The Cyan Language Metaobject Protocol

Abstract

Compile-time metaprogramming can tailor a language by changing the way the compiler generates code. Its support by current languages is either not powerful enough or demands the learning of a great number of details. Metaprogramming in Cyan is made through a compile-time Metaobject Protocol (MOP) that is both powerful, with many possibilities of altering the compiler behavior, and relatively simple for the power it offers. The Cyan MOP achieves that through a carefully designed set of classes and interfaces and at the expenses of time of compilation.

1998 ACM Subject Classification D.3.3 Language Constructs and Features

Keywords and phrases Object-oriented languages, computational reflection, metaprogramming, compiler, meta-object protocols

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

Before a software project can begin the designer has to select her tools among the wealth of options provided by modern computer science: database systems, user interface, networking, programming languages, and so on. The choice of programming language to be used in a project was always considered important. An adequate language for each job can reduce the amount of effort substantially. However, till a few decades ago corporations do not consider worthwhile to develop their own programming languages. This has changed.

Today there is a great availability of good Integrated Development Environments (IDE) and compiler tools such as compiler generators, backends, and more such as LLVM [19]. Not to say the virtual machine of Java [18] (JVM) and the intermediary language and JIT compiler of .NET [4], which frees the compiler constructor from using a complex real machine and supply the runtime system needed to run programs. These facts make it easier than ever to design and implement a language.

Because of the cited reasons, corporations then began to design and implement languages to use in their projects. It becomes worthwhile to do that. The gain with the new language offsets the loss with the design and construction of the compiler and libraries. If the new language uses CLR or JVM, it *may* gets all the libraries made in other languages such as C# or Java for free. The new languages that were created in the last decade or so include Xtend [33], Go [8], Swift [14], Rust [25], and Hack [13]. There are many more that will not be cited.

There are many reasons for the creation of a new language; each of them corresponds to a need that is not supplied completely by existing languages. These needs are related to static typing for decreasing runtime errors, faster runtime execution to meet user expectations, legibility of code, checkings beyond those made by the usual type systems, adequacy of the language to the problem domain, and greater productivity of programmers. Programmer's productivity may be increased with a simpler, cleaner, and more high-level language. A language that has a higher level than other allows the programmer to write less code for the same job than the other language.

This paper focus in the creation of new language features addressing the adequacy of the language to the problem domain, checkings beyond those made by usual type systems, and generation of boilerplate code (which makes the language more high level). A language may fit a problem domain better if it allows the creation of internal Domain Specific Languages

(DSL). An internal DSL is a DSL whose code appears inside the code written in the general language. Parts of the code may be programmed in specific languages thus decreasing the distance between the code and the domain. In its extreme, one can use language-oriented programming [32] to divide all the code into several languages and DSLs.

Compilers do the type checks demanded by the language. The user cannot usually ask for an additional check in its class or method call. This has changed with annotations such as that for Java [23], C# [7] (Data annotations), Groovy [17] and Xtend [33].

Boilerplate code is code that is repetitive or is included in several places with few or no changes. A prime example is getting and set methods for instance variables (fields) of classes. The format of these methods is almost always the same.

We have found a compelling partial solution for the needs of making internal DSLs, doing additional checking in the code, and generating boilerplate code automatically. Our solution does so using metaprogramming in a new language called Cyan [1]. This paper describes some of the metaprogramming facilities of this language. We start in Section 2 by showing the design goals of the language in respect to metaprogramming. The small subset of Cyan needed to understand this paper is in Section 3. Section 4 presents the Metaobject Protocol of the language. Finally, in Section 5 we compare the Cyan MOP with the MOP of other languages currently in use.

2 Design Goals

Metaprogramming relates to programs that generate or do any kind of handling of other programs, including themselves. This concept has been used with many meanings [6]. It may be applied to:

1. programs that take as input programs such as compilers and pre-processors. A special case is when a program is generated from a specification such as in compiler generators and lexical analyzer generators;
2. generic classes that take parameters. From a generic class, a new fresh class may be created if the parameters are supplied at compile-time. This occurs, for example, in template classes of C++ [27]. Metaprogramming does not happen in languages that do not duplicate the class code such as Java;
3. runtime reflection, which may be behavioral or introspective. Introspective reflection happens when a program examines itself at runtime. It may know its classes and functions, the methods of each class, the parameters of each method and so on. Some form of introspective reflection is present in most of the main object-oriented languages. A language that supports behavioral reflection allows the program to change itself at runtime. The prime examples are Smalltalk [9] and its prototype-based cousin Self [30]. In these languages, there is total liberty for the program to change everything at runtime: add and remove methods and instance variables of a class, create new classes, add code everywhere, and so on;
4. partial evaluation when part of code that would be executed at runtime is executed at compile-time by the compiler;
5. compile-time metaprogramming. A program is composed of regular code and a metaprogram which may change the compilation of the program. It may generate code that is inserted in chosen points of the regular code during compilation. It may do additional checkings in the regular code. The metaprogram can give hints to the compiler on how to parallelize code, mark a method as deprecated or communicate to the compiler that some warnings should be ignored.

Metaprogramming can be made through some form of metadata. For example, annotations as those of Java [10], Scala [20], C# [5], and Groovy [17]. And macros of Nemerle [22] and data annotations of C# [7].
In Lisp [3], metaprogramming can be made with macros. A macro is a function that generates code at compile-time. The code is incorporated into the program for that compilation only. The set of macro functions of a Lisp program is its metaprogram.

This is a very partial list of concepts related to metaprogramming. This name has been used since the beginning of Computer Science [24] (with self-modifying programs) in many different contexts with different meanings. In this paper, we will restrict ourselves to compile-time metaprogramming. The other meanings will not be necessary here. In particular, this article uses the definition of Metaobject Protocol (MOP), which is an interface between a metaprogram and the compiler. As the title says, this paper presents the MOP of the prototype-based statically-typed object-oriented Cyan language. The compiler of Cyan is made in Java. The metaprograms could be made in several languages that interoperate with Java but currently they are made only in this language.

The MOP is composed of selected classes and interfaces of the Cyan compiler and relationships between classes and methods of the metaprogram and the compiler. The MOP specifies which classes and methods of the metaprogram the compiler may use and which classes, interfaces, and methods of the compiler the metaprogram may use.

Groovy is an example of language that supports metaprogramming. This language supports *AST transformations* which are transformations that alter the Abstract Syntax Tree (AST) of a program. An example, found in [12], shows an annotation `@ToString` in a code. The annotation generates a `toString` method that returns a readable string representation of the receiver of the message.

```
import groovy.transform.ToString

@ToString
class Person {
    String firstName
    String lastName
}
```

`ToString` is called an *AST transformation*. It adds to the AST an object that represents method `toString`. This object is the AST of a Groovy method.

Groovy has global and local transformations. Global transformations are applied to every class compiled. They are not triggered by an annotation such as "`@ToString`" in the example above. There is a mechanism that tells the compiler which global transformations should be applied to the current compilation.

Local AST transformations are triggered by an annotation such as "`@ToString`" in the example. They can be performed in six of the nine compilation phases starting in semantic analysis. They cannot be applied during parsing. Both the local and the global transformations can visit nodes of the AST and change them. New code in the form of AST can be produced and inserted in the program during compilation.

The AST of even a simple expression is complex to produce because it demands the creation of objects of different classes. Objects are components of other objects and should be arranged in the correct way. Groovy offers three ways of creating AST automatically from three kinds of inputs. One of ways allows a string to be translated to an object of the AST and then, for example, be added to the AST of the program.

Many other languages such as Haskell [26], Nemerle [22], Scala [20], and LISP [3] support metaprogramming using some way of representing the AST in a more legible form than the creation of objects of the AST directly.

There is another way of representing the AST that we will call "quotation". It has many variations. It is not our goal to explain all of them. A quotation can be delimited by two symbols such as `[<` and `>]` as in

```
[< a = b + 1 >]
```

At compile-time this results in the AST of the assignment `a = b + 1`. The operation of unquote is to insert a value inside the quotation. It is done with $ as in

```
b = [< 4 >]
[< a = $b + 1 >]
```

Now the last quotation represents the AST of `a = 4 + 1`.

The MOP of CLOS [15] was not considered as a basis for the Cyan MOP because the languages are too different from each other and the former is made at runtime demanding optimizations for good performance. Aspects [16] perform many transformations that we would expect in Cyan. But they have a much more limited objective. The protocol for Cyan should be allowed to add code in more places than an Aspect language.

