José de Oliveira Guimarães

The Cyan Language Metaobject Protocol

Sorocaba, SP

15 de janeiro de 2023

José de Oliveira Guimarães

The Cyan Language Metaobject Protocol

Tese apresentada na Universidade Federal de São Carlos como parte dos requisitos exigidos para a promoção para Professor Titular

Universidade Federal de São Carlos – UFSCar

Centro de Ciências em Gestão e Tecnologia – CCGT

Sorocaba, SP

15 de janeiro de 2023

Guimarães, José de Oliveira
The Cyan Language Metaobject Protocol/ José de Oliveira Guimarães. – 2023.
239 f. : 30 cm.

Tese – Universidade Federal de São Carlos – UFSCar
Centro de Ciências em Gestão e Tecnologia – CCGT
.

Banca examinadora: , Prof. Dr. «Membro 1», Prof. Dr. «Membro 2»
Bibliografia

1. Protocolo de Metaobjetos. 2. Metaprogramação. 3. Linguagem Cyan. I. Universidade Federal de São Carlos. II. O Protocolo de Metaobjetos de Cyan

Esta tese é dedicada às minhas três meninas: Márcia, Lívia e Carolina

Acknowledgements

Agradeço,

à minha esposa Márcia por todo o apoio em palavras e em tempo durante toda esta pesquisa.

à Lívia e Carolina por, simplesmente, existirem.

Understanding a text is interpreting it. Every text has an interpretation.
Every interpretation has an interpretation.
The question is: is $\aleph_0$
enough for understanding?

Resumo

Metaprogramação é a manipulação de programas por programas. Quando feito em tempo de compilação, metaprogramação permite que código seja gerado e adicionado em um programa. Suas construções podem ser reinterpretadas, mudando a semântica do código. Isto possibilita a geração automática de código repetitivo, suporte a Linguagens Específicas de Domínio e a realização de novas conferências no programa além daquelas já feitas pelo sistema de tipos da linguagem. Contudo, o suporte à metaprogramação pelas linguagens existentes é falha. Cada uma delas possui um ou mais dos seguintes problemas: algumas funcionalidades estão ausentes, a metaprogramação é feita pela manipulação de estruturas de baixo nível, o metaprograma pode alterar código não diretamente ligado ao local onde ele foi chamado, metaprogramas podem ser não determinísticos e não modulares, erros não são apresentados no seu contexto e de maneira amigável e pode-se impedir o compilador de fazer as conferências normais (mudando radicalmente a semântica da linguagem). A linguagem Cyan possui um Protocolo de Metaobjetos (PMO), um tipo de suporte à metaprogramação que utiliza metaobjetos capazes de analisar e alterar o programa. Cyan possui a maioria das funcionalidades de outras linguagens em relação à metaprogramação e inúmeras características únicas. Há vários tipos de metaobjetos, todos eles, exceto um, são variações de uma mesma ideia. Mudanças são feitas localmente em relação aos códigos que ativam os metaobjetos e estes são codificados de maneira modular. Erros são apresentados no contexto em que eles ocorrem com explicações detalhadas. Metaobjetos podem adicionar comportamento e código, nunca remover nada. Finalmente, código é produzido usando strings, não estruturas de baixo nível. O Protocolo de Metaobjetos de Cyan foi utilizado para a construção de metaobjetos em inúmeros domínios, evidenciando a sua adequação à solução de problemas reais.

Palavras-chaves: protocolo de metaobjetos, reflexão computacional, metaprogramação, linguagens de programação, linguagens orientadas à objeto, linguagem Cyan, metaobjeto.

Abstract

Metaprogramming is the handling of programs by programs. When done at compile time, metaprogramming allows code to be generated and added to a program. Its constructs can be reinterpreted, changing the semantics of the code. This enables the automatic generation of repetitive code, support for Domain Specific Languages, and the realization of new checks in the program beyond those already made by the language type system. However, support for metaprogramming by existing languages is flawed. Each of them has one or more of the following problems: some features are missing, metaprogramming is done by manipulation of low-level structures, the metaprogram can change code not directly connected to the location where it was called, metaprograms may be nondeterministic and non-modular, errors are not presented in their context and in a user-friendly way, and one can prevent the compiler from making regular checks (radically changing the semantics of language). The Cyan language has a Metaobject Protocol (PMO), a type of metaprogramming support that uses metaobjects capable of analyzing and altering the program. Cyan has most of the functionalities of other languages in relation to metaprogramming and numerous unique features. There are several kinds of metaobjects, all but one are variations of the same idea. Changes are made locally in relation to the codes that activate the metaobjects and these are coded in a modular way. Errors are presented in the context in which they occur with detailed explanations. Metaobjects may add behavior and code, never remove anything. Finally, code is produced using strings, not low-level structures. The Metaobject Protocol of Cyan was used for the construction of metaobjects in many domains, demonstrating its adequacy to the solution of real problems.

Key-words: metaobject protocol, computational reflection, metaprogramming, programming languages, object-oriented languages, Cyan language, metaobject.

List of Figures

List of Tables

Listings

List of abbreviations and acronyms

AOP	Aspect-Oriented Programming
AST	Abstract Syntax Tree
BSJ	Backstage Java
CLOS	Common Lisp Object System
CTMP	Compile-time Metaprogramming
DSL	Domain Specific Language
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LOP	Language-Oriented Programming
LW	Language Workbench
MOP	Metaobject Protocol
OJ	OpenJava
RTMP	Runtime Metaprogramming
XML	Extensible Markup Language

Contents

1 The Cyan Metaobject Protocol

This chapter describes the original part of this thesis, the Cyan Metaobject Protocol (MOP). The MOP describes the *interactions* between the compiler, the base program and its annotations, the metaprogram, the compiler, and the MOP library. The MOP defines when and which metacode is called for each program annotation. The Cyan compiler is implemented in Java making it easy to build metacode in Java. Since Cyan code is translated into Java code, metacode can also be implemented in Cyan. The compiler is able to use either Java or Cyan as the metaprogramming language.

The next section explains a complete example of a Cyan program that uses metaprogramming. The explanation shows how the MOP works and fixes the terminology used in this chapter. Metacode in Cyan is composed of Java classes and Cyan prototypes that implement interfaces of a *MOP library*. These interfaces are described in section 1.2. The Cyan MOP addresses most of the problems of **??** as is attested in section 1.4. Some shortcomings of MOP are presented in section 1.4.

1.1 A Complete Example Explained

The base program and the metaprogram are linked by syntactic elements in the base program called *annotations* or *metaobject annotations* as shown in Listing 1.1. Prototype `Person` uses three annotations: `property`, `init`, and `compilationInfo`, each one preceded by "`@`". Annotation `init` takes a parameter and is attached to prototype `Person`. Its associated metaobject creates an `init` method, a constructor, to initialize field `name`. Annotation `property` is attached to the declaration of field `name` and its associated metaobject creates `get` and `set` methods for it. Annotation `compilationInfo` takes one

Listing 1.1 – Prototype `Person` that uses metaobject annotations

```
1  package human
2
3  @init(name)
4  object Person
5      @property var String name
6      func test {
7          let Array<String> list = @compilationInfo("field list");
8          list println;
9      }
10 end
```

Listing 1.2 – Prototype `Student`

```
// this is a comment
// the delimiters for 'doc' are {* and *}
// the delimiters for 'replaceCallBy' are {:< and >:}

@doc{* returns the double of the argument *}
@replaceCallBy(once){:< 2*n >:}
func twice: Int n -> Int = n + n;
```

literal string parameter and generates a literal array with the prototype fields.

Annotations can take as parameters one or more of the following literals: basic values (`0`, `2.71`, `'A'`), literal arrays, literal tuples, literal maps, and any combination of these. There may be a text between delimiters after the annotation name (if there is no parameters) or after `)` (if there are parameters). An example is the text

```
2*n
```

between delimiteres `{:<` and `>:}` of Listing 1.2. This text is called *attached text* or *attached DSL*[1] *code*. Annotation `doc`, in this example, also takes an attached text, which is the documentation in english of method `twice:`. There are rules for the creating of delimiters, described by Guimarães (**??**). For short, the characters allowed are:

```
=!?$%&*-+^~/:.\\|([{<>}])
```

And the right delimiter should mirror the left one. In this text, we will use `{*` and `*}` in the remaining examples.

Annotations can be *attached to* declarations or be expressions. In Listing 1.1, `init` and `property` are *attached* to the prototype and to a field and `compilationInfo` is an expression. Annotations `replaceCallBy` and `doc` are attached to method "`twice:`".

Each Cyan *source file* holds a *compilation unit* composed of a single prototype and *import declarations*. Thus, source files are in a one-to-one relationships to program prototypes. The first compilation phase of a Cyan source file is parsing, when the compiler builds an Abstract Syntax Tree (AST). For each annotation, the compiler creates three objects: a *metaobject*, an object of the AST that is private to the compiler, and a wrapper object of the private AST object. The later is a simplified and read-only version of the compiler AST object. Both AST objects keep information on the annotation such as its name, parameters, and attached text.

A *metaobject* is an object of a Cyan prototype or a Java class. We will consider

[1] Domain Specific Language

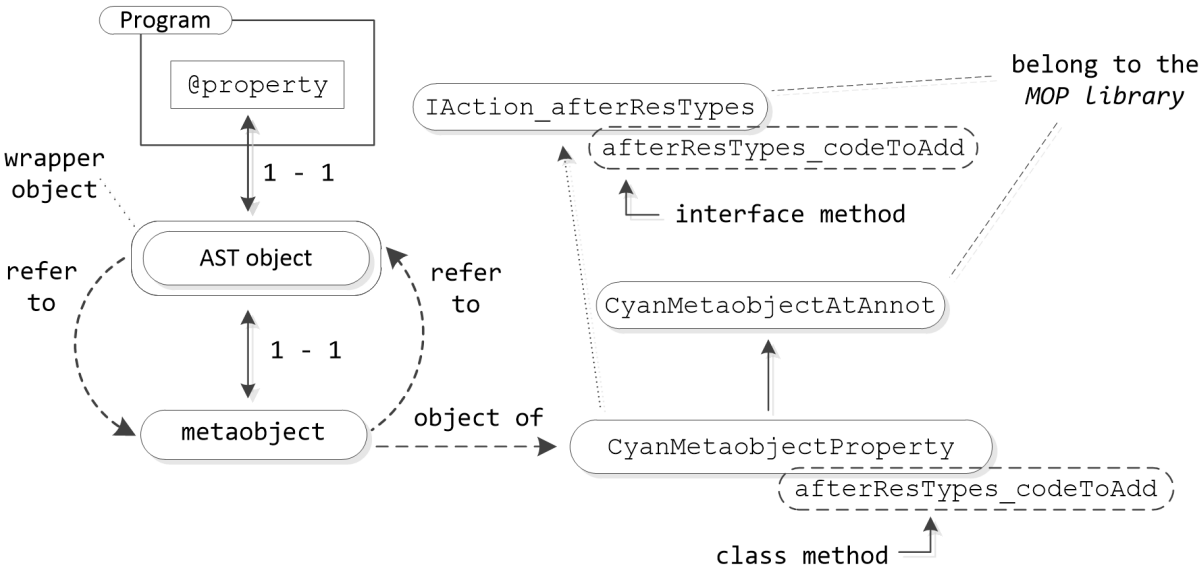

Figure 1 – Relation between metaobjects, annotations, metaobject classes, MOP interfaces, and compilation phases

metaobjects are implemented in Java unless said otherwise. A metaobject class should inherit from class `CyanMetaobjectAtAnnot`.[2] If the metaobject is implemented in Cyan, its prototype should inherit from an superprototype with this same name.

Figure 1 relates all previously cited elements. A program with an annotation `property` appears on the left and above, represented as a rectangle. For each annotation there is an AST object of the compiler wrapped by another AST object (the later is used by the MOP). Both are represented as rounded rectangles in the figure. On the left and bottom, there is the metaobject corresponding to the annotation `property`. There is a one-to-one relationship between metaobjects and AST objects. The figure also shows that the metaobject is an instance of class `CyanMetaobjectProperty` that inherits from `CyanMetaobjectAtAnnot` and implements interface `IAction_afterResTypes`, overriding the interface method `afterResTypes_codeToAdd`. This method generates get and set methods for the `name` field. The binary of class `CyanMetaobjectProperty` is supplied with the Cyan basic libraries. Appendix B presents the complete Java code of the class of metaobject `property2`. This is a simplified version of `property`.

