiler goes through six compilation phases for each source file, as shown inside the rectangle with dashed lines on the left of Figure 2. The flow of control is from top to bottom. Phase **parsing** does the syntactical analysis and builds the Abstract Syntax Tree (AST) of the source file. Some AST objects are associated with a type and have a `type` field that is initially set to `null`. A `type` field, for example, exists in AST objects representing method parameters, prototype fields, implemented interfaces, the superprototype, message passings, and expressions.

There are two kinds of AST objects associated with types: those representing expressions and local variable declarations, which are always inside method bodies, and those outside method bodies. The `type` field of the later AST objects is set in phase **resTypes** (resolving types). Thus, the field `name` of `Student` in Listing 1 is represented by an AST object whose field `type` is `null` at the beginning of phase **resTypes**. During this phase, the compiler sets the `type` field to the AST object representing the prototype `String`. Phase **resTypes**, therefore, is included in the *semantic analysis* of the source code. The compiler goes through phase **resTypes** on a source file only after parsing all source files referenced in this file or loading the jar file with the referenced types.

Phases **afterResTypes** and **afterSemAn** only exist because of the MOP; they could be eliminated if Cyan did not support metaprogramming. Phase **afterResTypes** means *after resolving types*. Phase **afterSemAn** means *after semantic analysis*. In phase **semAn** (the remaining part of the semantic analysis), the compiler sets the `type` field of AST objects representing expressions and local variable declarations. In this phase, the compiler also does other checks, as demanded by the language. The last compilation phase is code generation. Currently, no metaobject method is called in this phase.

During the compilation of a prototype `P`, for each phase `X` in the set **{ parsing**, **afterResTypes**, **semAn**, **afterSemAn }**, the compiler runs the following algorithm:

```
for every metaobject metaObj associated with P {
  if the class/prototype of metaObj implements any interface
     associated with phase X
  then
    for each method 'met' of each interface associated with phase X {
      metaObj.met(...);   // call method 'met'
    }
  endif
}
```

The right-hand side of Figure 2 represents metaobjects associated with annotations of a hypothetical prototype `Proto` (not shown). According to the above algorithm, during phase **parsing**, the compiler calls the method `parsing_codeToAdd` of the metaobject associated with `lineNumber`. During phase **afterResTypes**, the compiler sends the message `afterResTypes_codeToAdd` to the metaobject `property`, and so on.

The Cyan compiler calls specific metaobject methods in each compilation phase, as exemplified in Figure 2. It also calls some metaobject methods when some events happen, such as “message passing”, “method missing”, “field access”, “field missing”, “inheritance”, and “method overriding”. That is, some interfaces are associated with *triggers*. For example, methods of interface

`IActionMethodMissing_semAn`

are called whenever the compiler is unable to find a method that matches a message passing. The methods of this interface can return an expression that replaces the message passing. Hence, this interface allows the implementation of *virtual* methods. In light of what was presented in this Subsection, the reader is invited to study Table 1. It shows the goals and the interfaces that the metaobject class should implement to achieve them. For example, to intercept “message passing”, the metaobject class should implement the method

```
IActionMessageSend_semAn
```

Note that the interface name ends with the compilation phase.

3.4. Important MOP Features

The following list complements the information on the MOP given in the previous Subsection.

(a) Metaobjects always generate code as strings. The code is added to a copy in memory of the prototype source code; the original file is not changed.

(b) The class or prototype of a metaobject may implement any number of interfaces for any number of phases. As a result, a metaobject can act in multiple compilation phases.

(c) In each phase, all metaobjects associated with a prototype have the same view of the code, which is how the code was in the last compilation phase. In phase **afterResTypes**, however, each of a prototype’s metaobjects knows what fields and methods the other metaobjects generated. This information is passed through the parameter `infoList` (see line 11 of Listing 7). The methods are called in rounds. In the first round, `infoList` is `null`. In the second round, `infoList` contains information on the code generated in the previous round. And so on until a maximum of 5 rounds. Each metaobject can then change the code it generates based on the code generated in the previous round. In phase **semAn**, no metaobject is aware of the code added by other metaobjects in that phase. A consequence of the above is that the order in which metaobject methods are called is not important. It would be relevant if a metaobject could view the code just added by other metaobjects in the same phase.

(d) Fields and methods can only be added in phase **afterResTypes**, and statements and expressions can only be added in phase **semAn**. Hence, metacode developers can be sure that, in phase **semAn**, the fields and methods of prototypes will not change.

(e) Base code can be changed by metaobjects in only one way: method statements and expressions can be replaced by others. Metaobjects cannot change inheritance (add or remove), method parameter types, implemented interfaces, and so on.

(f) A metaobject may add code in several phases. The added code may have annotations. However, the associated metaobjects will only be active in the next phase. Accordingly, in compilation phase X, a metaobject may generate code with annotations. But the metaobjects associated with these annotations will only be used in the compilation phases following X.

For example, suppose an annotation `@addField(0)` adds the following code to a prototype in phase **afterResTypes**.

```
@addField(1)
var Int field0;
```

The annotation `@addField(1)` should also add field `field1` to the prototype. However, this does not happen: the metaobject associated with this annotation will only be active in the next compilation phase, `semAn`, and in this phase, metaobjects cannot add fields to prototypes.

(g) The compiler keeps track of which code was inserted by which metaobject. If there is a compilation error, it points out exactly the annotation that inserted the code with errors. That is only possible because: (i) metaobject methods return code as strings; and (ii) metaobjects do not change the AST directly, it is the compiler that inserts the code.

(h) The Cyan compiler is the active part of the metaprogramming system. It calls the metaobject methods, which may return code to be inserted. However, it is the compiler that inserts the code.

(i) Metaobject methods only have access to a simplified and read-only version of the compiler AST. For example, the class `WrFieldDec` used in line 13 of Listing 7 is the read-only version of the AST class `FieldDec` of the compiler (that represents a field of a prototype). Internal compiler details of the latter are not revealed in the former.

(j) A metaobject method can add fields and methods to the prototype `P` in which the metaobject annotation is. In this case, the *natural prototype context* of the metaobject is `P`. A metaobject method whose annotation is in prototype `P` can be called when `P` is inherited by `Q`. In this case, the metaobject method will be called in the context of `Q`, which will be the *natural prototype context* of the metaobject (*natural context*, for short). The AST of `Q` will be visible to the metaobject method.