We found the metaobject protocols of current languages inadequate for Cyan. The inadequacies can be seen by examining the goals we set for the Cyan MOP, given below. No current language, to our knowledge, has reached most of these goals.

1. The Cyan compiler is made in Java. A Cyan program could have annotations such as that of Groovy, Java, and XTend. An annotation could have parameters such as "`@init(name, number)`". Annotations should be triggers for the compiler to call code of *metaobjects* and one annotation should be associated with just one metaobject. These should be made in Java or in any language that interoperates with Java. A metaobject should be an object of a Java class. The metaprogram would be then a collection of Java classes. The use of Java enables the loading of a metaobject class at compile-time. The class may not be known when the compiler was itself compiled.
2. The MOP should allow code to be added to prototypes of Cyan programs as strings. It should not be necessary to build an AST object in order to insert code. The objective here is not to transform a string into an AST object and then insert this AST object in a larger AST object as is made in Groovy, for example. The objective is to insert the string in the text of the source unit (the contents of a file) code and then compile the whole file again — the original file would not be changed, only its copy in memory would be. The transformation of a string or other rich data into an AST object would be complex because the Cyan compiler uses information that is local to the insertion point in order to produce the AST objects. There could be limitations on the code to be inserted. Besides that, the insertion of strings in the source code makes it easy to add meta information used in error messages.
3. A single annotation in the source code should be able to act in several compiler phases and not just one of them.
4. In each compiler phase that a metaobject acts, it should be given a specific view of the compiler. This view is a compiler object whose type is a Java interface. This would make it easy to build metaobjects because the amount of information would be limited. Only the meaningful data would be available at a certain compilation phase. For example, the metaobject class should not be able to retrieve the type of an instance variable during

parsing, even if it annotates this variable. The compiler object should supply common operations such as getting the list of the methods of the current prototype (in the semantic analysis).

5. Based on the goals of a metaobject, its Java class should implement one or more interfaces of the MOP. The compiler would then discover that the metaobject implements an interface and it would call one or more methods in specific points of the program. Then the compiler is responsible for calling the metaprogram but based on the choices made by the metaprogrammer (when it chooses which interfaces to implement). The metaprogrammer choice is **static**, made at programming time. The runtime flow is left to the compiler. This is much safer than to let the metaprogram choose when to call the compiler.
6. Metaobjects should not call the compiler to insert code. Instead, they should return the code to be inserted. In order to produce code, a specific method of a specific interface of the MOP should be defined in the metaobject class. And this method should return the code to be inserted. The alternative would be to pass a compiler object to a method of the metaobject class and expect that this method calls a compiler method that adds the code. The responsibility of adding the code would be of the metaprogram. That would be error-prone. Returning the code to be inserted puts fewer responsibilities on the metaprogrammer and is much safer.
7. It should be easy to signal an error in a metaobject. The compiler should point the error line correctly.
8. There should be some way of tracking who inserted the code in which prototype. Error messages should be helpful such as "variable 'x' was not declared. This code was introduced by metaobject 'ameta' of line 23 of source code Test.cyan". We would know who introduced the code with the error.

Some desired functionalities were not design goals of the Cyan MOP. One of them was to have fast compilations. That is at odds with the goals given above. The other is to demand that the metaprogramming is made in Cyan. Since the compiler is made in Java and Cyan is translated to Java, that should not be difficult. It may be so in the future.

3 The Cyan Language

Cyan [1] is a statically-typed prototype-based object-oriented language that supports Java-like interfaces, generic prototypes, optional dynamic typing, anonymous functions, non-nullable types, and an object-oriented exception system. The next code listing shows the contents of a file "`Point.cyan`". It contains the declaration of a **prototype** `Point` introduced by keyword `object`. This prototype belongs to a package called `main` as indicated by the first line of the file. A method declaration starts with `func`. Method declaration and message passing use a Smalltalk-like syntax of keywords.

```
package main

object Point
    func init: Int x, Int y {
        self.x = x;
        self.y = y
    }
    func getX -> Int = x;
    func getY -> Int { return y }
    func x: Int x
```

```
11           y: Int y {
12          self.x = x;
13          self.y = y
14      }
15      var Int x, y
16  end
```

A constructor is a unary method called `init` (no parameters) or `init:` (with parameters). See line 4. This constructor takes two parameters of type `Int`, `x` and `y`. They are assigned to instance variables `x` and `y` in lines 5 and 6. These instance variables are declared in line 15. As in C-like languages, the type of a parameter, local variable, or instance variable is given before the name. `self` is the object that received a message, the same as in Smalltalk or `this` of C++, Java, or C#.

The constructor is called when an object is created. In this case, this can be made with `Point(5, 7)` or `Point new:  5, 7` — see line 7 of the next listing. Inheritance is made with keyword `extends`. Only single inheritance is supported. Any class that does not declare a superprototype inherits from `Any`. This prototype defines some useful methods such as `asString` and `println`.

Cyan has basic types `Boolean`, `Char`, `Byte`, `Short`, `Int`, `Long`, `Float`, `Double`, `Nil`, and `String`. Prototype `Nil` plays a role somehow related to `null` of Java or `nil` of Smalltalk. All types are reference types; that is, a variable of any type is a pointer to a dynamically-allocated object. It is no coincidence that the basic types of Cyan but `Nil` have the same names as the basic types of Java (not considering the case). The Cyan compiler produces Java as output and a prototype such as `Int` is built on `int` of Java.

A method return value type is given after `->` as shown in line 8. Method `getX` of this line is a **unary method** (no parameters) that returns the **instance variable** `x`. This method could have been declared as `getY` of line 9 and vice-versa. Keyword `return` is used for supplying the method return value as in C-like languages. The declaration of an instance or local variable may start with keyword `let` to indicate that the variable is read-only and with `var` for variables that are not read-only.

The method of lines 10-14 uses keywords `x:` and `y:` which are not to be confused with the parameters `x` and `y` of this method. The **name** of this method is “`x:1 y:1`”.

Prototype `Point` is used in prototype `Program` of the code that follows.

```
 1
 2  package main
 3
 4  object Program
 5      func run {
 6             // type of p is deduced from the expression
 7          var p = Point(5, 7);
 8          p getX println;
 9          Point x: 0 y: 0;
10          Out println: Point getY;
11      }
12  end
```

Cyan has literal arrays, literal tuples, and literal hash tables:

```
  // literal array
let Array<Int> ai = [ 0, 1, 2 ];
  // literal tuple
let Tuple<String, Int> t = [. "ecoop", 2018 .];
```

```
assert t f1 == "ecoop" && t f2 == 2018;
  // literal hash table
let IMap<Int, String> map = [ 0 -> "zero", 1 -> "one" ];
```

Assume that the program execution starts in a method `run` of a prototype `Program`. Line 8 shows two unary message sends: first message `getX` is sent to object `p` returning a value of type `Int`. Then message `println` is sent to this `Int` object. In line 9 message `x: 0 y: 0` is sent to prototype `Point`. This is possible because `Point` is an object whose type is `Point` itself. Then a prototype is considered an object whenever it appears in an expression and a type whenever it appears as the type of a variable, parameter, return value type, etc. That means `Int + 2*Int` is a valid expression whose result is `0`. In line 10 the result of `Point getY` is passed as parameter to method `println:` of prototype `Out`. Unary messages have higher precedence than non-unary messages which are called **keyword messages**. Prototype `Out` belongs to package `cyan.lang` imported by every Cyan source code. It has methods for output to the standard output device.

Information on the packages that compose a program is given by a code written in a DSL called **Pyan**. The Pyan code that describes a program should be in a file with extension "`pyan`" that is called **the project file** of the program. An example of such a file is given below.

```
1 program
2     package main
3     package university
```

In line 1 `program` is followed in lines 2 and 3 by the packages that compose the program. If the project file above is in a directory `myprog` there should be subdirectories `main` and `university`, each containing the prototypes of the respective packages. In fact, the Cyan compiler will consider every subdirectory of `myprog` as a package of the program, be it listed in the project file or not. Then the package declarations in this code are redundant. The directory of the program and each package can be specified by putting keyword `at` followed by a literal string with the directory name after the package name. In this case, the directories can be anywhere.

The complete description of Cyan can be found in its manual [1].

4 The Cyan Metaobject Protocol

A metaobject protocol is an interface between the compiler and the metaprogram. The Cyan MOP uses Java classes and interfaces that are part of the Cyan compiler. These Java elements will be called Compiler-MOP. A *metaprogram* is composed by classes that inherit from Compiler-MOP classes and that implement interfaces from Compiler-MOP.