We will use "*metaobject* `property`" for the metaobject associated with the `property` annotation of Listing 1.1. There is no ambiguity because there is only one annotation and, therefore, only one metaobject. Therefore, the annotation name will be used to identify its associated metaobject if no confusion arises.

A *package* in Cyan is a collection of prototypes, each one one in a file within the same package directory. A special package subdirectory contains compiled versions of

[2] "AtAnnot" means an "Annot"ation that starts with "At" (@).

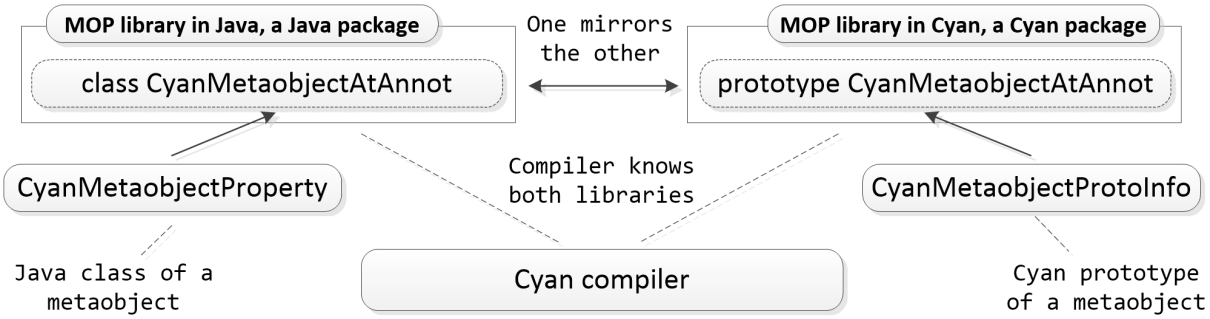

Figure 2 – The relation to the compiler to the MOP libraries

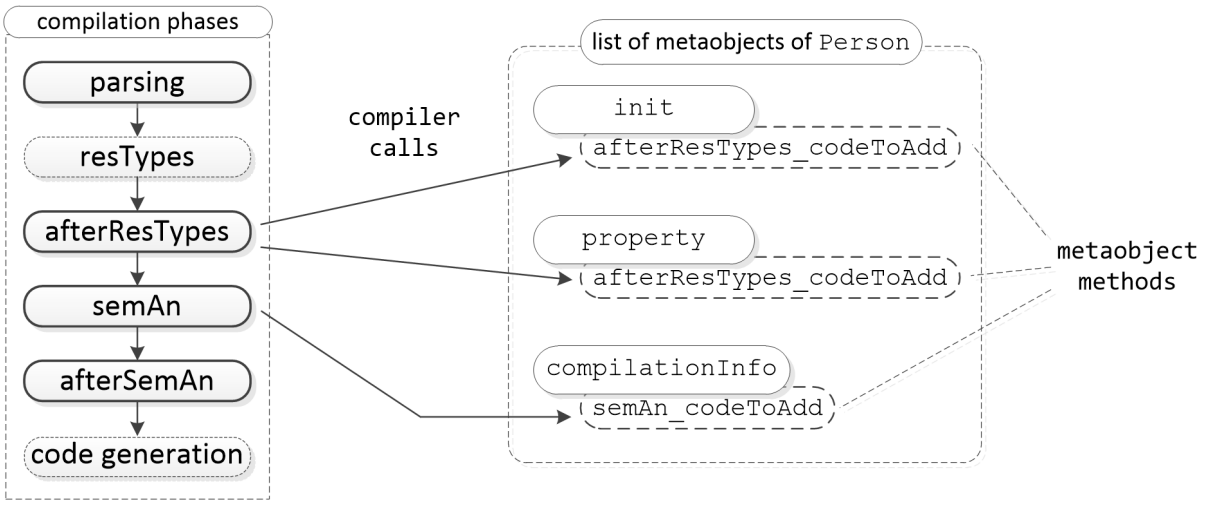

Figure 3 – The compilation phases and their links to methods of metaobjects at compile-time

metaobject classes or prototypes (`.class` files). Importing a package in a compilation unit means loading the package metaobjects. Thus, they can be used in that compilation unit. For each `.class` file of the imported package, the compiler creates an object and calls its `getName` method. The string returned is the annotation (and metaobject) name. Package `cyan.lang` is automatically imported by Cyan compilation units. It holds several largely-used metaobjects, including all metaobjects used in Listing 1.1.

There is a package in Cyan and another in Java with the prototypes and classes used by the MOP. Both packages are called the "*MOP library*". The Cyan package mirrors the Java package, as shown in Figure 2. Classes `CyanMetaobjectAtAnnot` and `IAction_afterResTypes` belong to the Java *MOP library*. The equivalent prototypes, with the same names, belong to the Cyan *MOP library*.

Figure 3 shows, on the left rectangle, the six compilation phases of the Cyan compiler. Each source file goes through all phases. The first phase, parsing, does the

```
1  package main
2
3  object VeryUseful extends Useful implements ILog
4      func factorial: Int n -> Int {
5          // non-recursive, just for fun
6          var Int r = 1;
7          for Int elem in 1..n {
8              r = r*elem
9          }
10         return r
11     }
12     func log: String what {
13         logArray add: what
14     }
15     func getLogArray -> Array<String> = logArray;
16     let Array<String> logArray = Array<String>();
17 end
```

Figure 4 – Type resolution by the Cyan compiler

syntactical analysis and builds the AST. Some Cyan constructs are associated with a type, such as expressions (all of them), fields, local variables, implemented interfaces, superprototypes, and so on. Their AST objects have a `type` field that is set to `null` when the objects are created (the compiler is implemented in Java). All source files of a program are first parsed and, then, they can undergo the remaining compilation phases.

In phase **resTypes**, the `type` field of all AST objects that represent constructs outside method bodies are resolved. Grayed constructs in Figure 4 are outside method bodies associated with a type. The field `type` of their objects are set in phase resTypes. The underlined constructs in this Figure are inside method bodies and are associated with a type. The field `type` of their objects are set in phase semAn.

The compiler calls some metaobject methods in phase **afterResTypes**, *after resolving types*. The method of metaobject `property` that adds get and set methods is called in this phase. Phase semAn is the second part of the semantic analysis (the first is resTypes). It resolves the types of all Cyan constructions that were not treated in the resTypes phase. It also does all remainding checks demanded by the language. Some metaobject methods are called in phase *after semantic analysis*, **afterSemAn**. This phase is only used by the MOP. The last phase is code generation, it is not used by the MOP.

Phases parsing, afterResTypes, semAn, and afterSemAn are used by the Cyan MOP. They are represented in Figure 3 using non-dashed rectangles. Phases afterResTypes and

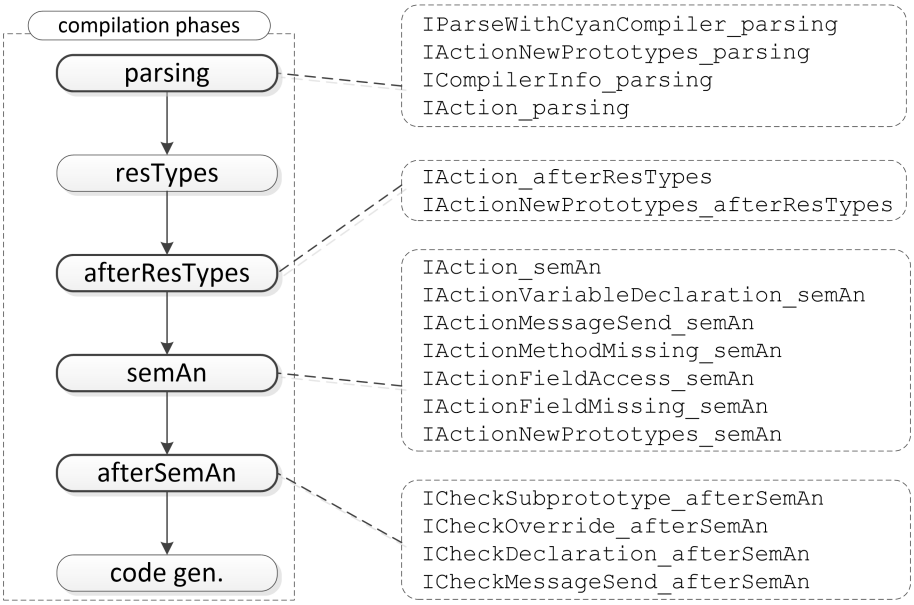

Figure 5 – The compilation phases and their links to the interfaces of the *MOP library*

afterSemAn would not be necessary if the language did not have a MOP. Each compilation phase is associated with several interfaces of the *MOP library* as shown in Figure 5. An interface is associated with exactly one phase and its name ends with its phase name.

The compiler creates, in phase parsing, a metaobject for each annotation. Then, in each phase associated with interfaces, the compiler calls all of the metaobject methods, declared in interfaces of that phase, of all metaobjects of all program prototypes. In the `Person` example of Listing 1.1, on the right, the dashed rectangle shows a list of metaobjects associated with this prototype. There are three metaobjects because there are three annotations associated with prototype `Person`. For this prototype, the compiler calls method `afterResTypes_codeToAdd` of metaobjects `init` and `property` in phase afterResTypes. And calls method `semAn_codeToAdd` of object `compilationInfo` in phase semAn. Method `afterResTypes_codeToAdd` is declared in interface `IAction_afterResTypes` and `semAn_codeToAdd` is declared in interface `IAction_semAn`.

The compiler calls some metaobject methods when some *events* happen. As examples,

a) `IActionMethodMissing_semAn` methods are called when the compiler does not find a method that matches a message passing;

b) `ICheckSubprototype_afterSemAn` methods are called when a prototype is inherited.

Some interface methods are called when declarations or annotations are processed. For example, metaobject methods declared in `IAction_afterResTypes` are called when the annotation is processed in phase afterResTypes.

Listing 1.3 – Prototype `Person` that uses metaobject annotations

```
1  package human
2
3  object Person
4      var String name
5      func test {
6          let Array<String> list = [ "name" ];
7          list println;
8      }
9      func getName -> String = name;
10     func setName: String name { self.name = name }
11     func init: String name {
12         self.name = name
13     }
14 end
```

Metaobjects can generated code, in string format, that is inserted, by the compiler, in the program source code. However, the program is changed only during the current compilation. That is, the original files are not modified. In the example of Listing 1.1, metaobject `property` generates methods `getName` and `setName:` and metaobject `init` generates an `init:` method. The compiler inserts, in the source file and in phase afterResTypes, the generated methods. Then, this source code goes through phases parsing and resTypes again. Phase afterResTypes is skipped so that the metaobject methods associated with this phase are not called again — that prevents an infinite loop. In phase semAn, metaobject `compilationInfo` generates string

```
"[ \"name\" ]"
```

that is inserted in the code. The compiler has to compile the source code again starting with parsing. In this time, phase afterResTypes is skipped and the compiler does not call metaobject methods declared in interfaces associated with phase semAn. That also may prevent infinite loops. Listing 1.3 shows the final code of prototype `Person` with some auxiliary code leaved out. Note that the number of compilation phases a source code undergoes is finite even when metaobjects generate code. At most, a source code goes through phases parsing, resTypes, afterResTypes, parsing, resTypes, semAn, parsing, resTypes, semAn, afterSemAn, and code generation.

Future versions of the compiler will, hopefully, not compile the source code again after metaobjects generate code. The compiler will only compile the generated code and inserts its AST in the prototype AST.

1.2 The Interfaces of the MOP Library

The compiler calls metaobject methods at compilation phase according to the interface the methods were declared. Thus, the interfaces of the *MOP library* are used to direct Cyan metaprogramming. This sections describes these interfaces and how they direct the compilation. The Java *MOP library* is used, although everything described here also applies to the Cyan MOP library.

The remainder of this text uses some terminology that is now described. If an annotation is attached to a prototype or textually inside it, we say that the prototype is the *current prototype* of the annotation. In Listing 1.1, the *current prototype* of all annotations is `Person`. Similarly, the *current compilation unit* of an annotation is the compilation unit in which the annotation is. If an annotation is textually inside a method of attached to a method, we say the method is the *current method* of the annotation. Thus, method `twice:` is the *current method* of annotation `replaceCallBy` of Listing 1.2 and `test` is the *current method* of annotation `compilationInfo` of Listing 1.1. In this last figure, `init` and `property` have no current method.

Cyan methods of the *current prototype* are called *base methods* or, if there is no confusion, just *methods*. Methods of metaobject classes or methods of interfaces of the *MOP library* will be called *metamethods* or just *methods* if no confusion arises.