A prototype cannot view the fields (all are private) and private methods of other prototypes.[11] A metaobject method cannot view the private fields and methods of a prototype different than its *natural context* prototype. The visibility of a metaobject method is the same as its *natural context*.

We will explain that using the example of Subsection 3.1. Suppose the expression

```
((WrPrototype) field.getType()).getFieldList(compiler.getEnv())
```

is inside the `afterResTypes_codeToAdd` method of the `CyanMetaobjectGet` class of Listing 7. This expression tries to get private information about the type of the field, which is the list of fields of the type. There would be a compilation error in the example of Listing 6 because a metaobject of `Painting` would be trying to access private information of `String`, the type of the field `name`. However, there would be no error if the type of `name` were `Painting` because a prototype has the right to know its own private parts.

(k) When the compiler calls a metaobject method, it passes as an argument an object describing the compiler itself. The method parameter has names such as `compiler` (line 10 of Listing 7) or `env`. The available information varies according to the interface phase in which the metaobject method was declared. During **parsing**, metaobjects do not know method statements, other prototypes, or any information on code that comes textually after the annotation. In phase **resTypes**, metaobjects have access to AST objects that describe everything outside method statements. In phase **semAn**, metaobjects have access to all information except the types of variables and expressions that come after the annotation. Finally, all the information is available for metaobjects in phase **afterSemAn** (the code and the AST cannot be changed in this phase).

(l) Metaobjects cannot add code in phase **afterSemAn**. Therefore, the metaobjects that act in this phase view the final code, which cannot be changed any longer.

To build a metaobject in Java, the developer has to create a class, import the compiled version of the Cyan compiler (a `.jar` file), define the appropriate methods, compile it, and put the compiled version of the metaobject class in a special directory of a package. The annotation will be available for those source files that import that package. Although this process is not painful, there is an easy way that will be presented using the metaobject

```
    action_afterResTypes_semAn
```

of package `cyan.lang`. An example of its use is shown in Listing 8, which shows an annotation

[11] For a complete discussion of the visibility rules of Cyan, refer to the Cyan manual.

Listing 8: Annotation that takes Myan code

```
// inside a method
@action_afterResTypes_semAn{*
  func semAn_codeToAdd {
    runFile: #printData_at_semAn; // load and interpret this file
    return "one = 1;  ";
  }
  func afterResTypes_codeToAdd {
     // return a tuple
    return [.   "func getZero -> Int = 0; ",    "func getZero -> Int"   .];
  }
*}
```

that is inside the body of a method (assume this). The attached annotation code is in the language Myan (interpreted Cyan), but methods of interfaces associated with **afterResTypes** and **semAn** can be defined. There should be no parameters to the methods. The metaobject will insert the expected parameters into the scope automatically. For example, a parameter `compiler` can be used inside the first method, `semAn_codeToAdd`, because the equivalent method in the MOP library has a parameter with this name. In this example, the first method loads and executes a Myan file from disk and returns a string that will be the generated code. The second method inserts a method `getZero` into the current prototype. Metaobjects like these streamline the process of metaprogramming.

3.5. Other Annotation Kinds

Listing 2 shows examples of annotations starting with `@`, the most common kind. There are other annotations with different syntax or having other roles: annotations to types (next subsection), macros, literal numbers and strings, and Codegs.

A macro is associated with an identifier that starts the macro call (no `@`). When the compiler finds the identifier, it passes control to the associated metaobject, which then controls the compilation process. Therefore, after the initial identifier, any syntax may follow. Besides generating code, Cyan macros have all the power of other metaobjects. They have access to the prototype AST and can add fields and methods to the current prototype.

A literal number ending with an identifier such as `110bin` or `0FFF_Hex` is an annotation. The associated metaobject can replace the number with any expression and even perform checks in the current prototype.

Literal strings starting with an identifier are annotations, as `r"x[A-Z]*"` and `xml""" code in XML"""`. They generate an object for a regular expression and an object for XML code, respectively. The last example uses a multiline string that starts and ends with triple quotes. The metaobjects associated with `bin`, `Hex`, and `r` are defined in package `cyan.lang`, and therefore these annotations can be used without importing any other package.

Our final metaobjects are known as Codegs (Code + eggs). They depend on an IDE plugin to work. There are several flavors of syntax for Codegs: `@id`, literal numbers ending with identifiers, and literal strings starting with identifiers. At editing time in the IDE, two clicks on an annotation linked to a Codeg trigger the call to a method of the associated metaobject that usually shows a graphical user interface (GUI). This GUI then allows the developer to make choices that are used

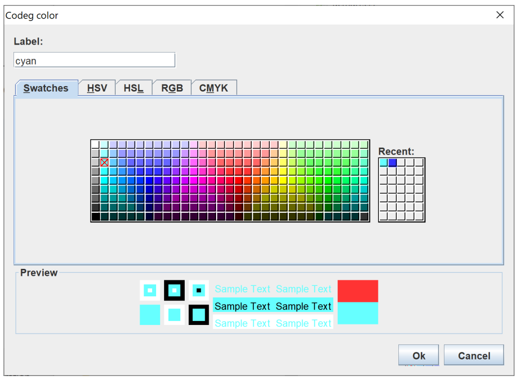

Figure 3: Codeg `color`

to generate data that is recorded in a file (this management is done by the IDE plugin). At compile time, the compiler passes to the metaobject the file data stored at editing time. The metaobject can use it to generate code and do checks. For example, suppose the following piece of code is in a file that imports Codeg `color`.

```
func defaultColor −> Int = @color(cyan);
```

At editing time, two mouse clicks on `color` are intercepted by the IDE plugin. Then, it calls a method of the metaobject `color` that shows the window of Figure 3. After a color is visually chosen, the developer presses the Ok button, and the window disappears. The plugin saves the color number in a file hidden from the developer. During compilation, the compiler reads the file and passes the file contents (just a number) to the metaobject, which returns this number as the code to be generated (no further processing is necessary in this case). Codegs cannot be replaced by IDE tools because they are metaobjects without any limitations. They can combine the information obtained at editing time with compiler data, such as the prototype AST.