The MOP is not just a set of classes and interfaces of the Cyan compiler. It is also the description of the *interactions* between the Cyan code being compiled, the Compiler-MOP, the metaprogram, and annotations in the Cyan code that tells the compiler which classes of the metaprogram should be used at a certain point of the code. These interactions and the Compiler-MOP will be described in this Section. Before doing that, we will show some examples of how to use the Cyan MOP.

An **annotation** is a linking between the Cyan source code and the metaprogram. In Listing 1, before the declarations of instance variables `name` and `number` there are two "`@property`" annotations. There is another annotation inside prototype `Student`, `@init(name, number)`.

Listing 1 Prototype `Student` that uses metaobject annotations

```
1 package university
2
3 object Student
4     @property var String name
5     @property var Int number
6     @init(name, number)
7 end
```

Listing 2 Prototype `Student` after the “expansion” of the metaobject annotations

```
package university

object Student
    func getName -> String = name;
    func setName: String name { self.name = name }
    var String name

    func getNumber -> Int = number;
    func setNumber: Int number { self.number = number }
    var Int number

    func init: String name, Int number {
        self.name = name;
        self.number = number;
    }
end
```

The compiler will create during parsing a **metaobject** for each annotation found in the source code. Then the compiler will create three metaobjects in this code.

The first annotation in line 4 will cause the creation of methods `getName` and `setName:`. Idem for the annotation of line 5. The annotation `@init(name, number)` of line 6 will ask for the insertion of an `init:` method that initializes instance variables `name` and `number`. After actions asked for by the annotations, the resulting code will be equivalent to that shown in Listing 2. The compiler in fact inserts in this code some metaobject annotations that will be shown later on.

Now we will describe the relationships between annotations, metaobjects, metaobject classes, the Cyan compiler, and packages. The definitions of these concepts are recursive, only after all of them are defined that a global picture can be seen.

There are five kinds of metaobject annotations in Cyan. An identifier preceded by `@` as in Listing 1 is just one of them. These are called at-annotations. Unless said otherwise, we will call an at-annotation just *annotation*. Every annotation in a Cyan source code is associated with a metaobject in a one-to-one relationship. A metaobject is an object of a **metaobject class**. Every at-annotation should be associated with a Java class that inherits from a class called `CyanMetaobjectWithAt` of the Cyan compiler. The other kinds of annotations should be associated with classes that inherit from other superclasses that will not be cited now.

There are rules to be followed to produce and use a *metaobject class*. It will be necessary to explain low-level details of classes and the compiler. Those details will be important in assessing the good and the bad about the Cyan MOP.

First, a metaobject class should be part of the Cyan compiler, one of its Java classes.

Listing 3 Class `CyanMetaobjectProperty` of package `cyan.lang`

```
package meta.cyanLang;
...
public class CyanMetaobjectProperty
    extends   CyanMetaobjectWithAt
    implements IActionProgramUnit_ati {

    public CyanMetaobjectProperty() {
        super("property", AnnotationArgumentsKind.ZeroParameter,
                new AttachedDeclarationKind[] {
                  AttachedDeclarationKind.INSTANCE_VARIABLE_DEC } );

    }
   ...
}
```

The Java compiler will compile the Cyan compiler producing a "`.class`" file for each "`.java`" file. A "`.class`" file contains the bytecodes of the metaobject class and it will be added to a special directory of a Cyan package. After that the metaobject class can be removed from the Cyan compiler and this can be compiled again. This new Cyan compiler, without the metaobject class, will load the "`.class`" of a metaobject class when a source code in Cyan imports the package, which happens during the compilation of a Cyan program. Then instances of the Cyan compiler that do not know anything on a metaobject class will be able to load it during compilation. There are no restrictions on the name of the package of the metaobject class, but we will use packages starting with "`meta.`" as "`meta.cyanLang`".

Class `CyanMetaobjectProperty` of Listing 3 is the real class associated with annotation "`@property`". Its constructor calls the superclass constructor passing as first parameter the metaobject name, which is the annotation name too. In this case, the metaobject name is "`property`" which can be retrieved by calling method `getName()` of class `CyanMetaobject`, the superclass of `CyanMetaobjectWithAt`.

After compiling the Cyan compiler, a file "`CyanMetaobjectProperty.class`" will be produced in a directory

`meta\cyanLang\`

This file should be copied to a sub-directory `--meta` of the directory of a cyan package. It could be any package — the metaobject class does not give any information on which package it will be part of. This particular `.class` file resides in package `cyan.lang`, the main Cyan package that is automatically imported by every source code in the language. Then file "`CyanMetaobjectProperty.class`" should be copied to sub-directory

`cyan\lang\--meta\meta\cyanLang`

"`cyan\lang`" is the directory of package `cyan.lang`, "`-meta`" is the directory where all compiled metaobject classes (.class files) are put, and "`meta\cyanLang`" is the directory of package "`meta.cyanLang`".[1]

Now the Cyan compiler can compile `Student.cyan` that contains the prototype `Student` of Listing 1. The compiler will import package `cyan.lang` at the beginning of parsing and create one object for each "`.class`" file of directory "`--meta`" of the package.[2] These objects,

[1] The name "`cyanLang`" is unrelated to the name of the main Cyan package, `cyan.lang`.
[2] This demands that each metaobject class should declare a parameterless constructor.

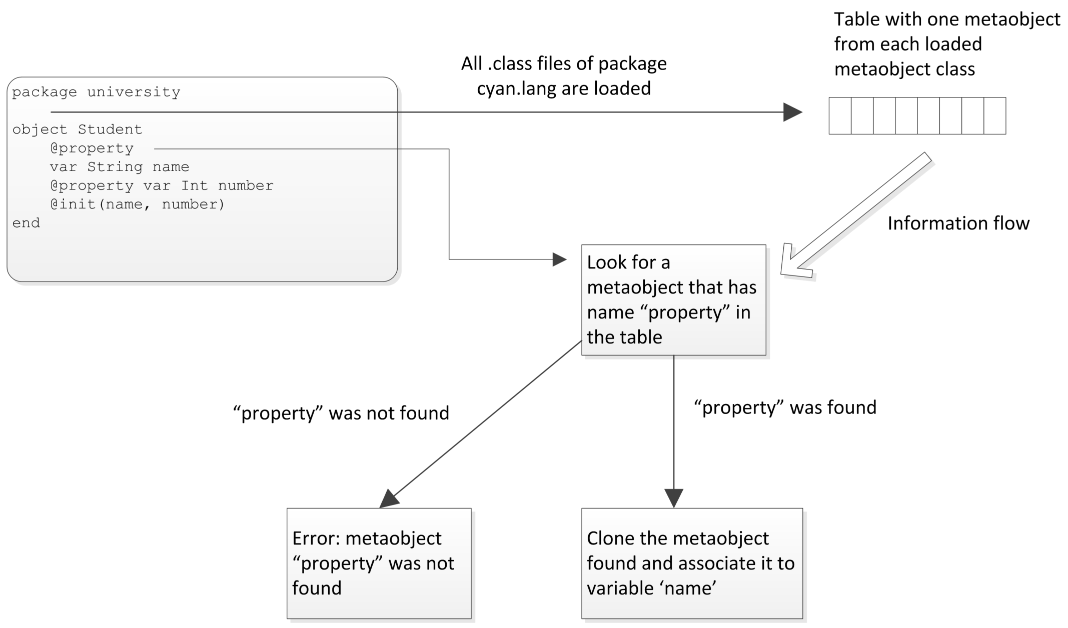
package university
object Student
@property
var String name
@property var Int number
@init(name, number)
end
All .class files of package cyan.lang are loaded
Table with one metaobject from each loaded metaobject class
Information flow
Look for a metaobject that has name "property" in the table
"property" was not found
"property" was found
Error: metaobject "property" was not found
Clone the metaobject found and associate it to variable 'name'

Figure 1 Compiler vision of the use of an annotation

which are in fact metaobjects, are added to a table as shown in Figure 1. This Figure also shows what happens when the compiler finds an annotation `@propert` in the Cyan source code. It searches in the metaobject table for a metaobject whose method `getName()` returns `"property"`. If a metaobject is not found, an error is issued. If it is found it is cloned — for each annotation there should be one object. Note that a metaobject table is built for each source code for it depends on which packages are imported. All metaobject classes of package `cyan.lang` can always be used because this package is always imported. Note that the "`.class`" file of a metaobject, read from a directory "`--meta`", may have been created from another instance of the compiler. Then one can create a metaobject class and make it available to other users without exposing either the metaobject class source code or the original compiler in which the metaobject class was compiled.