The goals of a metaobject defines which interfaces its class should implement. If the metaobject should adds fields and methods to the current prototype, its class should implement interface `IAction_afterResTypes`. It the metaobject is an expression, like `compilationInfo` of Listing 1.1, its class should implement interface `IAction_semAn`. And so on. The most important decision relating to a metaobject is taken before the coding of its methods. It is the choice of the interfaces its class will implement. In contrast to this, in some languages the main decisions are taken at metaobject runtime, which is compile-time for the program being compiled.

Metaobject classes inherit from class `CyanMetaobjectAtAnnot` that inherits from `CyanMetaobject`, both from the MOP library. A metaobject method may need to know its annotation, the current prototype, its environment, and so on. A method inherited from `CyanMetaobjectAtAnnot` return the AST object of the annotation (see the right arrow labeled “refer to” in Figure 1). This object holds the annotation parameters, line number, attached DSL code, and the declaration the annotation is attached to (if any). Every method declared in any MOP interface takes a parameter with information on the annotation environment: the current method, the current prototype, the current compilation unit, the statements of methods of the current prototype, and so on. The type and name of this parameter varies according with the compilation phase and method name. The type and name can be `WrEnv env`, `ICompiler_semAn compiler_semAn`, and

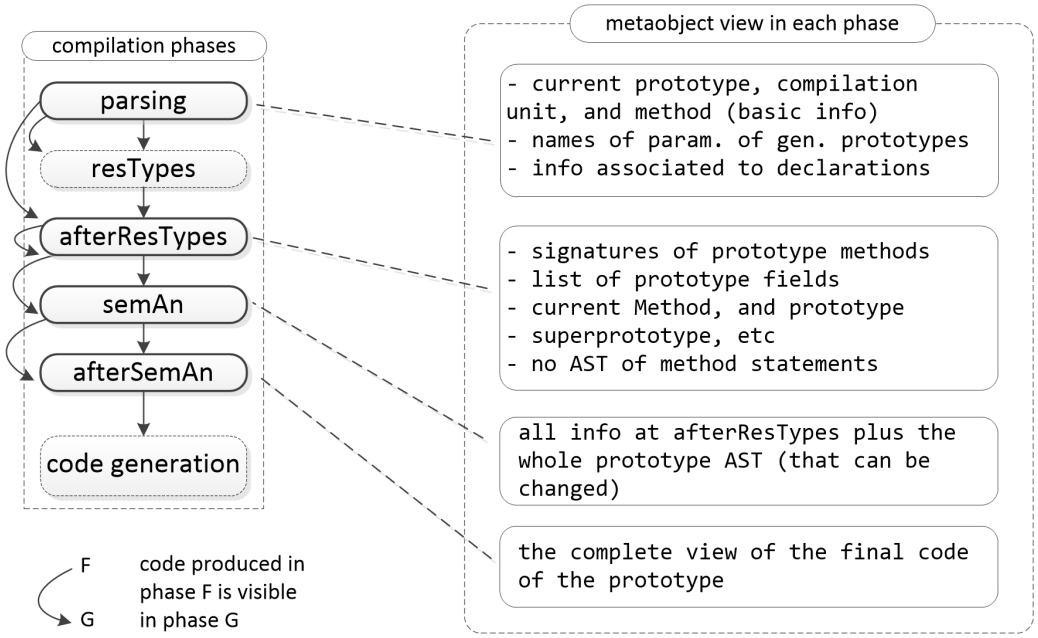

Figure 6 – The information available in each compilation phase

the like. This parameter restricts the information available in each compilation phase. For example, the AST object representing statement methods is not available in phase afterResTypes. It is available in phase `IAction_semAn`. The most important information a metaobject has in each compilation phase is shown in figure Figure 6.

The following sections describe the interfaces of the *MOP library* that can be implemented by metaobject prototypes.

1.2.1 Interfaces for Creation of New Prototypes

New prototypes can be created in phases parsing, afterResTypes, and semAn. For each phase, there is an interface whose name is composed of `IActionNewPrototypes_` and the phase name. The single method of each interface, when overridden in the metaobject class, should return the prototype name and its code as a tuple. Both as strings. The new prototype's package must be the package of the current compilation unit. That is, an metaobject can only create a prototype in the package of its associated annotation.

Why do three interfaces for prototype creation are needed? Why not just one? There are two reasons: (a) the semantic analysis is made in phases, first resTypes (outside method statements) and then semAn (method statements); (b) The latter phases provide more information than the former ones. Therefore, a prototype used, for example, as the type of a method parameter should exist in phase resTypes. It either exists in the original program or is created by a metaobject in phase parsing. Consider a prototype that is the type of a local variable whose declaration does exist at the start of phase semAn. That

Listing 1.4 – Prototype `Person` that uses metaobject annotations

```
1     // return the factorial of 10
2   func fat10 -> Int =
3         // calculates at compile-time the factorial of 10
4       @eval("cyan.lang", "Int"){*
5           var r = 2;
6           for n in 3..10 {
7               r = r*n
8           }
9           return r
10      *};
```

is, the local variable declaration was not added by metaobjects acting in phase semAn. This prototype should exist at the start of phase semAn. It either exists in the original program or it was created in phases parsing or afterResTypes.

A local variable declaration could have been created by metaobjects acting in phase semAn. If the variable type (a prototype) did not exist before phase semAn, it can be created in this phase. Ideally, a prototype is created as late as possible because more information is available in later phases. However, it may be necessary to create it earlier because it is used in early compilation phases.

1.2.2 Interfaces of Phase parsing

Annotations may have an *attached DSL code*, usually given between `{*` and `*}` as `eval` shown in Listing 1.4. Interface `IParseWithCyanCompiler_parsing` has a method with a parameter that is a restricted view of the Cyan compiler. The parameter has methods for lexical analysis and parsing of Cyan types, expressions, and statements. It is the ideal tool to use when the attached DSL code is similar to Cyan. Each method returns an AST object. In the above example, the AST objects created during parsing are later used, in phase semAn, for interpreting the code at compile-time.

Interface `ICompilerInfo_parsing` is used to pass information, like documentation, from the annotations to declarations. As an example, metaobject `doc`, cited in section 1.1, uses this interface for adding documentation to packages, prototypes, methods, and fields. Interface `IAction_parsing` declares a method to add code after the annotation (it will be removed in the next MOP version). Phase parsing has also interfaces for generating code and passing information to declarations.

1.2.3 Interfaces of Phase afterResTypes

Interface `IAction_afterResTypes` declares four methods. Method

`afterResTypes_beforeMethodCodeList`

returns a list of 3-tuples, each one composed of a method name (a string), a list of statements (another string), and a boolean value. The list of statements is added at the beginning of the method of the current prototype given in the first tuple element. If the boolean value is `true`, the metaobject is requesting exclusive rights to add statements at the beginning of the method. Therefore, if two metaobjects try to add statements to the same method and at least one of them request exclusive rights, the compiler issues an error.

Method

`afterResTypes_renameMethod`

is used for renaming methods. However, a method with the old method name should be added to the prototype. This prevents difficult-to-understand compilation errors. As an example, the developer could see, in the IDE, a base method but the compiler would issue the error "method was not found" in a message passing that should call that method.

A *method signature* is the method declaration without its body or the expression assigned to it. Parameter names are optional. The *signature of a field* is composed of `var` or `let`, the type, and the field name.

```
var Int count = 0;
let String name;
func getCount -> Int = count;
func add: String
      at: Int, Int
      doc: String {
      ...
}
```

The signatures of the methods and fields given above are

```
var Int count
let String name
func getCount -> Int
func add: String
      at: Int, Int
      doc: String
```

Method `afterResTypes_codeToAdd:` of interface `IAction_afterResTypes` is used to add fields and methods to the current prototype. It returns a tuple composed of two

elements: (a) base fields and methods and (b) the signatures of these fields and methods separated by ";"

```
Tuple2<StringBuffer, String>
afterResTypes_codeToAdd(
        ICompiler_afterResTypes compiler,
        List<Tuple2<WrAnnotation,
            List<ISlotSignature>>> infoList )
```

Parameter `compiler` is a restricted version of the Cyan compiler. It supplies some useful methods such as to return the fields and methods of the current prototype, the AST object of the current prototype, and so on. `infoList`, the second parameter, is a list of tuples, each one composed of the AST object of an annotation and a list of slot signatures. A *slot* is either a method or a field. The compiler calls method

```
afterResTypes_codeToAdd
```

passing, as the second argument, an empty list in the first time this method is called for each metaobject of a prototype. Parameter `infoList` is only useful when method

```
boolean runUntilFixedPoint()
```

of interface `IAction_afterResTypes` returns `true`. In this case, it is assumed that the code generated by other metaobjects of the same prototype should be known by the metaobject. Let us explain that using an example.

Metaobject `addFieldInfo` is used for demonstration only. It adds, to the current prototype, a field whose name is the first annotation parameter initialized with the number of the fields of the current prototype. The class of this metaobject implements interface `IAction_afterResTypes`.

```
1 @addFieldInfo(fieldNum)
2 @addFieldInfo(numOfFields)
3 object TestField
4     var Int count = 1;
5     func getCount -> Int = one;
6 end
```

In this example, if method `runUntilFixedPoint` of the class of `addFieldInfo` returns `false`, the `afterResTypes_codeToAdd` methods of the metaobjects associated with the annotations of lines 1 and 2 would generate

```
let Int fieldNum = 2;
let Int numOfFields = 2;
```

The compiler calls methods `afterResTypes_codeToAdd` passing, as the second argument, an empty list because `runUntilFixedPoint` returns `false`. Consequently, one metaobject

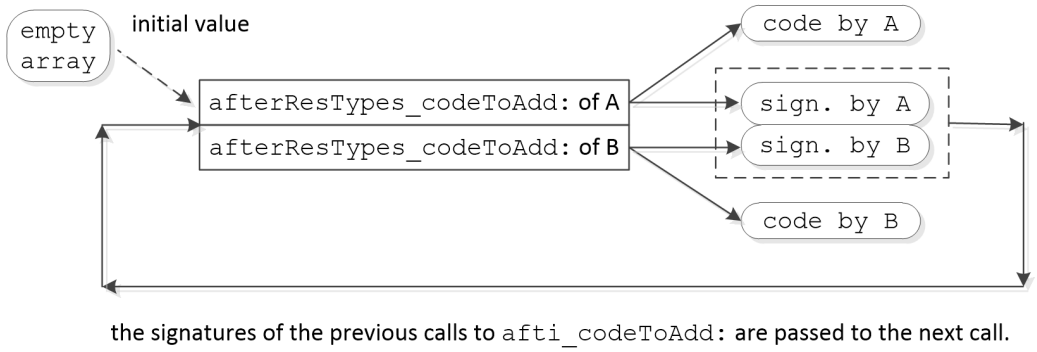

Figure 7 – Flow of control in algorithm **FixMeta** with two metaobjects

does not view the base fields added by the other. Both view the original prototype, which has a single field.

If `runUntilFixedPoint` returns `true`, method `afterResTypes_codeToAdd:` is called multiple times according to the algorithm `FixMeta` of Listing 1.5. Each metaobject knows the fields added by the other in the second time `afterResTypes_codeToAdd:` is called.

Algorithm FixMeta takes two lists as input, `fullList` and `roundList`. The former contains all metaobjects of the current prototype whose classes implement interface `IAction_afterResTypes`. The latter is a sublist of `fullList` containing the metaobjects whose methods `runUntilFixedPoint` return `true`. The `for` statement of lines 3-9 collects all *base method and field* signatures generated by all metaobjects into a list `infoList`. In lines 12-30, the algorithm makes *rounds* of calls. Each *round* is composed of calls to method `afterResTypes_codeToAdd:` of all metaobjects of `roundList`. These calls are made in lines 16-26. After each round, the algorithm checks, in line 12, if all metaobjects produced the same code as before. It they did, the loop ends. If the number of rounds is greater than the default number of rounds allowed, five, the algorithm issues an error (line 29). The maximum number of rounds can be changed by a compiler option.

Figure 7 shows the flow of control of algorithm `FixMeta` when used with two metaobjects. In the figure center, the two rectangles represent the calls to the `afterResTypes_codeToAdd:` methods. Each one produces code of fields and methods (e.g. "`code by A`") and the signatures of these fields and methods (e.g. "`sign. by A`"). The arrows show that the signatures are used as input to calls to method `afterResTypes_codeToAdd:` in the next round.