3.6. *Pluggable-Type Systems*

Annotations can be attached to types for *additional* type checking. Hence, Cyan supports a *limited* pluggable-type system (Bracha, 2004). In the last two lines of this code, for example, metaobjects `range` and `secretValue` issue compile-time errors.

```
1   var Int@range(1, 12) month;
2   // 's' can be assigned only to other secretValue variable
3   var String@secretValue s = "a secret";
4   month = 12; // ok
5   month = 0;  // error
6   var String gossip = s;   // error
```

Metaobjects whose annotations can be attached to types, such as `range` and `secretValue`, should implement a specific interface of the MOP. This interface declares two methods. One of them is called when a value of the type is used on the left-hand side of an assignment, as `month` in line 4. The other method is called when a value of the type is used on the right-hand side, such as `s` in the last line.

4. Comparison with Related Work

This section presents some metaprogramming systems and how they are related to Cyan. The first subsection describes mechanisms for code generation and the benefits and drawbacks of each of them. Subsection 4.2 compares Cyan with runtime metaprogramming. Cyan support for pluggable-type systems is covered in Subsection 4.3. The comparisons related to internal and external DSLs are made in Subsections 4.4 and 4.5. The last subsection compares Cyan with languages and systems supporting compile-time metaprogramming.

4.1. How Code is Generated and Represented

Metaprograms generate code in many representations using several mechanisms (Smaragdakis et al., 2015), described next.

(a) *As text.* Code is generated in string format. There is no guarantee that the generated code is error-free.

(b) *Handling of the program Abstract Syntax Tree.* Code is generated by creating objects of the AST representing it, considering that the compiler is implemented in an object-oriented language.[12] Therefore, the developer has to know a great number of classes (it would be more than one hundred in Cyan). Code generation is difficult because it demands the mapping, by the metaprogrammer, of human-legible source code into the creation of AST objects. AST handling has the advantage that the metaprogram compiler usually catches all syntactic errors in the generated code. However, the AST objects created by metaprogramming may have semantic errors. If this happens, the compiler will not be able to identify the metacode that produced the AST objects that caused the errors.

(c) *Quoting* A special language syntax is used to transform text into AST objects. Therefore, the metaprogram handles text that is converted into AST objects. As a short example, in the Python-based language Converge (Tratt, 2008), the AST of `1 + 2` is obtained by the *quasi-quoted* expression `[| 1 + 2 |]`.

(d) *Specialized languages* Domain-Specific Languages are used to generate code. There are many languages that fit in this category: Genoupe (Draheim et al., 2005*b*) (Draheim et al., 2005*a*), SafeGen (Huang et al., 2005), and PTFJ (Miao and Siek, 2012), CTR (Fähndrich et al., 2006), MorphJ (Huang and Smaragdakis, 2011), MTJ (Reppy and Turon, 2007), and PTFJ (Miao and Siek, 2012). They share the common characteristic that the modifications they can make are limited. Either the code generation is inflexible (based on a pattern) or limited to certain tasks (as adding fields and methods to a class and only this). These languages are not further discussed because they are not comparable with the Cyan MOP, which is more general.

Cyan generates code as strings, and any errors in them are not caught either at compile-time or runtime of the metaprogram (metaobject code). However, errors are discovered in the compilation of the base program. The compiler will give precise error messages because it associates every code added to the source file with an annotation, which is the annotation associated with the metaobject that generated the code. The compiler will indicate precisely which annotation is linked to the code with errors.

[12]The reasoning does not change if the AST is implemented in a language that is not object-oriented.

4.2. Runtime Metaprogramming

The Smalltalk MOP (Goldberg and Robson, 1983) (Nierstrasz et al., 2009) is fundamentally different from that of Cyan because it cannot change the program. However, a Smalltalk program can change itself at runtime using methods inherited from fundamental classes such as `Behavior`, which are outside the MOP. There are methods that can, for example, add a new method to a class.

In Python 3 (Ramalho, 2015), metaclasses are used to change classes, including adding code to them. Each class has a single metaclass, a limitation that drastically reduces the complexity of metaprogramming in Python because there would be no interactions between metacode. Many of the problems with CTMP cited in Section 1 do not exist in Python. But the limit of just one metaclass per class severely damages the usability of metaclasses. A metaclass cannot intercept class inheritance or method overrides in a subclass. There are no compile-time guarantees in relation to metaclasses because classes are created only at runtime. Metaobjects in Cyan have access to the AST of the current prototype. In Python, the AST is not readily available. In Cyan, there are four compiler phases in which metaobjects may act. In Python, metaclasses act only when the class is created.

The prime example of a Metaobject Protocol is that of CLOS (Kiczales et al., 1991) (Kiczales et al., 1993) (Paepcke, 1993) (Bobrow et al., 1993) (DeMichiel and Gabriel, 1987), an extension of Common Lisp (Steele, 1990) with features for object-oriented programming. The CLOS MOP acts at runtime, allowing the intercepting of several operations: object creation, memory allocation, the calculus of superclass precedence,[13] method calls, field access, and many more. The MOP of this language uses *metaclasses*, which are classes of classes and methods.[14] Metaclasses are objects too. By using a user-made metaclass for a class, we change its expected behavior. For example, a metaclass can introduce a field into a class that keeps track of how many objects were created. The method that creates instances of the class may increment this field every time it is called. The Cyan MOP has a great number of the features of the CLOS MOP, such as the intercepting of method calls and the addition of code to classes/prototypes. However, there are many differences:

(a) metaprogramming occurs at runtime in CLOS and at compile time in Cyan. In Cyan, errors are detected earlier, and DSL code attached to annotations can be checked at compile-time;

(b) in CLOS, a class or other element has a single metaclass. In Cyan, any number of metaobjects can be attached to a declaration, and each annotation may have parameters and an attached DSL code (as `insertCode` of Listing 3), allowing for easy metaobject configuration;

(c) the AST is not available for the metaclass. This limits what metaclasses can do;

(d) Cyan has a lot of security measures that CLOS does not support. In Cyan, a metaobject has limited view privileges, which are usually the same as the view of the prototype associated with it.

4.3. Pluggable-Type Systems

In Cyan, annotations can be attached to types, as seen in Section 3.6.

var Int@range(1, 12) month = 0; // error

[13]The superclasses have to be ordered because the language supports multiple inheritance.
[14]CLOS has both *methods* and *generic methods*. For our purposes, it is not necessary to distinguish between them.

Type annotations have a subset of the power of the Checker Framework (Papi et al., 2008) because the metaobjects of the former have access to local code only. The Checker Framework is implemented as a Java annotation processor that can access the whole program. However, most annotations only need information about the prototype in which the annotation is used, which is supplied by the Cyan Metaobject Protocol. For example, the metaobject method that checks whether "`month = 0`" is correct has no access to the AST of any other prototype. Since most checks only need the AST of the current prototype, most Checker Framework annotations can be implemented in Cyan.