At a certain point of the compilation, the Cyan compiler scans all metaobjects of the program being compiled. For each metaobject, the Cyan compiler asks if it implements interface `IActionProgramUnit_ati`. If it does, a method defined in this interface is called to generate code. This code is then inserted in the prototype as a string, changing the source code, an array of chars, in memory (the file with the original source code is not altered). The source code that results is shown in Listing 2. After the code insertion, the program is compiled again to bring to life all the changes introduced by metaobjects.

The explanation of the last paragraph lacks many details of what really happens. To understand the process, it is necessary to explain the compiler phases and how each of them interacts with metaobjects. This is made in the next Subsection.

4.1 The Compiler Phases

One of the goals of the design of the Cyan MOP was to make unnecessary the creation of AST objects by metaobjects. It should be possible to insert new code through regular Java strings. Unlike other languages, strings with code should not be transformed into AST objects and then inserted in some AST object. The strings should be inserted in the source code, represented as an array of chars, which should be compiled again. The original source code would not be altered. This preference for strings has several reasons.

First, it would be difficult to set up the Cyan compiler to compile a code snippet that could be in many different places: inside an expression, inside a method, inside a prototype, or outside a prototype. For each place, the compiler should be set up differently. And the AST produced would have objects in which the values of some instance variables are different too. This is all caused by specific characteristics of Cyan and the Cyan compiler. The metaobject annotations, in particular, make it difficult to insert code as AST objects. A metaobject annotation, for example, may demand information about the source code. Would it be information about the original source code (the char array) or about the new AST that was changed by this same annotation?

Another problem is that in our opinion it would be more difficult to give meaningful error messages. Currently, the Cyan compiler inserts, as text, some metaobject annotations for every code that is inserted by metaobject annotations. For example, annotation `init` in the first example, in fact, produces the following code

```
@pushCompilationContextStatement(ati_id_2, "init", university,
 "C:\Dropbox\Cyan\cyanTests\ecoop\university\Student.cyan", 6)
    func init: String name, Int number {
        self.name = name;
        self.number = number;
    }

@popCompilationContext(ati_id_2)
```

The "push" and "pop" annotations are used in error messages. The "push" annotation has as parameters an identifier `ati_id_2`, the name of the annotation that inserted the code, the package and the full path of the prototype in which this annotation is, and the line number of the annotation. The "pop" annotation takes an identifier that should match that of the "push" annotation. The line number of tokens that come after a "push" and "pop" metaobjects are corrected according to the number of lines between them, which are the number of lines of the code that was inserted. That means the compiler will sign an error in the original source code with the correct line number, being the error before or after the code inserted by a metaobject. For that, some compiler data is changed whenever it parses or analyzes the "push" and "pop" annotations.

Using strings, we have full access to the code that is finally compiled. It can be written to disk if we declare a variable like this:

```
    var Student .# writeCode student;
```

In the directory of the project, the compiler will create a file "`Student-full.cyan`". This file will have the original contents plus the code produced by metaobjects. It is in this way that we got the code above with the `init:` method surrounded by "push" and "pop" annotations.

The Cyan compiler adds code produced by metaobjects as strings to the source code. This code should be compiled again. If after this compilation more code is allowed to be inserted, there should be yet another compilation and so on. To prevent an undetermined number of compilations, the compiler employs a fixed schema. Every program compilation consists of three parsings, three partial semantic analyses, two complete semantic analyses, and one code generation. That could be optimized: the code inserted by metaobjects could be converted to AST objects (like many other languages), which would save at least two parsings. By the reasons given previously, that is not done.

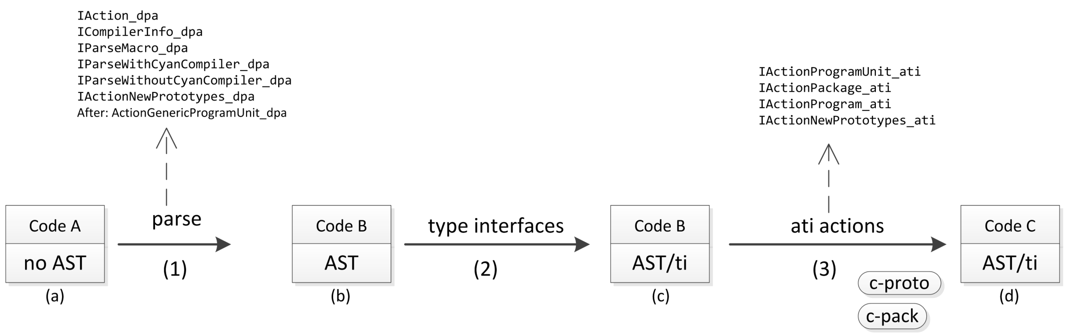

Figure 2 First compiler phases

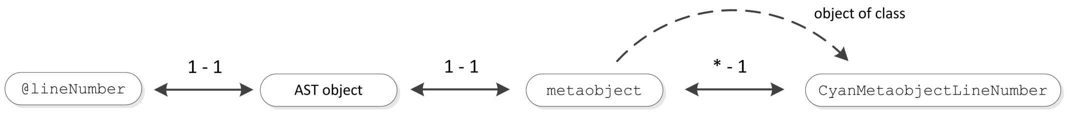

Figure 3 Relation between metaobjects, annotations, and metaobject classes

4.2 The First Three Compiler Phases

The Cyan compiler has ten compilation phases. The first three of them are shown in Figure 2 (each arrow is a phase). The compilation starts in the left of this Figure and proceeds in the direction of the solid arrows. The leftmost rectangle, divided into two parts, represents the source code (**S. code**) of the program which is composed of one source code for each Cyan prototype. The "**no AST**" below means that there is no AST at the start of the compilation. The rectangles are labeled using letters and represent the source code and the AST of the program.

Each solid arrow is labeled by a number and represents a compiler phase. The solid arrow between (a) and (b) represents the syntactic analysis which produces the AST of (b) and a possible modified source code, indicated by an asterisk: "**S. code***". The code can be modified by metaobjects that acts in phase (1).

The dashed arrows in phase (1) point to a list of Java interfaces of the Cyan compiler. These interfaces are part of the Cyan MOP and they can be implemented by metaobject classes. If one does and the associated annotation is used in the program, all the methods of the metaobject class that override the Java interface methods are called on the metaobject associated with the annotation. This is better explained by an example. Suppose there is a code

```
    Out println: "This is line number " ++ @lineNumber;
```

For annotation `@lineNumber` the compiler will create one AST object and one metaobject of class `CyanMetaobjectLineNumber`.

Each annotation in the source code causes the creation, during parsing, of one AST object that represents it. For each of such objects, there is exactly one metaobject. But a metaobject class can be used to create many metaobjects. The relationships are illustrated in Figure 3.

Class `CyanMetaobjectLineNumber` is associated with the Cyan package `cyan.lang` and it implements interface `IAction_dpa` that is used for specifying actions in phase (1) — see Figure 2. Then **during** phase (1), when the annotation is found, the compiler will call method `dpa_codeToAdd` of the metaobject. This method is the only one defined in interface

IAction_dpa and it will return a text that should be a valid Cyan expression. This text is inserted **after** the annotation. The parsing then proceeds in the inserted code. To make this clear, @lineNumber is replaced by the code below supposing that the annotation is on line 13. After the insertion, the parsing continues in the "push" annotation

```
@lineNumber#dpa
@pushCompilationContext(dpa0, "lineNumber", "main",
   "C:\Dropbox\Cyan\cyanTests\ecoop\main\Program.cyan", 13)
   13
   @popCompilationContext(dpa0, "cyan.lang", "Int")
```

This modified code will pass through two more parsings. However, this annotation will not be used anymore during parsing because it has a suffix "#dpa". It means "During PArsing" and indicates that the annotation is not to be considered in future parsings of the source code. However, this same annotation could be used in the semantic analysis to produce more code, do checkings, create prototypes, and so on.

The "pop" annotation of the above example has an identifier dpa0 and two strings which are a package name and a prototype name. The metaobject associated with the annotation @lineNumber of this example, an instance of CyanMetaobjectLineNumber, is asserting that the code produced by method dpa_codeToAdd of IAction_dpa has type cyan.lang.Int. That is, the metaobject asserts that "13", the code the method returns, has type Int. This is used to check if the metaobject really produces what it promises.