Let us return to the `addFieldInfo` example. Method `runUntilFixedPoint` of the metaobjects returns `true`. Therefore, there is a new round of calls to method

`afterResTypes_codeToAdd:`

of the two `addFieldInfo` metaobjects. In this second round, the second method parameter,

Listing 1.5 – Algorithm `FixMeta`

```
1  Algorithm FixMeta(fullList, roundList)
2    infoList = empty list
3    for every metaobject in fullList {
4      call method afterResTypes_codeToAdd of the metaobject
5        passing an empty array as the second parameter.
6        This method returns a tuple. Add the slot
7        signatures of the second tuple element to
8        list infoList.
9    }
10   somethingChanged = true;
11   count = 1;
12   while somethingChanged {
13     newInfoList = empty list
14     // each 'while' loop is a round
15     somethingChanged = false;
16     for every metaobject in roundList {
17       call method afterResTypes_codeToAdd of the metaobject
18          passing infoList as the second parameter.
19          This method returns a tuple. Add the slot
20          signatures of the second tuple element to a
21          list newInfoList.
22       if the code produced by this call are
23       different from the code produced by the
24       same metaobject in the previous round, stored
25       in infoList, set somethingChanged to true
26     }
27     infoList = newInfoList;
28     ++count;
29     if count > maxNumRoundsFixMetaDefaultValue { error; }
30   }
```

`infoList`, refers to a list of tuples. Each tuple is composed of the AST object of an annotation that produced the list of base field and method signatures of the second tuple element. The first tuple contains a reference to the AST object of the annotation of line 1 of the example and a list containg the signatures of fields `fieldNum` and `numOfFields`. The second contains a reference to the annotation of line 2 and the same signature list as the first tuple.

In this second round of calls, both metaobjects can adjust their generated code to take into consideration the code produced by other metaobjects. The real number of fields, assigned to the generated fields, is the original number of prototype fields (1) plus the number of fields generated by all metaobjects, 2. Therefore, the code generated by the metaobjects is "`let Int fieldNum = 2;`" and "`let Int numOfFields = 2;`". In the third round, the generated code does not change and the algorithm ends. Even if just one

metaobject returns a different code from the previous round, the algorithm goes through a new round of calls with all metaobjects of `roundList`.

The Cyan compiler checks whether the elements of the tuple returned by `afterResTypes_codeToAdd:` match. Thus, the compiler checks if the base fields and methods of the first tuple element are in the second tuple element and vice-versa. Method `afterResTypes_codeToAdd:` should return a tuple with empty strings, if used only for checks.

1.2.4 Interfaces of Phase semAn

Interface `IAction_semAn` has a single method that returns a string. This method may be used for checks (if it returns `null` or the empty string) or to add code after the annotation. Some annotations are expressions, like `compilationInfo` of Listing 1.1 or `eval` of Listing 1.4. Their metaobject classes should implement interface `IAction_semAn`.

Annotations associated with metaobject prototypes implementing interface `IActionVariableDeclaration_semAn` should be attached to local variable declarations. The single interface method adds code after the variable declaration and has access to the variable name, type, and expression used to initialize the variable (if any).

Interface `IActionMessageSend_semAn` is implemented by metaobject classes for intercepting message passings. The associated annotations should be attached to base methods. Suppose a metaobject `M` is associated with an annotation attached to method `m`. As a result, `M` can intercept message passings that could, potentially, call `m`. Metaobject `M` may check the message arguments, at compile-time, and replace the message passing by another expression. For each message passing, more than one method may be called and there may exist more than one annotation attached to every method. Therefore, we need rules for defining which metaobject replaces the message passing by an expression.

During phase semAn, the compiler collects the base methods that could be called for every message passing. If the message receiver is `T`, the compiler collects the `T` base methods that could be called. Then, it puts in a list the metaobjects associated with these *base methods* that implement interface `IActionMessageSend_semAn`. The *metamethods* of these metaobjects, declared in this interface, are called. There are three possibilities in relation to the number of metaobjects that return non-`null` or non-empty strings:

1. more than one, which is ambiguous. The compiler issues an error;
2. exactly one. The compiler replaces the message passing by the returned code;
3. none. The compiler searchs for methods that match the message passing in the superprototypes and superinterfaces of `T` (in this order).

For interface `IActionMessageSend_semAn`, the textual annotation order in a prototype is not important because at most one metaobject per prototype is allowed to return a non-`null` or non-empty string.

The metaobject class of `replaceCallBy`, shown in Listing 1.2, implements this interface. The metaobject replaces the message passing "`t twice: 4 + 1`" by

```
{ var tmp343 = 4 + 1; ^2*tmp343; } eval
```

`tmp343` is a temporary variable name.

Interface `IActionMethodMissing_semAn` allows the introduction of *virtual methods* into prototypes. A *virtual method* does not exist but its existence is simulated. In this case, a metaobject replaces the message passing that would call the method by an expression. As an example, a metaobject could read a file with thousands of color names, associate a number to all of them, and simulate the existence of methods with the color names. The metaobject replaces a virtual method call by the number associated to the color. Metaobject `grammarMethod` described in section 2.1 implements interface `IActionMethodMissing_semAn` to simulate the existence of a method whose keywords are specified using a regular expression. It makes it easy to create Domain Specific Languages using only regular Cyan message passings.

Let us explain how the compiler uses interface `IActionMethodMissing_semAn`. When there is no matching method for a message passing, the compiler puts in a list `moList` all metaobjects implementing interface `IActionMethodMissing_semAn` and whose annotations are in prototype `T`, the type of the message passing receiver. Then, a metamethod of each metaobject of `moList` is called. This is the metamethod declared in the metaobject that overrides a method of interface `IActionMethodMissing_semAn`. This interface declares two methods: one for unary and another for keyword message passings.

The value returned by each metamethod of metaobjects of `moList` can be a `null`, an empty string (with code), or otherwise. There are three possibilities based on the number of metaobjects' methods that return non-empty strings:

1. more than one. The compiler issues an error;
2. exactly one. The returned code replaces the message passing. This replacement is visible in the next compilation phase, afterSemAn;
3. none. The same algorithm is applied to the superprototype of `T` (implemented interfaces are not taken into account).

Interface `IActionFieldAccess_semAn` is used for intercepting field access. It declares two methods: one is called when the field is retrieved and the other called when the

field is in the left-hand side of an assignment. The compiler replaces the get or set of the field by the code returned by these metamethods. Annotations associated to metaobject classes that implements this interface can only be attached to prototype fields.

Several annotations whose metaobject classes implement `IActionFieldAccess_semAn` may be attached to a single field. When the compiler finds a get or set of the field, it calls the appropriate metamethods of the metaobjects associated to the annotations. If more than one metamethod returns a non-`null` and non-empty string, the compiler issues an error.

Interface `IActionFieldMissing_semAn` is used for introducing *virtual fields* to prototypes. There are two declared methods in the interface: one called when a non-existing field is get and the other when it is set. Annotations associated to metaobjects whose classes implement this interface should be attached to prototypes. The compiler issues an error if a field is accessed and more than one metaobject is able to replace this access by code.

When the compiler finds a get or set of a field that does not exist, it calls the appropriate metamethods of all metaobjects that could handle this event. These metaobjects are those whose annotations are attached to the current prototype and whose classes implement interface `IActionFieldMissing_semAn`. If more than one metaobject method return a non-empty string, the compiler issues an error.

In phase semAn, the Cyan compiler processes the AST in the textual order of the statements' declarations. That is, the compiler resolve types and do checks in textual order. Therefore, a metaobject `M2` whose associated annotation `a2` appears after annotation `a1` knows the types associated to expressions between `a1` and `a2`. The metaobject `M1` associated to `a1` does not.

The term "AST" used in this paper refers, unless stated otherwise, to the wrapped version of the real Abstract Syntax Tree, which is used by the compiler. The former tree is read-only and contains only methods that return information on the code. It is created by on-demand, lazily. Metaobject metamethods may get AST objects using several mechanisms: accessing parameters, calling parameter methods, and calling methods of the metaobject itself. AST objects have information on the current prototype, method, etc. The *Visitor* Design Pattern (GAMMA et al., 1995) may be used for visiting AST objects.

Method `replaceStatementByCode` is declared in class

`CyanMetaobjectAtAnnot`

which is inherited by every metaobject class whose annotations start with `@` (all annotations seen till now). This metamethod is used for replacing a statement or expression by a code given as string. The statement is given as an AST object. Therefore, any metaobject method can, in phase semAn, replace any statement of the *current prototype* by any other

code. Security mechanisms do not allow metaobjects of one prototype access statements of another prototype.

Metaobject `shout`, used for demonstration only, uses the visitor methods of the AST for visiting all expressions of the current method. It replaces all strings by the equivalent ones in uppercase. Method `semAn_codeToAdd` of its class follows.

```
@Override
public StringBuffer semAn_codeToAdd(ICompiler_semAn compiler_semAn)
{
      // 'annot' is the AST object of the annotation
    final WrAnnotationAt annot = this.getAnnotation();
      // the annotation is attached to method 'dec'
    final WrMethodDec dec = (WrMethodDec ) annot.getDeclaration();

      // 'accept' is the AST method implementing the
      // visitor pattern
    dec.accept( new WrASTVisitor() {
          // only literal strings are visited
          // literal strings in the Cyan code are
          // represented by WrExprLiteralString AST objects
        @Override
        public void visit(WrExprLiteralString node, WrEnv env) {
            final StringBuffer strUpper = new StringBuffer();
            final StringBuffer str = node.getStringJavaValue();
              // convert to uppercase
            for (int i = 0; i < str.length(); ++i) {
                strUpper.append(Character.toUpperCase(str.charAt(i)))
            }
            replaceStatementByCode(node,
                    strUpper, node.getType(), env);
        }
    }, compiler_semAn.getEnv());

    return null;
}
```

1.2.5 Interfaces of Phase afterSemAn

From this compilation phase onwards, metaobjects cannot change the code. Therefore, checks made in this phase will not be invalidated by code changes made afterwards.

An annotation of a metaobject whose class implements interface ICheckSubprototype_afterSe should be attached to a prototype.

```
interface ICheckSubprototype_afterSemAn extends
            ICheck_afterResTypes_afterSemAn, IStayPrototypeInterface
    void afterSemAn_checkSubprototype(
            ICompiler_semAn compiler_semAn, WrPrototype subPrototype)
}
```

The single interface method, defined in the metaobject, is called when the prototype is inherited, even if indirectly.[3] The second metamethod parameter is the AST object representing the subprototype that inherits the prototype with the annotation. Therefore, the metaobject can check the subprototype.

Interface ICheckSubprototype_afterSemAn is used for checks related to inheritance:

a) check whether the subprototype defines either none or all methods of a set of methods;

b) check if a prototype P inherits from a given superprototype whenever P implements an interface II. This is a language feature of language Hack (HACK, 2020). The check would be made by a metaobject associated to an annotation attached to II;

c) check if the interrelationships among the methods of a prototype are kept in the subprototypes. The pattern of calls among the methods of a prototype should be documented if it is intended to be inherited (BLOCH, 2018). Since a subprototype needs to know this call pattern, inheritance violates encapsulation (SNYDER, 1986). A metaobject whose annotation is attached to the superprototype can check if the subprototype follows the method call patterns expected by the superprototype.

An annotation of a metaobject whose class implements interface ICheckOverride_afterSemAn should be attached to a base method.

```
interface ICheckOverride_afterSemAn extends
            ICheck_afterResTypes_afterSemAn,
            IStayPrototypeInterface {

    void afterSemAn_checkOverride(
            ICompiler_semAn compiler, WrMethodDec method);
}
```

[3] If prototype C inherits from B that inherits from A, thus C indirectly inherits from A.

The compiler calls the single interface method, defined in a metaobject, whenever the base method is overridden in a subprototype. The second metamethod parameter is the AST object of the overridden base method.

An annotation of a metaobject whose class implements interface `ICheckDeclaration_afterSemAn` should be attached to a declaration, which may be a local variable, field, method, or prototype.

```
interface ICheckDeclaration_afterSemAn extends
            ICheck_afterResTypes_afterSemAn {
    void afterSemAn_checkDeclaration(ICompiler_semAn compiler);
}
```

The declaration can be obtained from parameter `compiler`.

Message passings should not be checked by metaobjects implementing interface `IActionMessageSend_semAn` of phase semAn. This is because, in this same phase, metaobjects may add new code which may invalidate the checks. Message passings should be checked by metaobjects whose classes implement interface `ICheckMessageSend_afterSemAn`. The compiler calls the methods defined in this interface as described for interface `IActionMessageSend_semAn` with one difference: metamethods of all metaobjects (whose classes implement this interface) are called. Therefore, the compiler calls all metamethods of annotations associated to superprototypes.