4.4. Internal DSLs

Internal DSLs (Fowler, 2010) are those built using a host general-purpose language. Regular language code is organized in such a way that it looks like the code of a DSL. The most common and old way of supporting this is through macros, as in CLOS (Steele, 1990) and Rust (Klabnik and Nichols, 2022). Languages like Ruby (Flanagan and Matsumoto, 2008), Groovy (König, 2007), and Scala (Odersky et al., 2021) offer other mechanisms for creating internal DSLs. These mechanisms include flexible syntax, anonymous functions, optional parentheses and punctuation, dynamic typing, extension methods (which simulate the addition of methods to classes at compile-time), metaprogramming (the insertion of methods into objects or classes), and many language-specific features. In Cyan, internal DSLs are implemented using *mainly* metaobject `grammarMethod`.

Metaobject `grammarMethod` intercepts the "method missing" error and tests if the message passing that caused it matches the regular expression given in the DSL code attached to an annotation.

```
@grammarMethod{*
  ((forward: Int | backward: Int | turn: Double)+
   (ligthOn: | beep:) )
*}
func robot: T t { ... }
```

For example, suppose a prototype `Robot` declares a single method, the one shown above. Variable `myRobot` refers to an object of this prototype.

```
myRobot forward: 5 turn: 30.0 backward: 10 turn: 60.0
        forward: 25 beep:;
```

This message passing causes a "method missing" error, which is intercepted by the metaobject `grammarMethod` which packs the message into an object of type `T`. This object is passed as an argument in a call to the method "`robot:`". Type `T` is a complex type composed of arrays, tuples, and unions. It is not explained here. Metaobject `grammarMethod` can be used for implementing internal DSLs that use keywords and regular grammars.

Metaobject `grammarMethod` can simulate *argument labels* of language Swift or named parameters of Groovy (Subramaniam, 2013) and Kotlin.[15] This feature allows the caller to use labels associated with parameter names to distinguish the arguments. The order is not important. Both this and optional arguments can be easily implemented using grammar methods.

[15]https://kotlinlang.org

It is interesting to note that metaobjects that intercept the "method missing" error can simulate the implementation of *implicit arguments* of language Scala (Odersky et al., 2021). Implicit arguments in Scala are arguments passed to a method without being explicitly stated. In Cyan, a metaobject that intercepts the "method missing" error can replace the message passing by any expression. In particular, the metaobject can replace it with another message passing with more arguments. Like Scala implicit arguments, one of the added arguments can be taken from the original message-passing environment. This is possible because the metaobject has access to the full AST of the prototype of the original message passing.

4.5. DSL Code Embedded Within Base Code

Cyan supports external DSLs embedded in its code. The DSL code is put either between `{*` and `*}` attached to the annotation (see Listing 3 and the grammar method examples) or inside a string preceded by an identifier (as in `xml"..."`). The metaobject associated with the annotation is responsible for parsing and analyzing the DSL code at compile time. Converge (Tratt, 2008), a Python-based language, supports a similar mechanism:

```
$<<compilerFunc>>
    dslCode
```

Function `compilerFunc` is called at compile time with `dslCode` as an argument (a string) and returns an AST object that replaces the *DSL block* (this whole example). Unlike metaobjects, this function cannot add fields, methods, or classes to the program. This Converge language mechanism is the closest we have found in the literature to Cyan annotations with attached DSL code.

4.6. Compile-Time Metaprogramming

This subsection is divided into parts. First, the text presents some compile-time metaprogramming systems. They are then compared with Cyan.

Aspects

AspectJ[16] (Kiczales et al., 2001) is a Java extension for *Aspect-Oriented Programming* (AOP) (Kiczales et al., 1997). In this paradigm, code for an *aspect* of a program, like error handling and logging, is grouped and put in just one place instead of being scattered in the program. In AspectJ, several operations can be intercepted, like method calls, field access, and the creation of objects. This is specified through an *aspect language*, a DSL, resembling Java. The AspectJ compiler, directed by user code, can add methods, fields, and constructors to classes and change inheritance and implemented interfaces.

AspectJ can change the inheritance hierarchy, which includes the replacement of the superclass and the addition of implemented interfaces. By design, this is not allowed in Cyan because big changes like this damage the readability of the base code. The DSL used by AspectJ to select the *joint points* (methods and classes that will be changed) is static, as is the code to be added. Hence, before adding a method to a class, AspectJ cannot examine the class AST to tailor the addition of the method. Instead of changing a specific class or method through an annotation, as Cyan does, AspectJ is able to change all classes and methods that match the pattern of *pointcuts*.

[16]https://www.eclipse.org/aspectj/doc/next/progguide/language.html

The Cyan MOP and AspectJ have different goals and, therefore, very different functionalities. In particular, most of the existing Cyan metaobjects[17] cannot be implemented in AspectJ. For example, the metaobject `fsmDSLMethods` adds code before the prototype methods to enforce that they are called in an order specified by a finite state machine (FSM). For example, method `open` should be called before `write`, and this should be called before `close`. The FSM is given in the *attached text* to the annotation. Although AspectJ can add code to classes, it does not support annotations attached to classes with a DSL code for directing the code generation.

Metaprogramming Systems with a MOP

OpenC++ (Chiba, 1995) is a C++ extension in which metaclasses for classes and methods are given the opportunity of changing the AST after parsing. A metaclass for a class `C` may intercept method calls whose receivers have type `C`. The method call may, after the interception, be changed or replaced. The MOP of OpenC++ also allows interception of variable declarations, creation of objects, and reading and writing in fields. OJ (Tatsubori et al., 2000) (Tatsubori, 1999) is a Java extension in which a class may be associated with a user-defined metaclass. Methods of the metaclass have the opportunity of changing the AST. For example, a method called `translateDefinition` of a metaclass may add methods to the class. `expandFieldRead` can change the read of a class field. The user-defined metaclass can also define methods for intercepting object creation, array allocation, writing to fields, method calls, and casts to the class.

Metaprogramming Systems without a MOP

Languages Xtend[18], Groovy (König, 2007), and Nemerle[19] (Skalski, 2005) support *compile-time metaprogramming* without a Metaobject Protocol. These languages support *metaprogramming features*. They share many similar characteristics, as described below, and therefore will be considered together.