Class CyanMetaobjectWithAt, superclass of CyanMetaobjectLineNumber, defines methods that return the package and prototype of the metaobject annotation if it is to be considered as an expression. The default is null, meaning that an annotation of that metaobject class should not be considered as an expression. This is the case of the metaobject class CyanMetaobjectInit associated with annotation init of the example of Listing 1.

Annotations associated with CyanMetaobjectLineNumber should be considered expressions of type Int and this class has methods that return strings "cyan.lang" and "Int". These two strings are put in the "pop" annotation of the last example.

The Java interface IAction_dpa defines a method

```
    default StringBuffer dpa_codeToAdd(ICompilerAction_dpa compiler) {
        return null;
    }
```

Most of the interface methods of the MOP are defined like this. A null return means that no code should be inserted. Even when returning null a method can be useful. It can do checkings, for example. Parameter compiler of this method returns a restricted view of the compiler. Through it, one can access the few available information during parsing such as the name of the current prototype.

Class CyanMetaobjectLineNumber implements interface IAction_dpa and overrides method dpa_codeToAdd:

```
    @Override
    public
    StringBuffer dpa_codeToAdd( ICompilerAction_dpa compiler ) {
        CyanMetaobjectWithAtAnnotation annot =
             this.getMetaobjectAnnotation();
        return new StringBuffer("" +
            annot.getSymbolMetaobjectAnnotation().getLineNumber() );
    }
```

The local variable `annot` refers to the AST object that represents the annotation (see Figure 3). Through it one can get the symbol of the annotation, “`@lineNumber`” in the code, and then the line number of the symbol. “Symbol” here is the token of the lexical analysis.

An annotation may be **attached** to a declaration such as prototype, method, or instance variable. Annotation `@property` of the first example (prototype `Student`) is attached to the declaration of an instance variable. The metaobject class of this annotation is `CyanMetaobjectProperty` of Listing 3. In line 9 an array is passed to the superclass constructor indicating to which declarations an annotation `@property` may be attached to. Annotations of the same metaobject class may then be attached to several different declarations. `property` may only be attached to instance variable declarations. An annotation need not be attached to a declaration. Annotation `init` in Listing 1, for example, is not attached to any declaration.

Using methods of local variable `annot` of the example above we can retrieve information on the annotation. For example, its parameters (if any), the symbol (token) of the annotation (as just seen), the declaration to which the annotation is attached to (if any), the file in which the annotation is, and so on.

Phase 2 of the compilation of a Cyan program is part of the semantic analysis. At the start of this phase, the compiler has parsed all the prototypes of the program but some information is missing: types. Many AST objects are associated with types: expressions, instance variables, local variables, parameters, method return value types, declared superprototypes, and implemented interfaces of a prototype. All of these AST objects have a field `type` that should refer to the AST object that represents the type. All of these AST objects but expressions have a field `typeInDec` for the declared type (of a variable, for example). The declared type `typeInDec` is represented by strings.[3] At the end of parsing, `typeInDec` refers to an AST object and `type` is `null`. Phase 2 sets the field `type` of instance variables, method parameters, method return value types, declared superprototypes, and implemented interfaces of a prototype.

```
var Int n;
```

As an example, just after parsing the AST object that represents `n` has a non-null field `typeInDec` with the data `"Int"`. Its field `type` is `null`. At the end of phase 2 `type` refers to the AST object that represents prototype `Int`.

Phase 2 assigns AST objects representing types to everything that is not inside methods. Therefore the word “interfaces” in “type interfaces” that appears in phase 2 shown in Figure 2 includes even private instance variables.

The Java interfaces associated with phase 3 of the compilation are those pointed to by the dashed arrow of (3) of Figure 2. The only Java interface that matters for now is `IActionProgramUnit_ati`. This Java interface should be implemented by all metaobject classes that need to add instance variables, shared variables, and methods to prototypes. Besides that, code can be add to the start of existing methods. This Java interface is implemented by the metaobject class `CyanMetaobjectInit` associated with annotation `init`. This class redefines a method of the interface that generates code for the `init:` method. First it checks whether the parameters to the annotation, `name` and `number`, are instance variables. It is necessary to retrieve their types too, which is possible since phase 2 already attributed types to instance variables.

[3] This is an oversimplification that does not invalidate the argument.

Listing 4 A Pyan code with annotations
```
@feature("author", "Isaac Newton")
program
    package main

    @feature(constants, [ "UA" -> 0, "UB" -> 1 ] )
    @checkStyle
    package university
```

In phase 3 the Cyan compiler scans the AST and collects all metaobjects whose classes implement `IActionProgramUnit_ati`. These metaobjects are associated with annotations in the source code. Then the compiler calls all methods of `IActionProgramUnit_ati` on the collected metaobjects. The methods may do checkings and may ask the compiler to insert code in prototypes. The insertion of code is made after all metaobject methods are called, at the end of the phase. The metaobjects may also visit the AST nodes they have access to. In particular, a metaobject attached to a declaration can visit the AST nodes of this declaration (a prototype, method, instance variable). It is also possible to visit the method and the prototype in which the annotation is. It is simple to visit a declaration because the Design Pattern Visitor is supported by the compiler. Interface `IActionProgramUnit_ati` defines a method that permits to rename a method and another to create a new prototype.

As explained in Section 3, every Cyan program has a "`.pyan`" project file that defines precisely which are the packages it is composed of and where are these packages in the file system. Annotations can also be used in the project file as shown in Listing 4. An annotation `feature` is attached to the whole program and another to package `university`. `checkStyle` is attached to `university` only. A metaobject class of metaobjects that acts on programs and packages should implement some selected Java interfaces, among them `ICompilerInfo_dpa`, `IActionPackage_ati`, and `IActionProgram_ati`. The last two of these interfaces define methods to add code to prototypes of the package and to every prototype of the program, respectively. Of course, both prevent separate compilation of prototypes. Currently, that is not a problem because the compiler does not support separate compilation — all the code is compiled in every compilation.

A metaobject of a metaobject class that implements `IActionProgramUnit_ati` can insert code in the prototype in which the annotation is. Only additions of code are allowed, removals can never be done. A metaobject can insert code in any other prototype of the same package if annnotation

```
@feature(communicateInPackage, on)
```

is attached to the package in the project (.pyan) file. Again, that would prevent separate compilation of prototypes.

4.3 Phases 4 to 6

After the changes of phase 3 all the program is parsed again in phase 4 as shown in Figure 4. The interfaces of this phase are only used in new prototypes created in previous phases. New prototypes may be created by:

1. metaobjects that implement Java interfaces `IActionNewPrototypes_dpa` and `IActionProgramUnit_ati` and;
2. instantiations of generic prototypes in phase 2. An instantiation of a generic prototype is the creation of a new prototype by replacing, in the source code, the formal parameters

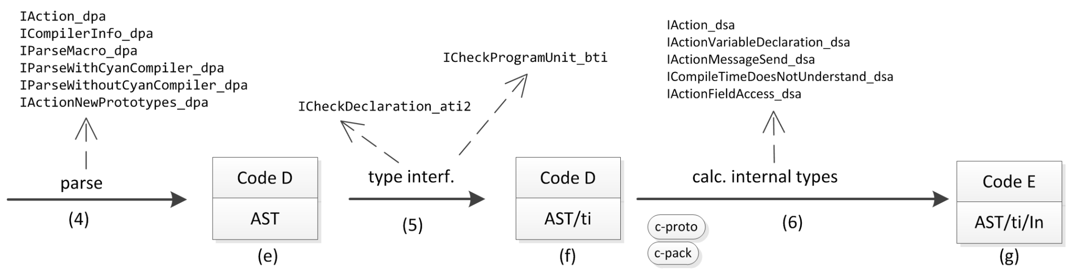

Figure 4 Phases 4 to 6

by the real parameters.

Phase 5 does the same job as phase 2 and has two Java interfaces used for checkings only. `ICheckProgramUnit_before_dsa` can only be implemented by metaobject classes whose metaobjects are attached to prototypes (Cyan interfaces and regular prototypes). It has one method that cannot add code. This method is called after phase 5.

The other Java interface of phase 5 is `ICheckDeclaration_ati2` which also has a method for checking but it is called during phase 5.