1.2.6 Interface for Metaobject Communication at Compile-Time

Metaobjects of the same prototype whose prototypes implement interface `ICommunicateInPrototype_afterResTypes_semAn_afterSemAn` can communicate before phases afterResTypes, semAn, and afterSemAn. When analyzing a prototype and before any of these phases, the compiler collects in a list all metaobjects whose classes or prototypes implement this interface. Then, by calling a metaobject method, overridden from the interface, it collects the objects each of these metaobjects want to share. After that, the compiler calls another metaobject method with the list of shared objects.

1.2.7 Interface for Communicate Compiler Errors

The annotations in the program being compiled can inform the compiler that it has errors. The class associated with the annotation has to implement interface `IInformCompilationError`. If the compiler does not sign an error in the line pointed out by the annotation, the compiler issues an error saying that iteself has an error.

1.3 Other Metaobject Kinds

This section describes metaobjects whose annotations are of a different kind than those already presented. These metaobjects should implement interfaces or inherit from classes not yet described. They have all or most of the power of regular metaobjects.

1.3.1 Annotations to the Project

A project file, detailed in section A.9, contains a code of a DSL called Pyan for describing a Cyan program. It should contain a `program` keyword followed, optionally, by package declarations. It is assumed that all directories of the project file directory contain program packages unless they start with "`--`". A package can also be given explicitly. We repeat here an example of section A.9.

```
program at "C:\Dropbox\Cyan\cyanTests\tese"
    main main.Program  // the main prototype
    package main at "C:\Dropbox\Cyan\cyanTests\tese\main"
    package cap.dynamic at "C:\Dropbox\Cyan\cyanTests\tese\cap\dynamic"
```

Annotations may be attached to the program and packages in Pyan code.

```
@setVariable(testOverride, true)
program
    @checkStyle
    package main at "C:\Dropbox\Cyan\cyanTests\simple\main"
    package cyan.math at "C:\Dropbox\Cyan\lib\cyan\math"
```

Annotation `setVariable` associates an identifier or string to any literal value. In this example, it associates `testOverride` to `true`.

Annotation `checkStyle` may be attached to several kinds of declarations, including prototypes. `checkStyle` does a raw style checking considering the identifiers only, it is a demonstration metaobject. In this example, the annotation is attached to package `main`.

The Cyan compiler, Saci, is also responsible for compiling the project file. It parses and builds an AST for the Pyan code before the start of the Cyan compilation. The parsing phase is called `dpp` for *During Pyan Parsing*. Interface `IAction_dpp` should be implemented by classes of metaobjects that should act in phase dpp.

```
public interface IAction_dpp {

    void dpp_action(ICompiler_dpp project);
```

```
}
```

Interface `ICompiler_dpp` does not currently declare any methods by itself although it inherits from `IAbstractCyanCompiler` (see section E.2).

Metaobjects `setVariable` and `checkStyle` belong to different kinds of metaobjects. The class of the `setVariable` metaobject implements `ICompiler_dpp` and method `dpp_action` is called during Pyan code parsing.

The class of the `checkStyle` metaobject does not implement `ICompiler_dpp` and therefore its methods are not called during Pyan parsing. Annotations of this metaobject can be attached to prototypes and for this reason its annotations can be attached to the program or to packages. If `checkStyle` is attached to a package, the result is the same as to attach it to every prototype of the package. If annotation `checkStyle` is attached to the program, the result is the same as to attach it to every prototype of the program.

1.3.2 Annotations for Creating Prototypes Before Compilation

Cyan programs can be composed of source files of any language in addition to those containing Cyan prototypes. The only requirement is that there is a metaobject able to convert the source file into one or more prototypes. A file whose name does end with "`.cyan`" should be put in directory `--dsl` of a Cyan package. This is called a `DSL` directory.

For each file with a extension `ext` in a `DSL` directory, the compiler looks for a metaobject whose class inherits from

`CyanMetaobjectFromDSL_toPrototype`

and whose method `getName` returns `"ext"`. The extension name is user-defined, it can be anyone. The metaobject class should override method

```
List<Tuple3<String, String, char []>>
parsing_NewPrototype(ICompiler_dsl compiler_dsl)
```

inherited from its superclass. When the compiler finds a file ending with extension `ext` in a directory DSL of a package, it looks for a metaobject whose name is `ext` and calls the above method. This is made before phase parsing of the compilation to produce one or more Cyan prototypes. This method returns an array of tuples, each one composed of a prototype name, the file name in which this prototype should be, and the prototype code. A package can only create new prototypes inside itself. Therefore, there is no package name in the returned tuples.

The Pyan source file can import packages to the whole program or only to a package. This is just importation of metaobjects, not prototypes. In Listing 1.6, the metaobjects

Listing 1.6 – Pyan file with 'import' declarations

```
import cyanHelper at "C:\Dropbox\tests\cyanHelper"
program
    import stylePack
    @myCheckStyle
    package main
    package company
```

of package `cyanHelper` are visible to the whole program during Pyan processing and `stylePack` is visible only to package `main`. Annotations of a metaobject are allowed in any place it is visible. A visible metaobject can be used to process DSL files. Hence, if method `getName` of a metaobject of package `cyanHelper` returns `"python"`, this metaobject will be used to process a file with this extension in a DSL directory of package `company`. And it is legal to attach annotation `myCheckStyle` to package `main` assuming there is a metaobject in package `stylePack` whose method `getName` returns `myCheckStyle`.

If the name of a file in directory `DSL` starts with an uppercase letter, method `dpa_NewPrototype` should return exactly one tuple. The prototype name should be exactly equal to the file name unless the prototype is generic. In this last case, the file name should match the prototype name according to the rules of file names for generic prototypes.

If the name of a file in directory `DSL` starts with lowercase letter, method

```
parsing_NewPrototype
```

should return one or more tuples. The names of the prototypes of each tuple need not be related to the name of the file of directory `DSL`.

`CyanMetaobjectFromDSL_toPrototype` is one of the Cyan ways of supporting Language Oriented Programming, a paradigm first proposed by Ward (WARD, 1995). The idea is that a software system encompasses several small domains and there should be a language for each of them. This paradigm needs easy-to-use tools for building other tools for languages such as compilers, debuggers, and IDEs. These tools are called *Language Workbenches* (ERDWEG et al., 2015) and, from a language description, are able to generate tools for the language, even including full IDEs. Class `CyanMetaobjectFromDSL_toPrototype` is small step in supporting Language Oriented Programming. Its importance is that the compiler of the main language, Cyan, is aware of the DSLs and supports them.

1.3.3 Literal Numbers as Annotations

Cyan supports literal numbers ending with either the name of the type or the first uppercase letter of the name:

```
var size = 100Int;
var count = 10I;
```

```
let pi = 3.141592Double;
let e = 2.7182818284D;
```

The language also supports literal numbers that are annotations. There are metaobjects associated with them with most of the power of regular metaobjects. For example, there is an annotation `bin` for binary numbers:

```
assert 101bin == 5;
assert 111bin == 7;
```

The metaobject class of `bin` is in package `cyan.lang` and therefore it is always imported.

A number ending with an identifier is considered a metaobject annotation if the identifier is not `Byte`, `B`, `Short`, `S`, `Int`, `I`, `Long`, `L`, `Float`, `F`, `Double` or `D`.

The class of a metaobject for a literal number should inherit from class

`CyanMetaobjectNumber`

The subclass of this class, `CyanMetaobjectLiteralObject` implements interface

`IAction_semAn`

The code generation is made by method `semAn_codeToAdd` declared in interface `IAction_semAn`.

Method `semAn_codeToAdd` of a metaobject of a user-defined literal number should produce an expression. Classes of metaobjects of literal numbers can implement interfaces of the MOP library that are compatible with `IAction_semAn` and that do not act in phase afterSemAn.

1.3.4 Literal Strings as Annotations

Literal strings preceded by an identifier are considered metaobject annotations. They may be used for DSL code or special strings. Package `cyan.lang` offers two metaobject classes, one for strings without escape characters and the other for regular expressions. Package `cyan.util` offers support for XML strings.

```
  // cyan.util was imported
let mydog = xml"""
        <dog>
            <name> Meg </name>
            <age> 3 </age>
        </dog>
    """;
  // unescaped strings are prefixed with 'n' or 'N'
assert n"\n\r" size == 4;
  // r"str" is translated into RegExpr("str")
```

```
assert "six" ~= r".*i.*";
```

The classes of metaobjects of literal strings must inherit from class `CyanMetaobjectLiteralString` that implements interface

`IAction_semAn`

The constructor of `CyanMetaobjectLiteralString` takes an array of `String` elements, the prefixes allowed for the literal string.

As with literal numbers, method `semAn_codeToAdd` of a metaobject for user-defined literal strings should generate an expression. The metaobject class can implement interfaces compatible with `IAction_semAn`. The class should not implement interfaces of phase afterSemAn.

1.3.5 Macros as Annotations

Cyan supports macros which are considered special metaobjects. A macro call is considered an annotation whose associated metaobject class should inherit from class `CyanMetaobjectMacro` of the MOP library. This class implements interfaces `IParseMacro_parsing` and `IAction_semAn`. A macro metaobject defines keywords that can start a macro and keywords that can be used in it (they are unrelated to the Cyan keywords). A metaobject method, overridden from `IParseMacro_parsing`, is responsible for parsing the macro call and building the AST. It is expected that this AST is used by the metaobject method that overrides the only method of `IAction_semAn`.

A macro call in Cyan always starts with a *macro keyword*. After it, the macro may use any syntax, although it necessarily must use at least the Cyan compiler for lexical analysis. `assert` is an example of a Cyan macro. Its full code is in section E.5. A code

```
assert n"\n\r" size == 4;
```

is replaced by something like the following.

```
if ! (n"\n\r" size == 4) {
    "Assert failed in line 58 of prototype 'main.Test'" println;
    "Assert expression: 'n\"\n\r\" size == 4'" println;
}
```

Unlike other metaobject annotations, the original annotation, the macro call, is deleted from the source code before the new code is inserted. Therefore a macro class cannot implement any interfaces that act in compilation phase afterSemAn.

Cyan offers only low-level support for macros, even lower than that of Lisp. However, a future work will add a small change in the MOP that will allow other metaobjects to produce macro metaobjects at compile-time. With this change, an annotation in Cyan

of a would-to-do metaobject class `defmacro` could take a DSL that specifies the macro, much like the macro specification in language Rust (Rust. . . , 2018), and produce a new macro metaobject.

1.3.6 Annotations Attached to Types

A pluggable type system is a set of algorithms for type checking that may be incorporated into the compiler for the whole or part of a compilation. It is a *pluggable* system. In Cyan, a type in the source code may have an attached annotation that demands additional checks in the use of the type in that specific place. The regular type checking is not changed but additional checkings are allowed by the type annotation. Therefore, the type of a variable may be "`T@plug`" to mean that it will undergo extra checks by metaobject `plug`. The effect is local, only this variable is affected. Some examples will make that clear.

Line 1 of the following code declares a variable whose type is `Char` with an attached annotation `letter` (declared in package `cyan.lang`). The associated metaobject checks if the values assigned to the variable are letters. When a literal char is assigned to the variable, the check is made at compile-time. If an expression is assigned, the check is postponed to runtime. An exception is thrown if the check fails. Metaobject `range` assures that the variable is between the first and second values of the annotation parameters. A `String` variable may only contain valid regular expressions if `@regex` is attached to its type. There is a compile-time error if its parameter is not a regular expression, as would happen in line 17[4] if uncommented. If an expression that does not match the regular expression is assigned to the variable, there would be a runtime error, an exception is thrown.

```
1   var Char@letter ch = 'a';
2   ch = 'b';
3
4   var Int@range(1, 12) month = 12;
5   // if uncommented, compile-time error
6   // month = 13;
7
8   var Char@range('a', 'f') af = 'c';
9   // if uncommented, compile-time error
10  // af = 'm';
11
12  var String@regex("[_A-Za-z][_A-Za-z0-9]*") id;
13  id = "pluggableTypeTest";
```

[4] We use class `java.util.regex.Pattern`.

```
14      // if uncommented, compile-time error
15      // id = "#id";
16      // if uncommented, compile-time error
17      // var String@regex("[0-9") wrongRegExpr;
18      // if uncommented, compile-time error
19      // id = "0not an id";
20      var String s = "not an id, there are spaces";
21
22      {
23            // exception ExceptionStr will be thrown
24          id = s;
25      } catch: CatchStr;
26
27
28      var Int@restrictTo{* self prime *} aPrime = 5;
29      {
30          var p = 3;
31            // exception ExceptionStr will be thrown
32            // 4 is not prime
33          aPrime = p + 1;
34      } catch: CatchStr;
35      {
36            // exception ExceptionStr will be thrown
37            // 9 is not prime
38          aPrime = 9;
39      } catch: CatchStr;
40
41      var Int@type(inBytes) size, numBytes;
42      var Int@type(inKbytes) numKbytes;
43      size = 100; // ok
44      numBytes = size;
45      // if uncommented, compile-time error
46      // numKbytes = numBytes;
```

Annotation `restrictTo` takes an attached expression that, when evaluated, should be true. "`self`" inside this expression represents the value of the variable. Hence, in line 28, the metaobject checks if `5` is a prime number using method `prime` of `Int`. This metaobject only do runtime checks.