(a) Annotations are attached to classes, methods, and other declarations;

(b) An annotation is linked to a *Processor Class* (PC) that can implement interfaces and define methods that change the compilation;

(c) Methods of the PC are invoked in several phases of the compilation, like before parsing, after parsing, before typing members (similar to **afterResTypes** of the Cyan compiler), after semantic analysis, during code generation, etc.;

(d) Methods of the *Processor Class* have parameters that represent language elements that can be changed at compile time. For example, the AST object of the annotated class or method is passed as an argument. Methods of the PC can, using these AST objects, add methods to an annotated class, change inheritance, add statements to an annotated method, change method statements, and so on. Any AST object reachable from the method arguments can be changed. Therefore, a method can be added to a class that is not annotated or is not directly related to the annotated class. The class may be, for example, just the type of a method parameter accessible to the PC method;

[17]`www.cyan-lang.org`
[18]https://www.eclipse.org/xtend
[19]http://nemerle.org

(e) A method of the *Processor Class* that overrides an interface method is used in the compilation phase associated with that interface (much like Cyan).

Compiler Plugins

A *compiler plugin* is composed of metacode that interacts at low-level with the compiler, changing the compilation process. The difference between the terms *compiler plugins* and *metaprogramming systems without a MOP* is that the former emphasizes implementation aspects (low-level details), whereas the latter emphasizes conceptual aspects (high-level specifications). *Compiler plugins* are supported by the languages Scala[20], Java[21], X10 (Nystrom and Saraswat, 2007), Kotlin,[22] TypeScript[23], and Rust[24]. Java annotation processors (Darcy, 2006) are compiler plugins for Java that allow checks but not code modifications. They are used, for example, for implementing pluggable types (Bracha, 2004). Project Lombok (Kimberlin, 2010) is a Java annotation processor whose supported annotations can add code to classes because it uses non-supported downcasts. Compiler plugins will not be discussed in depth in this paper because there is a shortage of good documentation about them. However, languages whose compilers accept plugins have all the main characteristics of languages supporting *compile-time metaprogramming* without a Metaobject Protocol, as discussed above.

Comparison with the Cyan MOP

The Cyan MOP was built on the metaprogramming system of modern languages, such as Groovy, but with some characteristics of MOPs. From Metaobject Protocols, Cyan took the interception of message passings, inheritance, field access, and errors such as method or field missing. From the other systems, it took all the rest.

The following comparison will be guided by the list of problems with metaprogramming given in the introduction. The problem letter precedes each discussion.

(a) *A metacode associated with a source file can change the base code of another file. Hence, to know the final version of a file, the developer has to know the behavior of every metacode in the metaprogram.*

Languages OJ, Xtend, Groovy, and Nemerle allow non-local changes through AST handling. Hence, a metacode of one class can change another class. In particular, metacode associated with annotations of a source file can change another source file, which is called *obliviousness* (Filman and Friedman, 2000). Generally, any metaprogramming system that supplies the AST to metacode has this problem (Clifton and Leavens, 2003). CLOS, OpenC++, and BSJ limit the changes to the scope of the metaclass or metacode. In Cyan, metaobjects have access to a read-only AST and it is the compiler that changes the base code. The compiler only allows changes in the current prototype or the prototype that the metaobject was intended to change, which is its *natural context prototype*, as explained in item j of Subsection 3.4.

(b) *A class may not have access to the private fields and methods of another class. But the metacode can because it has access to the compiler data structures.*

[20]https://docs.scala-lang.org/scala3/reference/changed-features/compiler-plugins.html

[21]https://docs.oracle.com/en/java/javase/11/docs/api/jdk.compiler/com/sun/source/util/Plugin.html

[22]https://kotlinlang.org/docs/all-open-plugin.html

[23]https://github.com/microsoft/TypeScript/wiki/Using-the-Compiler-API

[24]Compiler plugins are an unstable feature of the language. See https://doc.rust-lang.org/beta/unstable-book/language-features/plugin.html

As explained in item j of Subsection 3.4, there is a compilation error when a metaobject attempts to access private data of a prototype that is not its *natural context prototype*.

(c) *The developer has to know many details about the compiler since the metacode interacts with low-level compiler algorithms and data structures. Therefore, metaprogramming is not only harder, but the metacode can bypass compiler checks, invalidate invariants, damage data structures, and crash the compiler.*

In Cyan, metaobjects have access to a read-only and simplified AST of the base code. In most metaprogramming systems, the metacode has direct access to the compiler's AST. Thus, metacode can easily damage the AST. If the compiler discovers the damage, it will not be able to point out the guilty metacode. The metacode may break compiler invariants after the compiler does the final checks, leading it to generate incorrect code or crash. That is impossible in Cyan. Besides that, Cyan metaobjects generate code as strings. Hence, metaobject developers do not need to have deep knowledge of the AST or the compiler.

(d) *Metacode can insert new code into the base code. If the new code has errors, the compiler is unable to point out which metacode (or associated annotation) is responsible for adding the new code. This is because the metacode itself changes the compiler AST to add new AST objects. No trace is left of who did what.*

This problem occurs in all metaprogramming systems that allow metacode to handle the AST. In Cyan, metaobject methods return code as strings that are inserted in the base code by the compiler. The compiler associates the inserted code with the annotation that asked for its insertion. If there is an error in the code produced by the metaobject, the compiler will point out exactly which annotation is responsible for it.

The Converge programming language (Tratt, 2005) (Tratt, 2008) supports compile-time metaprogramming. The compiler tracks who produced which code to issue precise error messages. It goes beyond Cyan in two aspects: (a) every bytecode[25] knows its origin, which can be used in runtime error messages, and (b) an AST node can be associated with more than one location (an error may be associated with more than one source).

(e) *The order in which the metacode is called is not specified by the language. It can even change between compilations (with the same base code). By changing the metacode calling order, their view of the program can change if a metacode can view the code added by a metacode executed previously.*

To our knowledge, the only metaprogramming languages that guarantee the execution order of metacode are BSJ (Palmer and Smith, 2011) and Cyan (as explained in item c of Subsection 3.4). The order is important if there are two or more metacode associated with the same class, prototype, or source file. What one metacode does may impact the following metacode the compiler calls.

(f) *A check made by one metacode may be invalidated by code added by another metacode.*

In Cyan, metaobjects should do checks in phase `afterSemAn`, in which the code cannot be changed anymore. In languages that allow irrestrict changes in the AST, checks made by a metacode may be invalidated by code added later by another metacode.