Phase 6 is responsible to assign types for variables and expressions that are inside methods. It does the rest of semantic analysis too which comprises all the checkings specified in the Cyan manual. This phase has the most interesting Java interfaces. `IAction_dsa` has a method to add code after the annotation and another to create new prototypes. This same two actions can also be done in phase `IActionProgramUnit_ati`. So why should one use `IAction_dsa`? The answer is the parameter that each of these methods takes:

```
    StringBuffer dsa_codeToAdd(ICompiler_dsa compiler_dsa);
    ArrayList<Tuple2<String, StringBuffer>> // return type
        dsa_NewPrototypeList(ICompiler_dsa compiler)
```

The parameter has type `ICompiler_dsa`. This interface supplies methods with much more information on the compilation than the parameter of the methods of `IActionProgramUnit_ati`. For example, a method can return a hash map with elements (key, value) in which key is a package/prototype name and value is the set of all subprototypes of the prototype. Another method can search for local variables visible at the point of the annotation. This information is not available in phase 3.

The Java interface `IActionVariableDeclaration_dsa` should be implemented by metaobject classes whose metaobjects should be attached to local variable declarations and that need to add code after the declaration. `IActionMessageSend_dsa` should be implemented by metaobject classes whose annotations should be attached to method declarations. Its methods allow replacing a message passing by any code. But not every message passing, just the message passings that could cause the calling of the annotated method. For example, metaobject class `CyanMetaobjectInline` implements this Java interface. The annotation name is, of course, `inline`:

```
object M
    @inline{*  2*n  *}
    final func twice: Int n -> Int = n + n;
end
```

Annotation `inline` has a novelty. It has a text delimited by `{*` and `*}` which is "`2*n`". We will call this the "text of the annotation" or "DSL of the annotation" because this text between the delimiters may be a Domain Specific Language (DSL). A metaobject class whose metaobjects should take a text or DSL should override method `shouldTakeText` inherited from `CyanMetaobjectWithAt`. This method should simply return `true`.

Class `CyanMetaobjectInline` overrides method

```
Tuple2<StringBuffer, Type> dsa_analyzeReplaceMessageWithSelectors(
            ExprMessageSendWithSelectorsToExpr messageSendExpr,
            Env env)
```

of interface `IActionMessageSend_dsa`. This method should return a tuple composed by the code that replaces the message send and the type of the code, which should be an expression. Parameter `messageSendExpr` is the AST object of the message send that caused the method of `CyanMetaobjectInline` to be called. `env` is the environment which can be used to retrieve compilation information. In a message send

    d = M twice: 2 + 3;

`messageSendExpr` is the AST object of "`M twice: 2 + 3`". The code returned is

```
    2*(2 + 3)
```

which replaces "`M twice: 2 + 3`". A parameter should not appear twice in the DSL of the annotation — this is a crude example of metaobject.

Interface `ICompileTimeDoesNotUnderstand_dsa` of Figure 4 defines methods for intercepting message sends for which the compiler could not find the appropriate method. This is the compile-time version of method `doesNotUnderstand:` of Smalltalk [9]. A metaobject of a metaobject class that implements this interface can be attached to a prototype or a method. `CyanMetaobjectGrammarMethod` is the only metaobject class of the standard library that implements this interface. A metaobject of this class can be used to implement Domain Specific Languages (DSL) whose tokens are message keywords and parameters. The language is specified through regular operators as in the example of Listing 5. The annotation name is "`grammarMethod`" and it takes a DSL that specifies a language through regular expressions operators. The terminals of this language are method keywords as "`move:`" and typed expressions, specified as types in the DSL. In the example, this annotation is attached to a method `action:` that takes a complex parameter that will soon be explained. A message send to `Car` could be

```
    Car sound: Sound("The Girl from Ipanema") move: 20 left:
        move: 30 right: 70
        sound: Sound("Swan Lake") time: 100;
```

Prototype `Car` does not declare a method with all of these keywords. Then the compiler looks for a metaobject that implements interface `ICompileTimeDoesNotUnderstand_dsa`. This search is made in the metaobjects attached to methods of the prototype of the receiver of the message (in textual order), this prototype, methods of the superprototype, superprototype, and so on (up in the hierarchy). If a metaobject implementing `ICompileTimeDoesNotUnderstand_dsa` is found, one of its methods is called. If the method returns `null` the searches continues. Otherwise, it returns a code that should replace the message.

The metaobject class `CyanMetaobjectGrammarMethod` tests whether the message matches the regular expression specified by the DSL. If it does, it returns a call to the method the annotation is attached to. The sole argument to this method is a raw AST composed by literal arrays, literal tuples, and union objects. The type of the sole parameter of `action:` in the example was calculated from the regular expression. If the programmer leaves the

Listing 5 Example of grammar method use

```
package main

object Car
    @grammarMethod{*
        ( move: Int |  (sound: Sound (time: Int)? ) |
          left: | left: Int | right: | right: Int  )+
    *}
    func action: Array<
        Union<f1, Int,
              f2, Tuple<main.Sound,
                      Union<some, Int, none, Any>>,
              f3, Any, f4, Int, f5, Any, f6, Int>>
            params {
        // code that moves the Car
    }
    ...  // elided
end
```

parameter without a type the compiler will sign an error and tell the correct parameter type. There are many details on this method that were omitted. They will be explained in a future article.

The DSL that is attached to an annotation needs, in general, to be parsed. That can be made with or without the help of the Cyan compiler. In the first case the metaobject class should implement interface `IParseWithCyanCompiler_dpa` as `CyanMetaobjectGrammarMethod` does. In the second case, the metaobject class should implement interface `IParseWithoutCyanCompiler_dpa`. The methods of these interfaces are called in phases 1 and 4 of the compilation — see the Figures.

Interface `IParseWithCyanCompiler_dpa` declares a single method

```
void dpa_parse(ICompiler_dpa compiler_dpa)
```

Interface `ICompiler_dpa` declares methods for parsing such as `getSymbol()` that returns the current symbol (a symbol of the Cyan compiler), `type()` that parses a type, and `statement()` that parses a Cyan statement. `dpa_parse` makes it easy to implement DSL that resemble Cyan such as that of `grammarMethod`. It is even possible to pass expressions back to the Cyan compiler so that these expressions go to the semantic analysis as if they were inside a method.

The metaobject class `CyanMetaobjectConcept` also demands that DSL code be attached to its annotations. Metaobjects of this class are used to implement *concepts* [11] [2] which are predicates on types used as real parameters to generic prototypes.

```
package main

@concept(test){*
    T has [ func * (T other) -> T
            func unit -> T
            func inverse -> T ],
        """T should have methods *, unit, and inverse
           in order to be considered an element of a Group""",

    axiom opTest: T a, T b, T c {%
```

```
        if a * (b * c) != (a * b) * c ||
           c * (b * a) != (c * b) * a {
             ^"T is not associative"
        }
        ^Nil
    %},
    // elided
*}
object GroupWork<T>
    // elided
end
```

In this example the DSL code of annotation `concept` demands that parameter `T` have methods `*`, `unit`, and `inverse` with the specified signatures. When the generic prototype is instantiated, a new source code is created from the source code of the generic prototype. A real parameter replaces `T` in this new source code. `T` in the DSL code is also replaced by the real parameter — the metaobject class can choose if it will be replaced or not.

The metaobject associated with the annotation of this example checks whether the real parameter to the instantiation has the methods specified in the attached DSL code. If it has not, an error is issued by the metaobject. The DSL of `concept` supports a rich syntax. The DSL code can demand that a prototype is a superprototype or subprototype of another one, that a prototype implements an interface, that a parameter of a method is equal to the return value of another method (and many variations of this), that a prototype is in a list of prototypes, etc.

Method `dpa_parse` of `CyanMetaobjectConcept`, from `IParseWithCyanCompiler_dpa`, parse the DSL code of an annotation and puts all the types found in a list managed by the Cyan compiler. This is made in phase `1`. In phase `3` the Cyan compiler does the semantic analysis of this list. It does so for all metaobject annotations that have a non-empty list like that. Not only types but also any expression can be inserted in this list.

The metaobject associated with the annotation `concept` also generates a test code with the contents that follow "`axiom`" in the DSL code. Let us explain that.

Cyan reserves a directory called `--test` in the project directory for tests. In the directory `--test` a metaobject may add Cyan files for testing the program. These test files are not considered part of the program, it is necessary to build another project to use them.

Each compiler object passed as a parameter to methods of the MOP interfaces declare a method to write a test prototype to a file. The method demands only high-level information such as the prototype name associated with the test and its package. The programmer is responsible to create Cyan code that calls methods of the test prototypes.