Metaobject `type` takes a tag parameter. A variable whose type is annotated with

`type` can only receive values of that type or values whose types are annotated with the same tag. Therefore, the assignment of line 43 is correct, the value `100` has type `Int`, a non-annotated type. In line 44, one tagged value is assigned to a variable whose type is tagged. It is correct because the tags are the same. However, there will be a compile-time error in line 46, if uncommented. The tags are different.

The pluggable type system of Cyan is part of the Metaobject Protocol and it can use most of the interfaces of the MOP library. There are a few limitations. The metaobject associated with the type annotated cannot generate code *after* the annotation using interfaces `IAction_parsing` and `IAction_semAn`. The annotation should only be attached to the types of variables, parameters, fields, and return value of methods. The annotation cannot be attached to the superprototype that comes after keyword `extends`, to the interfaces that come after `implements`, or to generic prototype parameters.

An annotation to a type is associated with a metaobject as any other annotation. The metaobject should implement interface

`IActionAttachedType_semAn`

Before explaining when the metaobject methods of this class are called, let us list the statements that are assignment-like:

a) assignments. Because of polymorphism, the right-hand side of an assignment may be a subtype of the type of the left-hand side. A variable whose type `T` is annotated may receive in an assignment an expression whose type is subtype of `T`.[5] This type may not have annotations, it may have the same annotation as `T`, or it may have a different kind of annotation;

b) parameter passing, which is just a different kind of assignment from the real arguments of a message passing (right-hand side) to the formal parameters of a method (left-hand side);

c) method return and function return, which are also kinds of assignments. The right-hand side is the returned expression. The left-hand side is an implicit variable that has the same type as the method or function;

d) `type-case` statements (section A.3). It is not necessary to cover this kind because the type of a `type` expression cannot have an attached annotation and neither can the type that appears after keyword `case`;

e) `for` statements. A value is assigned to the `for` variable in each iteration. However, the type of this value cannot have an attached annotation. It would be necessary to have an `Iterator` prototype whose real parameter, a type, have an attached annotation. Currently, a type with an annotation cannot be a

[5] That includes `T` itself.

parameter to a generic prototype. So, statement `for` only works with regular types.

Consider that all statements cited have a left-hand side and a right-hand side. The metaobject methods of the metaobject associated with an annotation attached to a type are called in these statements when the left-hand or the right-hand side has a type with an attached annotation. The metaobject method

`semAn_checkLeftTypeChangeRightExpr`

declared in

`IActionAttachedType_semAn`

is called when the type of the left-hand side has an attached annotation. The metaobject method

`semAn_checkRightTypeChangeRightExpr`

declared in the same interface is called when the type of the right-hand side has an attached annotation. Both methods return the expression that replaces the right-hand side. If the types of the left-hand side and right-hand side have attached annotations and both generate code for replacing the right-hand side, the compiler will issue an error.

Method `doNotCheckIn` of `IActionAttachedType_dsa` return a list of tuples, each composed of a package and a prototype name. In each of the prototypes of the list, the methods for checking types, shown below, are not used for this metaobject.

```
dsa_checkLeftTypeChangeRightExpr
dsa_checkRightTypeChangeRightExpr
```

This is useful when some trusted code needs to convert values of one type to another. For example, a would-to-be-made annotation `sql` may be attached to a `String` to mean it contains valid SQL code (BATRA, 2018). The prototypes of the list returned by `doNotCheckIn` are allowed to have methods that take a string as parameter, check if it contains a valid SQL code, and return it with type `String@sql`. That is, the prototypes of the list are the ones trusted by the designer of metaobject `sql`. Assuming all SQL code is produced by these trusted prototypes, the conversion methods `dsa_check...` of the class of `sql` should always issue an error. Convertion is only allowed in the chosen prototypes.

Annotation `setVariable` attached to the program or a package in the project file is used for defining trusted prototypes for a given annotation. Use

`@setVariable(moNameDoNotCheckIn, [ "P1", ..., "Pn"])`

for assuring that prototypes `P1`, ... `Pn` are trusted for metaobject `moName`. `Pi` is the full path of a prototype, with its package.

For example, if the annotation name is `tainted`, the project file would be as follows.

```
@setVariable( taintedDoNotCheckIn, [ "badToGood.CastT2UT" ] )
```

```
program
    package main
```

Thus, prototype `badToGood.CastT2UT` is not subject to the rules of metaobject `tainted`, its conversions are not checked for this specific metaobject. That is, methods

```
dsa_checkLeftTypeChangeRightExpr
dsa_checkRightTypeChangeRightExpr
```

of metaobject `tainted` are not used in this prototype.

The Cyan MOP only considers project variables of the form `moNameDoNotCheckIn` if method `allowDoNotCheckInList` of metaobject `moName` returns `true`. The default is `false` and in this case `@setVariable` simply does not work.

1.3.7 Codegs, the Visual Metaobjects

Programming in all of the main languages is made through the use of a text editor, even when using an IDE. Although graphical tools may be used for generating code, they are external to the language. The compiler does not interact with them, the compiler is always called *after* the tools and therefore data is not exchanged in both directions.

This section describes *Codegs*,[6] visual metaobjects that act both at editing time and at compile-time. For that, it is necessary an IDE plugin that has been built by Cassulino (SOUZA, 2017) in his master thesis. The plugin works in Eclipse (ECLIPSE, 2018). At editing time, when the mouse hovers over a Codeg annotation, a method of the Codeg, a metaobject, is called. It is expected that it opens up a window to gather data. After the user gives the data, she or he closes the window and the editing continues. Afterward, during compilation, the compiler asks the Codeg which data was gathered at editing time. It may be used for checks and for generating code. Before going into the details, we will show an example.

There is a Codeg called `color` that helps to choose a color visually at editing time in the Eclipse IDE with the Codeg plugin. During the compilation, the annotation is replaced by the chosen color. In the IDE, if the mouse hovers over an annotation like `@color(red)`, a window opens up as displayed in Figure 8. The color may be chosen visually and the annotation parameter, `red`, may be changed. The first parameter of a Codeg annotation is called *label*. It must be an identifier. If two Codeg annotations of the same source file use the same label, they keep the same data. Any changes in one are reflected in the other.

[6] pronounced as the words *code* and eggs together. The meaning is "originator of code".

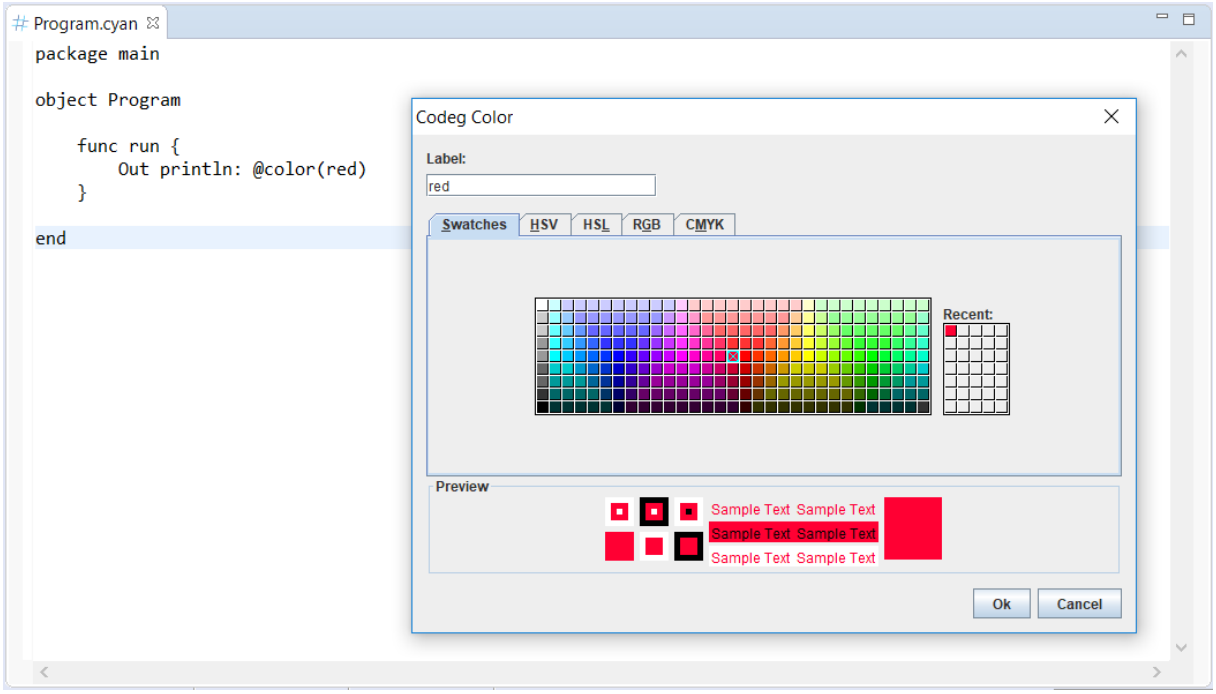

Figure 8 – Codeg `color`

After the user chooses a color, she or he should press button "`Ok`". The color window is closed and the editing may continue. It is expected that a Codeg window has an "`Ok`" button. The color chosen is stored in a hidden file by the compiler (this is soon explained). During the compilation, the metaobject associated with annotation `@color(red)` has access to the content of the file saved at editing time with the color chosen. It is used to generate the correct color number. Therefore, the `color` annotation has type `Int` and, effectively, is replaced by a number in phase semAn.

Another interesting Codeg that is `cyan`, the same name as the language. This Codeg employs the Cyan interpreter (section 2.3) for interpreting Cyan code at editing time. The user can type statements in a text window and click the `Run` button to interpret them. The output or any compilation errors are shown in an output window. Or can click on the `Live` button for automatic interpretation: after some tenths of a second of the last key pressed, the interpreter is automatically called.

1.3.7.1 The Plugin

The Codeg plugin calls the Cyan compiler to parse the source code being edited every few milliseconds after the user stops typing. Only the first compilation phase, parsing, is done. That is enough for identifying the imported packages and loading their metaobjects. A Codeg should be imported from a package as any other metaobject. In the example of Figure 8, Codeg `color` is in package `cyan.lang` (it will be moved to a more appropriate

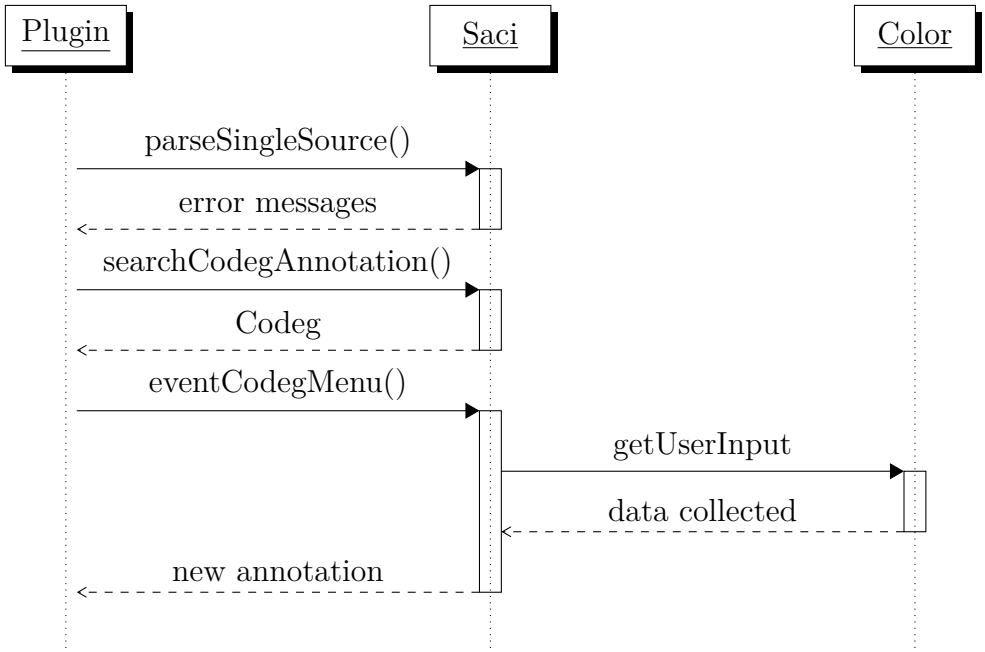

Figure 9 – The sequence diagram of calls between the compiler and the Codeg plugin

place in the future).