(g) *Metacode may generate metacode that may generate metacode ad infinitum.*

This problem usually does not occur in metaprogramming languages because the base code is processed just once for each metacode. In Cyan, metaobjects may generate code with annotations, but these will only be used in the next compilation phase. As the number of phases is finite, so will be the compilation.

[25]The source code is translated into bytecodes of a Converge VM.

(h) *Metacode may not terminate its computation. The compiler will not terminate either.*

In Cyan, there is a time limit for each metaobject method to terminate its execution, which can be changed by a compiler option. To our knowledge, no other metaprogramming language with or without a MOP employs this solution.

(i) *Metacode may generate code that uses identifiers already in use. This is the hygiene problem of macros.*

This problem does occur in Cyan and, to our knowledge, in all metaprogramming systems with or without a MOP. In Cyan, it can be easily avoided. There is a MOP method that returns an identifier name that is guaranteed not to crash with any other identifier in the program.

4.7. Comparison of Features

The Cyan MOP has some features that are not found in other metaprogramming systems without a MOP, generally. These are the systems directly related to Cyan, both in terms of implementation and functionality. These features are described next. It is not difficult to add them to any language.

In Cyan, a source file that imports a package has access to all metaobjects defined in that package. In other languages, the path (directory and file name) of the metacode should be specified through some configuration file or through the command line.

In Cyan, annotations can be expressions such as `eval` presented in Subsection 2.2 or
`@currentmethodname("currentmethodname")`
that return the current method name as a string. We are unaware of any other metaprogramming language that allows this.

Codegs, presented in Subsection 3.5, collect at editing time information that may be used at compile time. Codegs play the role of plugins to the IDE but, at the same time, have the full power of metaobjects. They are not directly related to any metaprogramming system cited in this article.

Cyan has a project file that specifies which packages compose the program. Metaobjects can be used in the project file to specify compiler options, create and associate variables with values (that can be used by metaobjects), and apply an annotation to a bunch of prototypes. For example, instead of attaching metaobject `checkStyle` to all prototypes of the package `cyanPack`, we can just attach the annotation to the package in the project file.

Metaobjects can transfer information from a prototype `P` to a generic prototype that takes `P` as an argument. As an example, a metaobject associated with prototype `Int` supplies the code of a method `sum` that is added to `Array<Int>` by a metaobject associated with this last prototype.

In phases **afterResTypes**, **semAn**, and **afterSemAn**, metaobjects of the same prototype can communicate with each other through the exchange of data. This may be used for coordinating their actions.

Some metaobjects take interpreted Cyan (Myan) code as the attached DSL text. Using this feature, metacode can be specified inside regular Cyan code. As examples, see Listings 3 and 4 of Subsection 2.2 and Listing 8 of Subsection 3.4. The attached text of these annotations are metacode in Myan, which can be put in standard directories of a package and loaded at compile time as shown in line 4 of Listing 8.

When there is a compilation error in a source file, the compiler produces a file with the final source code, which includes the code added by metaobjects. The developer can view the code generated by metaobjects and discover possible errors in them.

5. Conclusion

No sufficiently powerful metaprogramming system is easy to use, which includes the Cyan MOP. However, we consider that this MOP is not too difficult to use in relation to its power because of the following reasons. The goals direct the implementation of metaobject classes (which interfaces to implement; see Table 1 and Subsection 3.1). Therefore, the metaprogrammer, guided by the goals, makes the most important decisions *before* starting to code. In each compilation phase, metaobject methods *ask* the compiler to add code. As a result, the metaprogram acts passively in relation to the compiler, who is in control of the execution flow of the metaprogram. The developer has access to simplified read-only compiler data structures (which include the AST of the base code), the developer does not need to know the compiler internal details, there cannot be a compiler crash caused by metacode. When there is a compilation error in a source file, the compiler points out exactly the annotation that caused the error and produces a new file with the code added by metaobjects. There are many security mechanisms that are not present in other systems.

Metaobjects have been extensively used in the Cyan libraries. There are many general metaobjects, such as `property`, `init`, `immutable`, `eval`, and `doc`, but also those for specific prototypes and methods. Among the latter, there are metaobjects for generating code, such as `createTuple` and `createFunction` (they create all methods of prototypes `Tuple<T>` and `Function<T+>`). And there are ones for doing checks based on the semantics of methods, such as `checkIsA` (check if an argument is a prototype), `checkMethodEqualEqual` (check if the result of "`e1 == e2`" is known at compile time and, therefore, the comparison is unnecessary), and `overrideToo` (demand that, if == is overridden in a subprototype, method `hashCode` is too). The use of metaobjects in the Cyan libraries is a good justification of the main goals of the Cyan MOP, which are to build general metacode and to make libraries easier to build (because of code generation) and safer to use (because of compile-time checks).

We have plans to further improve the Cyan MOP. One of the planned features is to support variable ownership, like in language Rust (Klabnik and Nichols, 2022). The Cyan compiler is available for download at `cyan-lang.org`. There one can find the language manual, a complete description of the Cyan MOP, and a list of around one hundred metaobjects with examples.

Acknowledgments. This project was partially financed by FAPESP (São Paulo - Brazil) under Process number 2014/01817-3.

References

Belyakova, J. and Mikhalkovich, S. (2015), 'Pitfalls of c# generics and their solution using concepts', *Proceedings of ISP RAS* **27**(3), 29–46.
URL: *http://mi.mathnet.ru/tisp134*

Biboudis, A., Inostroza, P. and Storm, T. v. d. (2016), Recaf: Java dialects as libraries, *in* 'Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences', GPCE 2016, Association for Computing Machinery, New York, NY, USA, p. 2–13.

Bobrow, D. G., Gabriel, R. P. and White, J. L. (1993), Object-oriented programming, *in* A. Paepcke, ed., 'Object-oriented Programming', MIT Press, Cambridge, MA, USA, chapter CLOS in Context: The Shape of the Design Space, pp. 29–61.

Bracha, G. (2004), Pluggable type systems, *in* 'In OOPSLA'04 Workshop on Revival of Dynamic Languages'.

Burmako, E. (2013), Scala macros: Let our powers combine! on how rich syntax and static types work with metaprogramming, *in* 'Proceedings of the 4th Workshop on Scala', SCALA '13, Association for Computing Machinery, New York, NY, USA.

Chiba, S. (1995), A metaobject protocol for c++, *in* 'Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications', OOPSLA '95, ACM, New York, NY, USA, pp. 285–299.