The metaobject associated with the annotation `concept` of the previous example creates a test prototype containing the contents of the axioms of the DSL code. It also creates prototypes that emulate the parameters to the prototype with all restrictions of the DSL code. More information on that can be found in the Cyan language manual.

There are other special directories like `--test` in which the metaobjects may write code to. These directories may have files with DSL code, permanent data, and temporary data. Some directories are read-only and are part of the Cyan program.

`concept` does not demand that it is used in generic prototypes. It can be used in non-generic prototypes to assure that the code obeys the DSL requirements and to generate prototypes for testing.

Interface `ICheckMessageSend_dsa` of Figure 4 is used to check message sends. The annotation should be attached to a method. Metaobject class `CyanMetaobjectDeprecated`

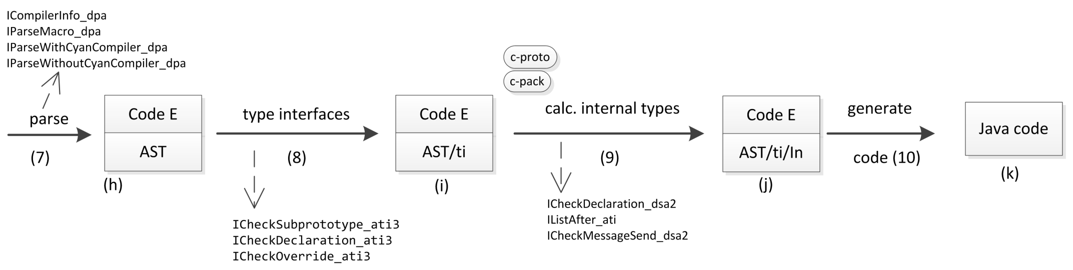

Figure 5 Phases 7 to 10

implements this interface. If an annotation `deprecated` is attached to a method, the metaobject will issue an error in every message send that can possibly call this method. A metaobject class implementing `ICheckMessageSend_dsa` can do interesting checkings. For example, it can demand that the receiver is a prototype, that a certain parameter is a literal string that obeys a certain pattern, that the type of a real parameter to the message defines a method “`eval:`”, etc. It is even possible to design a DSL that demands which are the restrictions of the parameters and message receiver.

4.4 Phases 7 to 10

The source code cannot be changed in the remaining phases of the compilation, which are shown in Figure 5. Only checkings are allowed. Interface `ICheckDeclaration_ati3` of phase 8 should be implemented by metaobjects that need to do any checkings. The annotations associated with this interface should be attached to a declaration such as prototype, instance variable, method, package, or program. The declaration to which the annotation is attached can be retrieved. It is an object of the AST that can be visited using the Visitor Design Pattern. Interface `ICheckDeclaration_dsa2` is similar to `ICheckDeclaration_ati3`. The sole method of this interface takes a parameter that supplies information on types of local variables and expressions of the Cyan code.

A metaobject class that implements interface `ICheckSubprototype_ati3` should define a method

```
    ati3_checkSubprototype(ICompiler_ati compiler_ati, Type subproto)
```

An annotation of this metaobject class should only be attached to a prototype. This method is called whenever the prototype is inherited. Parameter `subproto` is the subprototype.

Interface `ICheckMessageSend_dsa` of phase 9 can be used to check message sends. It is used to check if a method calls non-deterministic methods (directly or indirectly) in a replication framework made in Cyan [29].

5 Comparison to other Metaobject Protocols

The Cyan MOP does not have many features found in other Metaobject Protocols or reflective systems. For example, a metaobject cannot intercept the access to an instance variable as in CLOS [15], inheritance cannot be changed as in Xtend [33], objects cannot be added or removed from the AST as in Groovy [17] (and many other languages), methods and instance variables can be added but not removed as in Self [30]. The Cyan MOP is unsuitable for multi-stage programming as in MetaOCaml [28], which is made at runtime. In Cyan the evaluations should be made at compile-time in only two or three steps because of the fixed number of compilation phases. Then `power(2, 5)`, 2^5, would not result in `32` even if the language supported a kind of multi-stage programming at compile-time.

C++ templates [27] are Turing complete [31]. Integer numbers can be passed as parameters to templates (classes or functions) and a sum in a template parameter such as `1 + 1` is made at compile-time. This is not allowed in Cyan. The compiler can use metaobjects for computations but it does not do any arithmetic evaluation by itself at compile-time.

Language Scala [21] supports virtualization of some control structures such as `while` loops and `if` statements. These can be translated into method calls. Nothing similar exists in Cyan.

Language Nemerle [22] supports **macros** which play the role of typical Lisp macros and some roles of Cyan metaobjects. A macro may be attached to a class and insert code in it, for example. Nemerle distinguishes three compilation phases on which a macro may act: after parsing and before setting the superclasses, after the superclass of every class have been set, and a phase in which the types of fields and methods have been set (roughly equivalent to phase ati of Cyan). However, a macro may act on just one of these phases.

Nemerle, Scala, Groovy, and Xtend use the language itself for metaprogramming. The base and the metalevel language is the same. Unfortunately, this is not true for Cyan. The metalevel is in Java. It would not be impossible to change that since the compiler translates Cyan to Java. But this is not planned. Nemerle, Scala, Groovy, and Xtend have certainly much faster compilations than Cyan. The whole design of the Cyan MOP did not care for compilation time. This can be changed but we suspect that the programming time for that would be very high.

6 Conclusions

This article described one kind of metaobjects in Cyan, those associated with at-annotations, with “`@`”. There are more: literal numbers ending with an identifier such as `0101bin` or `FFAB_Hex`, literal strings starting with an identifier such as `r"0*1+"`, macros, **Codegs**, and literal objects delimited by a sequence of characters such as

```
    [# 1..100 | it*2 #]  // produces a set with the elements
```

All of these metaobject kinds have all or almost all of the power of the metaobjects described here. For example, Cyan macros can add methods to the current prototype, communicate with other, and generate code in phase 6 of compilation.

Codegs are a visual form of metaobjects associated with at-annotations. A metaobject class of a Codeg should implement interface `ICodeg` and it can implement any interface a regular metaobject class can. They are supported by a plugin to the Eclipse IDE. When editing a Cyan source code the plugin pays attention to the mouse position. If the mouse is over a Codeg annotation such as `color`,

```
    var Int redColor = @color(red);
```

a method of the metaobject associated with the annotation is called. A window is opened that allows one to choose a color through a graphical interface. The chosen color is kept by the compiler (not the metaobject) in a hidden file that can be retrieved at compilation time.

The Metaobject Protocol of Cyan and other languages share many features. Some are unique to Cyan. Even the common features are supported in a different way in this language. Below we cite the important characteristics of the Cyan MOP.

Metaobjects are associated with regular Cyan packages. When a package is imported its metaobjects can be used. No special compiler option is necessary.

Interfaces direct how a metaobject will be used by the compiler. Then the designer of a metaobject class chooses the interfaces needed for the goals of the metaobjects. This is a design-time decision made before compilation. Therefore the metaobject class does not need to choose, at compilation time, when to add code, do checkings, etc. The compiler, directed by the interfaces, take the actions. This simplifies the metaobject classes because the flow of execution becomes predictable. In our experience in implementing some one hundred metaobject classes, this is very important. It brings peace of mind.

When the compiler calls a method of an interface, implemented by a metaobject class, it usually passes a compiler object as a parameter. This object depends on the compilation step. During parsing, for example, the compiler object is able to supply too little information. It can supply more during the semantic analysis. With the code completion of modern IDEs, it becomes easy to discover what is available and program metaobject classes. Because of this and the overall design of the MOP, simpler metaobject classes like that of `inline` can be made and tested in 30 minutes.

Metaobjects can act in multiple compiler phases. A single metaobject can produce code during parsing, in phase 3, and during semantic analysis. It may also do checkings in phases 1, 3, 6, 8 and 9. A metaobject can also generate code that has metaobjects that generate code or do checkings in later phases.

A metaobject can add code in multiple places in a single step. In phase 3, a metaobject can create a new prototype, add code after the associated annotation, add code to the prototype the annotation is, and add code to other prototypes of the same package.

Cyan metaobjects can communicate with each other in the same source file, even if their metaobject classes are different. Then a macro call can communicate with the metaobject associated with a string prefixed by an identifier or an at-annotation. If the package feature `communicateInPackage` has value `on`, metaobjects of the same package can communicate with each other. A package feature is a value associated with a package.