After the parsing, Saci knows which annotations are associated with Codegs and where they are in the source code, their line, column, and offset from the start of the text. When the user hovers the mouse over the text being edited, the Codeg plugin knows the offset of the mouse pointer in relation to the start of the text, an information supplied by Eclipse. The plugin calls method `searchCodegAnnotation` of Saci that returns the Codeg that is over the mouse pointer or `null` if none. If a Codeg is returned, the plugin calls method `eventCodegMenu` passing the metaobject as a parameter. The sequence diagram of these calls is displayed in Figure 9.

```
char []eventCodegMenu(CyanMetaobject cyanMetaobject)
byte []getUserInput(
            ICompiler_ded compiler_ded,
            byte []previousCodegFileText );
```

Method `eventCodegMenu` of Saci is responsible for calling method `getUserInput` of the Codeg. It is expected that this method gather user input through a Graphical User Interface.

The compiler calls `getUserInput` passing two parameters. The first one is a restricted view of the compiler. The second one is the text stored in a hidden file associated with the Codeg annotation. The value returned by `getUserInput` is written to this file. Therefore, during an edition, if `getUserInput` is called two times and in the first one the user gives an input (press "`Ok`"), parameter `previousCodegFileText` in the second call is

the value returned by the first call. In the first time, `null` is passed as this parameter.

The byte array returned by `getUserInput` is read and written as a text in the hidden file. It is usually the code of a small DSL that represents the data collected by the Codeg GUI. The file is managed by the compiler, users need not be aware of it.

At compilation time, the byte array returned by `getUserInput` is got by calling method `getCodegInfo` of the Codeg annotation, a method of class

`WrAnnotationAt`

Then, inside a method like `semAn_codeToAdd` of a Codeg, the byte array recorded in a file at editing time is retrieved by the expression

```
this.getAnnotation().getCodegInfo()
```

1.3.7.2 Interface ICodeg

A Codeg is a metaobject whose class implements interface `ICodeg`. There are no special restrictions on which interfaces of the MOP library a Codeg class may implement. Hence, Codegs have the full power of the Cyan MOP. They can even have an attached DSL code. This code may be the text returned by `getUserInput`, for example. To synchronize the GUI with the Codeg annotation (including its attached text), the Codeg class should redefine method `newCodegAnnotation` (see next example) that returns a string that replaces the whole annotation, including the parameters and attached DSL code. The IDE plugin calls this method and does the replacement.

```
public interface ICodeg {

    byte []getUserInput(
              ICompiler_ded compiler_ded,
              byte []previousCodegFileText);

    default boolean demandsLabel() { return true; }

    default String newCodegAnnotation() { return null; }

    default String getNewFirstParameter() { return null; }

    // some methods are elided
}
```

Method `getNewFirstParameter` returns just the new first annotation parameter, the label. Again, the IDE plugin is responsible for changing the source code. This is used in Codeg `color` in which the label can be changed in the window shown in Figure 8.

Method `demandsLabel` returns `true` by default. Therefore, unless said otherwise, every Codeg annotation should have a first parameter that is a label. Some Codeg annotations may not have a label but it is demanded that they are replaced, at editing time, by an expression or some other code. With this feature, user-defined literal numbers can be Codegs. Currently, only the class of literal number `bin` implements `ICodeg`. When the user hovers the mouse over `101bin`, a window opens. The user can choose a number either in binary or in decimal. When "`Ok`" is pressed, the source code is changed.

Method `getUserInput` takes a first parameter of type `ICompiler_ded`, an interface that inherits from `IAbstractCyanCompiler` (section E.2). This interface declares a few methods beyond the inherited ones. There are methods that return information on the fields and methods of the current prototype and on the visible local variables (at the annotation). The information is in the form of strings (it should be because the AST has not been typed yet). Therefore, a Codeg window may show to the user a list of fields or methods of the current prototype and let she or he chooses one to apply some transformation.

A Codeg class may implement interface `ICommunicateInPrototype_ded` for metaobject communication at editing time. After the user stops typing, the IDE plugin calls the compiler for parsing the source file being edited. After the parsing, Saci calls the methods of this interface of the Codegs in the source file. It works like the regular metaobject communication of Subsection 1.2.6 but at editing time. So it is possible to build a *master* Codeg that manages all Codeg annotations in a source file.

1.4 The Cyan MOP and the Problems with Metaprogramming

This section shows how the Cyan MOP addresses some of the problems with metaprogramming described section **??**. The problem name is in **boldface** and a short description of it is in *italics*. Methods of interfaces `IAction_dpp` (subsection 1.3.1, used in the project file), `CyanMetaobjectFromDSL_toPrototype` (subsection 1.3.2, for creating new prototypes before the compilation) and `ICodeg` (subsection 1.3.7, for visual data input) act before the compilation of any source files. They are not discussed here because they cannot be *directly* compared with features of other metaprogramming systems.

MessWithOthers *A metacode in a file changes another source file.*

The creation of new prototypes by the use of interfaces with names

`IActionNewPrototypes_phaseName`

does not cause this problem, which only occurs when a metaobject in one source file is able to change another file.

A metaobject whose current prototype is `P` may replace a message passing that is in-

side another prototype `Q` when the metaobject class implements interfaces `IActionMessageSend_semAn` or `IActionMethodMissing_semAn`. This replacement is *expected*, it should obey the message passing semantics. Therefore, we consider that this non-local change, from `P` to `Q`, does not cause problem MessWithOthers. This problem only happens if the changes are *unexpected*.

All interfaces other than `IActionMessageSend_semAn` or `IActionMethodMissing_semAn` can only replace or add code to the current prototype. Several mechanisms guarantee that:

a) the AST is read-only and, therefore, a metaobject cannot use AST objects to change the code;

b) there is no metamethod of any interface of the Cyan MOP for replacing or adding code to a prototype different from the current prototype;

c) a statement is replaced by a code, given as a string, by metamethod `replaceStatementByCo` described in section 1.2.4. The AST object representing the statement is passed as an argument to this metamethod. Methods of the *MOP library*, including those of AST classes, do not leak AST of base method statements to metaobjects of other prototypes. That is, an exception is thrown whenever a metaobject whose current prototype is `P` calls a method `getStatement` or `getStatementList` that returns the AST object of a statement or list of statements of another prototype `Q`. All metamethods that reveal private parts of a prototype, including method statements, have security checks for preventing leakage. The exception is thrown at compile-time. Note that there is no need to put this type of test on all AST methods. For example, there are no checks in method `getStatementList` of the AST class of Cyan statement `while`. If a metaobject has a reference to an AST object representing a `while` statement, it has already passed a check previously.

WhoDependsOnWho *Metacode are not taken into account when the compiler builds the dependency graph among source files.*

Some metamethods of the *MOP library* return information that links two prototypes or source files.[7] For example, a metaobject whose current prototype is `P` asks for the superprototype of another prototype `Q`. Therefore, whenever `Q` changes, `P` has to be recompiled. Metamethods that link two prototypes in this way also add the dependencies in a compiler *dependency table*.[8] There are statements at the start of every metamethod that link two prototypes to test and add the dependency to the table.

As an example, a metaobject whose current prototype is `P` walks in the `P` AST and, after calling several AST methods, get a reference to another prototype `R`. The compiler

[7] Every prototype is in its own source file.
[8] Currently, this table is not used by the compiler. It will be in future versions.

will add the dependency from `Q` by `P` to the table. All AST methods that are related to *dependencies* between prototypes have statements to add entries to the *dependency table*.

KnowsFriendsSecrets *Metacode in one source file know private information of another file.*

The program view of a metaobject is the same as the view of its current prototype. This is enforced by two techniques:

a) AST metamethods return more or less information according to the caller. The amount of information is exactly the same the compiler makes available to the caller. For example, the AST class that represents a prototype is `WrProgramUnit`. It declares a method `getMethodDecList` that returns prototype method list. Lets assume that a metaobject whose current prototype is `P` calls method `getMethodDecList` of the AST object that represents prototype `Q`. One of the arguments to this method is the compilation environment (usually, the parameter name is `env`). Using `env`, the method is able to know the caller, `P`. `getMethodDecList` returns a list of Cyan methods that includes the `Q` public methods and, based on the relationships between `P` and `Q`, it also returns: (a) the methods of `Q` whose visibility is `package`[9] if `P` and `Q` are in the same package; (b) the methods of `Q` whose visibility is `protected`[10] if `P` is a subprototype of `Q`;

b) if a metaobject whose current prototype is `P` tries to retrieve private information about another prototype `Q` using a method of the AST, this method throwns an exception. An example of that was given in MessWithOthers with method `getStatementList`. Another example is method `getFieldList` of `WrProgramUnit`. It also takes an argument that is a compilation environment. If a metaobject whose current prototype is `P` calls this method of the AST object representing another prototype `Q`, it throws an exception.

The checks cited above are made with a compilation environment object of class `WrEnv` that is passed as an argument to metaobject methods or retrieved from other metaobject method arguments. It cannot be user-created because its constructor takes an object of a class hidden to metaprogrammers. If a developer could create an object of `WrEnv`, she or he could build it to falsify the original object. Hence, a metaobject whose annotation is inside `P` could call, without errors, method `getFieldList` of a `Q` AST object because it pretended to be inside `Q`.

Compiler-Interactions *Metacode interact with compiler low-level structures.*

Metaobjects use wrapped versions of the compiler data structures. Therefore, we

[9] A method preceded by the Cyan keyword `package`. It is visible in all package prototypes.
[10] Methods preceded by the Cyan keyword `protected`, visible in all subprototypes.

consider that the problem Compiler-Interactions is addressed by the Cyan MOP for several reasons:

a) metaprogrammers need not to know complex compiler classes because they handle wrapped and simplified versions of these classes. The wrapped AST classes were designed based on the Cyan language which is less subject to change than the compiler;

b) the wrapped data structures are read-only. There is no way of crashing the compiler by calling the wrong methods;

c) metaobjects do not add code by handling the AST (calling its methods or changing fields). Therefore, metaobjects cannot bypass a compiler check by adding code after the compiler does that check.

WhoDidWhat *The compiler does not link an inserted code to the metacode that made the insertion.*

Metaobjects ask the compiler to add or replace code, they never do this directly. The compiler keeps track of the metaobjects that asked for the changes and, if there is a compilation error aftwards, it points out exactly which annotation introduced the malformed code. The line, column, and source file of the annotation is shown in the compiler error message.

OrderMatters *The order metacode is called inside a source file changes metacode behavior.*

From now on, we will use *metaobject class* for the class of the metaobject (it is in the metaprogram). During a compilation phase and for each prototype, the compiler calls the metamethods of the interfaces associated to that phase in the textual order of the corresponding metaobject annotations. As an example, suppose there are three annotations inside a prototype: `@aaa`, `@bbb`, and `ccc`. Textually, they appear in this order in the source code. The classes of metaobjects associated with annotations `@aaa` and `@ccc` implement interface `IAction_semAn` of the MOP library. In phase semAn, the compiler will call the metamethods `semAn_codeToAdd` (the single method of the interface) of the metaobjects associated with `@aaa` and `@ccc`.

The textual annotation order in the source code is irrelevant if the metamethod call order is not important. The following paragraphs examine all interfaces to discover if the metamethod call order is important or not.

Metaobjects can generate new prototypes but these are created in a new file. Therefore, their creation order is irrelevant. If two metaobjects try to create prototypes with the same name, the compiler issues an error. In phase parsing, the calling order of metaobject methods is irrelevant because: (a) code added by metaobjects will be only

visible, by other metaobjects, in the next phase and (b) information such as documentation can be added by metaobjects to declarations; however, this data cannot be read in phase parsing. Metaobjects can communicate with each other in phases afterResTypes, semAn, and afterSemAn (subsection 1.2.6). The compiler coordinates the communication by first collecting data that every metaobject wants to communicate, which is an object for every metaobject (or `null`). These objects are placed in a list that is shared with all metaobjects. Therefore, the annotation order is not relevant. Metaobjects can warn the compiler that it should issue an error if their classes implement the interface

`IInformCompilationError`

of subsection 1.2.7. The order of the annotations is irrelevant since the metaobjects will warn the compiler regardless of their position in the source code.