Clifton, C. and Leavens, G. T. (2003), Obliviousness, modular reasoning, and the behavioral subtyping analogy, Technical Report 329, Computer Science Technical Reports - Iowa State University.

Csh (2023), 'C# language specification'.
URL: *https://learn.microsoft.com/en-us/dotnet/csharp/*

Darcy, J. (2006), 'Java specification request 269: Pluggable annotation processing api'.
URL: *http://jcp.org/en/jsr/detail?id=269*

DeMichiel, L. G. and Gabriel, R. P. (1987), The common lisp object system: An overview, *in* 'European Conference on Object-oriented Programming on ECOOP '87', Springer-Verlag, Berlin, Heidelberg, pp. 151–170.

Draheim, D., Lutteroth, C. and Weber, G. (2005*a*), 'Generative programming for c#', *ACM SIGPLAN Notices* **4**0(8), 29–33.

Draheim, D., Lutteroth, C. and Weber, G. (2005*b*), A type system for reflective program generators, *in* 'Proceedings of the 4th International Conference on Generative Programming and Component Engineering', GPCE'05, Springer-Verlag, Berlin, Heidelberg, p. 327–341.

Fähndrich, M., Carbin, M. and Larus, J. R. (2006), Reflective program generation with patterns, *in* 'Proceedings of the 5th International Conference on Generative Programming and Component Engineering', GPCE '06, Association for Computing Machinery, New York, NY, USA, p. 275–284.

Filman, R. E. and Friedman, D. P. (2000), Aspect-oriented programming is quantification and obliviousness, *in* 'OOPSLA 2000 Workshop on Advanced Separation of Concerns', Minneapolis, MN.

Flanagan, D. and Matsumoto, Y. (2008), *The Ruby Programming Language*, first edn, O'Reilly.

Fowler, M. (2010), *Domain Specific Languages*, 1st edn, Addison-Wesley Professional.

Goldberg, A. and Robson, D. (1983), *Smalltalk-80: the language and its implementation*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Gosling, J., Joy, B., Steele, G. L., Bracha, G. and Buckley, A. (2014), *The Java Language Specification, Java SE 8 Edition*, 1st edn, Addison-Wesley Professional.

Guimarães, J. d. O. (2020), The cyan language.
URL: *http://cyan-lang.org/articles/*

Guimarães, J. d. O. (2022), The cyan language metaobject protocol.
URL: *http://cyan-lang.org/docs*

Huang, S. S. and Smaragdakis, Y. (2011), 'Morphing: Structurally shaping a class by reflecting on others', *ACM Trans. Program. Lang. Syst.* **3**3(2).

Huang, S. S., Zook, D. and Smaragdakis, Y. (2005), Statically safe program generation with safegen, *in* 'Proceedings of the 4th International Conference on Generative Programming and Component Engineering', GPCE'05, Springer-Verlag, Berlin, Heidelberg, p. 309–326.

Kamin, S., Clausen, L. and Jarvis, A. (2003), Jumbo: Run-time code generation for java and its applications, *in* 'Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization', CGO '03, IEEE Computer Society, USA, p. 48–56.

Kiczales, G., Ashley, J., Rodriguez, L., Vahdat, A. and Bobrow, D. G. (1993), *Object-oriented Programming: The CLOS Perspective*, MIT Press, Cambridge, MA, USA, chapter Metaobject protocols: Why we want them and what else they can do, pp. 101–118.

Kiczales, G., des Rivières, J. and Bobrow, D. G. (1991), *The Art of Metaobject Protocol*, MIT Press, Cambridge, MA, USA.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. (2001), An overview of aspectj, *in* J. L. Knudsen, ed., 'ECOOP', Vol. 2072 of *Lecture Notes in Computer Science*, Springer, pp. 327–353.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. (1997), Aspect-oriented programming, *in* M. Akşit and S. Matsuoka, eds, 'ECOOP'97 — Object-Oriented Programming', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 220–242.

Kimberlin, M. (2010), 'Reducing boilerplate code with project lombok'.
URL: *http://jnb.ociweb. com/jnb/jnbJan2010.html*

Klabnik, S. and Nichols, C. (2022), *The Rust Programming Language*, second edn, No Starch Press.
URL: *https://doc.rust-lang.org/book*

Kohlbecker, E., Friedman, D. P., Felleisen, M. and Duba, B. (1986), Hygienic macro expansion, *in* 'Proceedings of the 1986 ACM Conference on LISP and Functional Programming', LFP '86, ACM, New York, NY, USA, pp. 151–161.

König, D. (2007), *Groovy in Action*, Manning, New York.

Lilis, Y. and Savidis, A. (2019), 'A survey of metaprogramming languages', *ACM Comput. Surv.* **5**2(6).

Liskov, B., Snyder, A., Atkinson, R. and Schaffert, C. (1977), 'Abstraction mechanisms in CLU', *Communications of the ACM* **2**0(8), 564–576.

Miao, W. and Siek, J. (2012), Pattern-based traits, *in* 'Proceedings of the 27th Annual ACM Symposium on Applied Computing', SAC '12, Association for Computing Machinery, New York, NY, USA, p. 1729–1736.

Nierstrasz, O., Ducasse, S. and Pollet, D. (2009), *Squeak by Example*, Square Bracket Associates.

Nystrom, N. and Saraswat, V. (2007), An annotation and compiler plugin system for x10, Technical report, Technical Report RC24198, IBM TJ Watson Research Center.

Odersky, M., Spoon, L., Venners, B. and Sommers, F. (2021), *Programming in Scala, Fifth Edition*, Artima Incorporated.

Paepcke, A. (1993), Object-oriented programming, *in* A. Paepcke, ed., 'Object-oriented Programming', MIT Press, Cambridge, MA, USA, chapter User-level Language Crafting: Introducing the CLOS Metaobject Protocol, pp. 65–99.

Palmer, Z. and Smith, S. F. (2011), 'Backstage java: Making a difference in metaprogramming', *SIGPLAN Not.* **4**6(10), 939–958.

Papi, M. M., Ali, M., Correa, Jr., T. L., Perkins, J. H. and Ernst, M. D. (2008), Practical pluggable types for java, *in* 'Proceedings of the 2008 International Symposium on Software Testing and Analysis', ISSTA '08, ACM, New York, NY, USA, pp. 201–212.

Parr, T. (2013), *The Definitive ANTLR 4 Reference*, 2nd edn, Pragmatic Bookshelf.