The MOP was built to minimize the use of the AST. Code to be inserted should be given as a Java string and most of the information necessary to build metaobject classes may be got from the compiler object passed as a parameter to most methods. Only a few of the nearly one hundred metaobjects need to access the AST. However, the AST is available. Every metaobject annotation attached to a declaration has access to the AST of this declaration. Every metaobject has access to the current prototype, current method, package of the current prototype, and so on. All AST classes of the compiler implement interface `ASTNode` that declares an `accept` method. Then the compiler supports the Design Pattern Visitor. A metaobject can visit an AST node in order to generate code, for example.

Parameters to a metaobject can be literals of basic types, strings, literal arrays, literal tuples, and literal hash tables. Any combinations and nestings of these values are allowed. In the bottom (leafs) there should be literal values of basic types. An annotation can also have an attached DSL as `grammarMethod`, `inline`, and `concept`. The DSL code can direct the code generation and checkings. We found this to be very important. Metaobject `overrideTest` is used to create test cases. An annotation of it should be attached to a method. Whenever the method is overridden in a subprototype a test case is created based on the DSL code of the annotation. As another example, a metaobject associated with an annotation attached to a prototype could do checkings when the prototype is inherited. Depending on the DSL code of the annotation, the metaobject could check if some methods are correctly overridden. For example, if a certain method is overridden, the subprototype method should call the original method. The call relationships among the subprototype methods should obey a pattern defined by the DSL code of the annotation. The most basic one would be like this:

"the method that overrides method `draw` should call the superprototype method as its first statement". Inheritance would be less error-prone with a metaobject like this.

As another example, a metaobject associated with an annotation attached to the program could restrict the way the packages interact. Depending on the DSL code of the annotation, a package would not be allowed to import another, for example. And a package could only import some other specific packages — it could not, for example, import network packages. The DSL code could describe part of the high-level architecture of the program and enforce it.

The Cyan compiler is responsible to handle files that may be created or read by metaobjects. These may be temporary files, files with DSL code or with test code. For example, the code attached to a `concept` annotation may be put in a file an reused. All the file management is made by the compiler, metaobjects do not know the absolute file paths. Metaobjects may generate test code too that is put in a special directory. These files can be used as a memory between compilations. A metaobject may detect mistakes caused by changes in the environment between compilations. For example, an instance variable `key` is used in a method changed by the addition of a parameter `key`. The meaning of the previous use of this variable is changed. Based on information on the previous compilation, a metaobject could detect the problem. This is called Link Past Future (LPF).

Metaobject annotations can be attached to regular declarations such as methods, instance variables, local variable declarations, and prototypes. They can also be used in the project file of a program and can be attached to packages and the program itself. We do not know any other language that allows that.

Unlike the languages that support compile-time MOP cited in Section 5, metaobjects of Cyan are used extensively in the prototypes of package `cyan.lang`. This package contains the fundamental prototypes of the language: the basic types, `Any`, `Array<T>`, prototypes for the exception system, tuples, and anonymous functions. The code of a tuple prototype such as `Tuple<Int, String>` is produced by a metaobject. Idem for prototypes that are the types of anonymous functions. Without metaobjects, Cyan would be a very different language. Unfortunately, there is no space here to describe metaobjects used in package `cyan.lang` and metaobjects that support generic prototypes, documentation, testing, tests for the Cyan compiler itself, etc.

Since the Cyan MOP has all of the characteristics cited above, we believe that it has already reached its goals. The Cyan compiler is fully functional and supports all of the features cited in this text. And many more, there is no space to describe further details. The compiler and documentation can be found in `www.cyan-lang.org`. This site has an "articles" page with all the code of this paper.

The design of the MOP has not ended. We hope to improve the protocol in the following months and add annotations to types like `Int@range(1,10)` and `Char@letter` (they explain themselves).

References

1 Same author as the paper. The Cyan language, 2017. URL: `http://www.cyan-lang.org/`.

2 Anya Helene Bagge and Magne Haveraaen. Axiom-based transformations: Optimisation and testing. In Jurgen J. Vinju and Adrian Johnstone, editors, *Proceedings of the Eighth Workshop on Language Descriptions, Tools and Applications (LDTA 2008)*, volume 238, pages 17–33. Elsevier, 2009. URL: `http://www.ii.uib.no/~anya/papers/bagge-haveraaen-ldta08-axioms.html`, doi:10.1016/j.entcs.2009.09.038.

3 Conrad Barski. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

4 Common language runtime (clr), November 2017. https://docs.microsoft.com/en-us/dotnet/standard/clr. URL: `https://docs.microsoft.com/en-us/dotnet/standard/clr`.

5 C# language specification, September 2014. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf. URL: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`.

6 Robertas Damaševičius and Vytautas Štuikys. Taxonomy of the fundamental concepts of metaprogramming. In *Information Technology and Control*, volume 37. Kaunas University of Technology, 2008.

7 Adam Freeman, Matthew MacDonald, and Mario Szpuszta. *Pro ASP.NET 4.5 in C#*. Apress, Berkely, CA, USA, 5th edition, 2013.

8 The go programming language, November 2017. https://golang.org/. URL: `https://golang.org/`.

9 Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

10 James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

11 Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: Linguistic support for generic programming in c++. *SIGPLAN Not.*, 41(10):291–310, October 2006. URL: `http://doi.acm.org/10.1145/1167515.1167499`, `doi:10.1145/1167515.1167499`.

12 Runtime and compile-time metaprogramming, November 2017. http://groovy-lang.org/metaprogramming.html. URL: `http://groovy-lang.org/metaprogramming.html`.

13 The hack programming language, November 2017. http://hacklang.org/. URL: `http://hacklang.org/`.

14 Apple Inc. *The Swift Programming Language (Swift 4)*. Apple Inc., 1st edition, 2014.

15 Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

16 Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In Jørgen Lindskov Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

17 Dierk Koenig, Andrew Glover, Paul King, Guillaume Laforge, and Jon Skeet. *Groovy in Action*. Manning Publications Co., Greenwich, CT, USA, 2007.

18 Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java Virtual Machine Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.

19 The llvm compiler infrastructure, November 2017. https://llvm.org/. URL: `https://llvm.org/`.

20 Bill Venners Martin Odersky, Lex Spoon. *Programming in Scala: Updated for Scala 2.12 : a comprehensive step-by-step guide*. Artima, Incorporated., Artima Press [Imprint, 3ed. edition, 2016.

21 Adriaan Moors, Tiark Rompf, Philipp Haller, and Martin Odersky. Scala-virtualized. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*, PEPM '12, pages 117–120, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2103746.2103769`, `doi:10.1145/2103746.2103769`.

22 The nemerle programming language, October 2017. http://nemerle.org/. URL: `http://nemerle.org/`.

23 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 201–212, New York, NY, USA, 2008. ACM. URL: `http://doi.acm.org/10.1145/1390630.1390656`, `doi:10.1145/1390630.1390656`.

24 Raul Rojas and Ulf Hashagen, editors. *The First Computers: History and Architectures*. MIT Press, Cambridge, MA, USA, 2002.

25 The Rust language, September 2018. http://www.rust-lang.org/. URL: `http://www.rust-lang.org/`.

26 Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, Haskell '02, pages 1–16, New York, NY, USA, 2002. ACM. URL: `http://doi.acm.org/10.1145/581690.581691`, `doi:10.1145/581690.581691`.

27 Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, 4th edition, 2013.

28 Walid Taha. Generative and transformational techniques in software engineering ii. chapter A Gentle Introduction to Multi-stage Programming, Part II, pages 260–290. Springer-Verlag, Berlin, Heidelberg, 2008. URL: `http://dx.doi.org/10.1007/978-3-540-88643-3_6`, `doi:10.1007/978-3-540-88643-3_6`.

29 the author names will be present in the final version. Transparent replication using metaprogramming in cyan. In *Proceedings of the 21st Brazilian Symposium on Programming Languages*, SBLP 2017, pages 9:1–9:8, New York, NY, USA, 2017. ACM. URL: `http://doi.acm.org/10.1145/3125374.3125375`, `doi:10.1145/3125374.3125375`.

30 David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Not.*, 22(12):227–242, December 1987. URL: `http://doi.acm.org/10.1145/38807.38828`, `doi:10.1145/38807.38828`.

31 Todd L. Veldhuizen. C++ templates are turing complete. Technical report, 2003.

32 M. P. Ward. Language oriented programming. *Software — Concepts and Tools*, 15:147–161, 1995.

33 Xtend — modernized java, November 2017. https://www.eclipse.org/xtend/. URL: `https://www.eclipse.org/xtend/`.