Pelenitsyn, A. (2015), 'Associated types and constraint propagation for generic programming in scala', *Program. Comput. Softw.* **4**1(4), 224–230.
URL: *https://doi.org/10.1134/S0361768815040064*

Ramalho, L. (2015), *Fluent Python: Clear, Concise, and Effective Programming*, O'Reilly Media.

Redmond, B. and Cahill, V. (2002), Supporting unanticipated dynamic adaptation of application behaviour, *in* 'Proceedings of the 16th European Conference on Object-Oriented Programming', ECOOP '02, Springer-Verlag, Berlin, Heidelberg, pp. 205–230.

Reppy, J. and Turon, A. (2007), Metaprogramming with traits, *in* 'Proceedings of the 21st European Conference on Object-Oriented Programming', ECOOP'07, Springer-Verlag, Berlin, Heidelberg, p. 373–398.

Rompf, T., Amin, N., Moors, A., Haller, P. and Odersky, M. (2012), 'Scala-virtualized: Linguistic reuse for deep embeddings', *Higher Order Symbol. Comput.* **2**5(1), 165–207.

Siek, J. G. and Lumsdaine, A. (2011), 'A language for generic programming in the large', *Sci. Comput. Program.* **7**6(5), 423–465.

Skalski, K. (2005), Syntax-extending and type-reflecting macros in an object-oriented language, Master's thesis, University of Wroclaw, Poland. Nemerle.

Smaragdakis, Y., Biboudis, A. and Fourtounis, G. (2015), Structured program generation techniques, *in* 'Grand Timely Topics in Software Engineering - International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures', Vol. 10223 of *Lecture Notes in Computer Science*, Springer, pp. 154–178.

Steele, G. L. (1990), *Common LISP: The Language (2nd Ed.)*, Digital Press, USA.

Stroustrup, B. (2003), Concept checking - a more abstract complement to type checking, Technical Report N1510=03-0093, C++ Standards Committee Papers. ISO/IEC JTC1/SC22/WG21.
URL: *http://www.stroustrup.com/n1510-concept-checking.pdf*

Stroustrup, B. (2013), *The C++ Programming Language*, 4th edn, Addison-Wesley Professional.

Subramaniam, V. (2013), *Programming Groovy 2: Dynamic Productivity for the Java Developer*, number v. 2 *in* 'Pragmatic Bookshelf', Pragmatic Bookshelf.
URL: *https://books.google.com.br/books?id=jvbTmAEACAAJ*

Taha, W. (2007), A gentle introduction to multi-stage programming, part II, *in* R. Lämmel, J. Visser and J. Saraiva, eds, 'Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Revised Papers', Vol. 5235 of *Lecture Notes in Computer Science*, Springer, pp. 260–290.
URL: *https://doi.org/10.1007/978-3-540-88643-3_6*

Tatsubori, M. (1999), An Extension Mechanism for the Java Language, Master's thesis, University of Tsukuba, Japan.

Tatsubori, M., Chiba, S., Itano, K. and Killijian, M.-O. (2000), Openjava: A class-based macro system for java, *in* 'Proceedings of the 1st OOPSLA Workshop on Reflection and Software Engineering: Reflection and Software Engineering, Papers from OORaSE 1999', Springer-Verlag, London, UK, UK, pp. 117–133.

Tratt, L. (2005), The Converge programming language, Technical Report TR-05-01, Department of Computer Science, King's College London.

Tratt, L. (2008), 'Domain specific language implementation via compile-time meta-programming', *ACM Trans. Program. Lang. Syst.* **3**0(6), 31:1–31:40.

Wadler, P. and Blott, S. (1989), How to make ad-hoc polymorphism less ad hoc, POPL '89, Association for Computing Machinery, New York, NY, USA, p. 60–76.
URL: *https://doi.org/10.1145/75277.75283*

Wehr, S. and Thiemann, P. (2011), 'Javagi: The interaction of type classes with interfaces and inheritance', *ACM Trans. Program. Lang. Syst.* **3**3(4), 12:1–12:83.

Zhang, Y., Loring, M. C., Salvaneschi, G., Liskov, B. and Myers, A. C. (2015), 'Lightweight, flexible object-oriented generics', *SIGPLAN Not.* **5**0(6), 436–445.
URL: *http://doi.acm.org/10.1145/2813885.2738008*

Table 1: Table goal-“which interface to implement” for a metaobject `myAnnot` whose annotation is attached to prototype `P` or is inside `P`

Goal	Metaobject interface
Create a new prototype in the package of `P`	IActionNewPrototypes_parsing IActionNewPrototypes_afterResTypes IActionNewPrototypes_semAn
Associate information with a declaration that can be later retrieved by other metaobjects	ICompilerInfo_Parsing
Add new fields and methods to `P`, add code before the first statement of a method of `P`, or check the signatures of fields and methods of `P`	IAction_afterResTypes
Add code after the annotation `myAnnot`, know the fields and methods of `P`, know the AST of statements before the annotation, or replace a statement of `P` by another statement	IAction_semAn
Add code after the annotation, which should be attached to a local variable declaration	IActionVariableDeclaration_semAn
Replace a message passing by any statement. Annotation `myAnnot` should be attached to a method that can be called by the message passing	IActionMessageSend_semAn
Replace a message passing for which there is no associated method. `myAnnot` should be attached to `P` or one of its methods. The type of the receiver of the message passing should be in the hierarchy of `P`	IActionMethodMissing_semAn
Replace assignments to a field or the *get* of a field. `myAnnot` should be attached to a field	IActionFieldAccess_semAn
Replace the *get* of a non-existing field by an expression; replace the assignment to a non-existing field by an expression. `myAnnot` should be attached to `P`	IActionFieldMissing_semAn
Implement pluggable-type systems	IActionAttachedType_semAn
Check the subprototypes of `P`. The metaobject methods are executed in the context of the subprototype and can check anything: if the subprototype defines some methods, if a method has some statements, etc	ICheckSubprototype_afterSemAn
Check overridden subprototype methods. If the annotation `myAnnot` is attached to method `m` of `P`, the metaobject can check methods of subprototypes that override `m`. The metaobject has access to the AST of the subprototype method	ICheckOverride_afterSemAn
Check a declaration, which may be a prototype, method, or field. The metaobject has access to the whole AST of the declaration	ICheckDeclaration_afterSemAn
Check every message passing that may call the method attached to `myAnnot`	ICheckMessageSend_afterSemAn