There are four methods in interface `IAction_afterResTypes`.

```
afterResTypes_renameMethod
afterResTypes_beforeMethodCodeList
afterResTypes_codeToAdd
runUntilFixedPoint
```

The first one in this list is used to rename methods. The calling order is not important because at most one metaobject can rename each base method. If two or more metaobjects try to rename the same base method, the compiler issues an error. The second method adds statements at the beginning of base methods. Statements generated by two or more metaobjects to be added to the same base method are added in the textual order of the metaobjects' annotations. Therefore, the calling order, which is the textual order, is important. The metaprogrammer may demand a metaobject is the only one allowed to add code to a given base method. In this case, if other metaobjects try to add statements to the base method, the compiler will issue an error.

The compiler calls method `afterResTypes_codeToAdd:` in algorithm FixMeta. The methods of all metaobjects associated with the same prototype are called in rounds. In each round, all metaobjects have access to the same information, produced in the previous round. Therefore, the calling order is irrelevant.

In phase semAn, metaobjects whose classes implement interface `IAction_semAn` can only add code after the annotation. The metamethod calling order is not important because the code added will only be visible in the next compilation phase, afterSemAn. The single method of interface `IActionVariableDeclaration_semAn` can add code after a local variable declaration. If several annotations add code to the same local variable declaration, the code added follows the textual annotation order. However, this is not a serious problem because the annotations and the added code are textually very close to each other.

Methods of interfaces `IActionMessageSend_semAn`, `IActionMethodMissing_semAn`, `IActionFieldAccess_semAn`, `IActionFieldMissing_semAn`, and `IActionAttachedType_semAn` replace a message passing, field access, or an expression by some other code. At most one metaobject can replace a message passing, field access, or expression by some other code. Therefore, the calling order is not important.

The order of calling the metamethods is not important in the afterSemAn phase because, in this phase, the source code is immutable.

In all phases but semAn, all metaobjects view the AST of the previous phase (maybe changed by it) and therefore subproblem **DifferentViews** does not occur. However, in phase semAn, the compiler assigns types to AST objects representing expressions in textual order. Therefore, if one annotation comes after another, the first will view a more complete AST than the last. Hence, subproblem **DifferentViews** does occur in this case.

Metaobjects can associate to declarations[11] documentation, examples, and features (each composed of a name and a literal object). If a metaobject associates, for example, a feature with a declaration that is got by another metaobject in the same compilation phase, the annotation order is important.

```
// add feature ("x", 0) to the current prototype
@aaa
// generate code if feature "x" is associated with the
// current prototype
@bbb
```

In this example, if annotation `@bbb` is put before `@aaa`, metaobject `bbb` will not generated code (assuming feature `x` has not been associated with the current prototype previously). However, this cannot happens because documentation, examples, and features can only be added to declarations in phases parsing and afterSemAn and they can only be read in the other phases. Therefore, in each phase all metaobjects share the same program view and subproblem **DifferentViews** is not caused by information added or got from declarations.

In each compilation phase, metaobjects view the AST built or changed in the previous phase, they do not view the changes made in the same phase. In phase

`afterResTypes_codeToAdd:`,

metaobjects that participate in algorithm FixMeta view the signatures generated by other metaobjects. All of them view the same information and therefore they share the same program view.

If metaobjects do checks in phase afterSemAn, subproblem **InvalidateChecks** of OrderMatters does not happen because, in this phase, the program cannot be changed

[11] The program, packages, prototypes, methods, fields, and local variables

anymore. If a metaobject do checks in other compilation phases, it is badly designed. Therefore, we consider that the Cyan MOP addresses the subproblem **InvalidateChecks**.

InfiniteMetaLoop *Metacode can generate metacode that, in its turn, generate metacode, and so on.*

This problem cannot occur in Cyan because, although the code added by metaobjects can contain annotations, their associated metaobjects are only active in the *next* compilation phase.

Nontermination *Metacode may not finish its computation.*

Metaobject methods cannot take more than a time limit, given by a compiler option, to finish their executions. Otherwise, the compiler exits after issuing an error message. Therefore, the Cyan MOP addresses problem Nontermination.

Nondeterminism *Metacode is nondeterministic.*

Metaobjects can read and write to files, get the current time, call a random number generator, interact with the network, and so on. That makes metaobjects nondeterministic. There is no easy way to make them deterministic. That could only be done if they use a special language in which any interactions with the world are prohibited. A restricted version of interpreted Cyan could be used for that. However, this would place such great limitations on metaobjects that we prefer not to use this solution.

NoGeneratedCodeGuarantees *Metacode may generate defective code.*

In Cyan, metaobjects can generate malformed code. However, the compiler will point out any defective code introduced by metaobjects.

NoContracts *The contract between the metacode and the base code is explicitly stated.*

The Cyan MOP does not require a contract between the base code and metaobjects. However, this contract can be specified by metaobject `concept` of subsection 2.2. In the following example, let us assume that metaobject `addSort` adds a `sort` method to `PersonList` for sorting the list.

```
@addSort
@concept{*
    Person has [
        func <=> Person -> Int
    ]
    PersonList has [
        func sort
```

```
    ]
*}
object PersonList
    ...
end
```

This metaobject requires that `Person` has a `<=>` method[12] used in the `sort` method. Metaobject `concept` is used for enforcing that `Person` has a `<=>` method and for demanding that the final version of `PersonList` has a `sort` method. If we assume that only `addSort` adds code to `PersonList`, `concept` is demanding that this metaobject adds a `sort` method to this prototype.

C++ *concepts* for templates (GREGOR et al., 2006) were motivated by a problem similar to NoContracts. In this language, *concepts* are predicates on generic prototype parameters. They restrict parameters. For example, a concept may demand that a parameter has a method `compareTo` that would correspond to the Cyan method `<=>` in the previous example.

Metaobjects generate code that may not fit any patterns and, therefore, the code would not be easily described by a DSL code. And the requirements on metaobject parameters and environment may be arbitrarily complex. Therefore, the requirements and demands should be specified by a DSL that could be as complex as the metaobject metacode. Therefore, it is not worthwhile to solve the NoContract problem in Cyan. On the other side, C++ concepts support a simple DSL with easily described requirements because the code of instantiated classes is generated from a template (the template class) and the demands on the parameters are simple.

CircularDependency *Metacode may depend on information produce or changed by other metacode. This dependency relation may be circular.*

In phase parsing, metaobjects cannot view the code or information generated by other metaobjects. Therefore, this problem does not occur in this pase. Algorithm FixMeta of subsection refafixMeta deals with some kinds of circular dependencies in phase afterResTypes. All metaobjects that participate in this algorithm have the same information on the current prototype. However, no algorithm, includingFixMeta, is able to deal with unsolvable circularity. As an example, suppose a metaobject `field0` generates, for each field `f`, another field with name `f0` if there is no field with this name. If the current prototype has a single field `f`, the metaobject would generate fields `f0`, `f00`, `f000`, and so on. In this case, algorithm FixMeta would run the maximum number of rounds and issue an error.

[12] `e1 <=> e2` returns `-1`, `0`, or `1` if `e1` is less than, equal, or greater than `e2`, respectively.

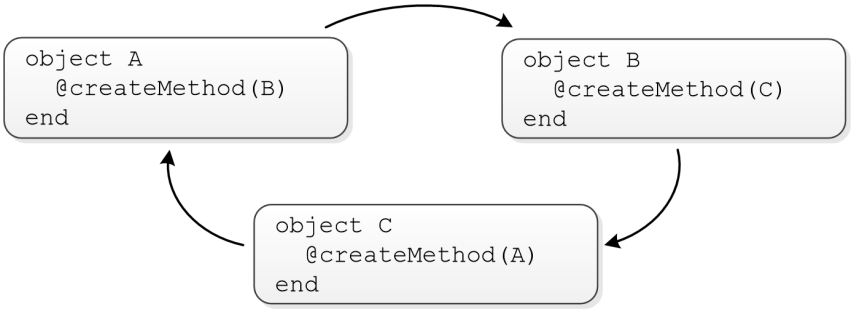

Figure 10 – Dependencies among prototypes

There may be circular dependencies among metaobjects of several prototypes as shown in the example of Figure 10. An arrow from prototype `X` to `Y` means that the single metaobject of `X` uses information about `Y`. Metaobject `createMethod` adds to the current prototype method

```
func numMethodsY -> Int = numMethods;
```

in which `Y` is the annotation parameter and `numMethods` is the number of methods declared in `Y`. Currently, all of the `createMethod` metaobjects will generate incorrect code because base methods are added in phase afterResTypes and, in this phase, metaobjects view AST generated in phase resTypes. Methods added in phase afterResTypes are not considered. Therefore, the value each `numMethodsY` method returns is `0`. Let's assume that metaobjects view the code added by other metaobjects in the compilation phase afterResTypes. In this case, whenever the order that the metaobjects add code to their prototypes, one of them will generate incorrect code: the number returned by the base method `numMethodsY` will be `0` when it should be `1`.

Circular dependency among metaobjects of different prototypes in phase afterResTypes could be addressed by extending algorithm FixMeta to deal with all prototypes of a *dependency cycle*. But how to build this cycle? It could be the dependency graph built by the compiler, before phase afterResTypes, based on the types used by the prototypes (considering every type appearing outside method bodies in the prototype). This would not work in the example of Figure 10. Metaobject `createMethod` adds a dependency from its current prototype and its parameter. This dependency is not discovered by the compiler in phase resTypes because the parameters to the annotations are just symbols, they do not represent the types with the same name. The dependency would be discovered in phase afterResTypes during the execution of algorithm FixMeta, when it should be added to the algorithm data. The dependencies added during FixMeta execution could be removed in the next round of calls. Any language solution to this complex mesh would make the Cyan MOP too complex for regular use.

In phase semAn, there may be *circular dependency* for the same reasons it occurs in phase afterResTypes. Algorithm FixMeta could be adapted to address some kinds

of circularity. However, we deemed that is not worthwhile since no real case of circular dependency in phase semAn was found either in the literature or in real-case examples of metaprogramming languages. The program code cannot be changed from phase afterSemAn onwards and, therefore, there cannot be any *circular dependency*.

A metaprogramming system with severe restrictions on how it changes the code and does checks will have few or none of the problems described in this section. This is not the case with the Cyan MOP. It is powerful enough to implement complex metaobjects as demonstrated in chapter 2 which presents some non-trivial metaobjects that could not have been done in a limited metaprogramming system.

1.5 Shortcomings of the Cyan MOP

The code generated by a metaobject in a compilation phase may contain annotations. However, the associated metaobjects will never generate code in the same phase. They will only be activated in the next phase. Therefore, a metaobject cannot make use of annotation generation in order to help itself generate code. As an example, suppose we want to implement metaobject `propertyAll` that takes pairs (name, Type) and generates, for each pair, a field (with that name and type) and get and set methods. This metaobject cannot generate, for each field, the code

```
@property var Type name
```

because `@property` would only be *activated* in the next compilation phase, semAn, when it does not generate code. Metaobjects can only be composed by putting code generation methods into a library imported by several metaobject classes or prototypes.

Some features are missing in the Cyan MOP:

a) to intercept compiler error and warning messages. Metaobjects should be able to intercept and change the error and warning messages. For example, to make them clearer;

b) generate code in the subprototype of an annotated superprototype. Currently, inheritance is intercepted but the subprototype can only be checked;

c) intercept code generation. The target language could be changed or the generated code could be optimized.

In phase afterResType, metaobjects of the same prototype can view the code generated by others because of algorithm FixMeta. This does not happens in phase semAn. The consequences are that some uncommon metaobjects cannot be implemented correctly. For example, suppose annotation `@printStat` generates

```
numStat println;
```

in which `numStat` is an `Int` literal, the number of statements of the current method (the

annotation is inside this method). Since a metaobject cannot view the code generated by others, `numStat` may not reflect the final code.

```
func wrong {
    @printStat
    @printStat
}
```

The final code of the method above is

```
func wrong {
    1 println;
    1 println;
}
```

It should be

```
func wrong {
    2 println;
    2 println;
}
```

This problem can be solved if algorithm FixMeta is adapted for phase semAn. That would be worthwhile if metaobjects that need to know the code generated by others in phase semAn are common. There are not — we could not find any metacode in any language that needs this feature. Therefore, FixMeta was not adapted to phase semAn.

2 Metaobjects in Action

This chapter presents metaobjects used in the Cyan libraries and some others that are themselves research topics. They attest that the Metaobject Protocol of Cyan is robust enough to build complex metaobjects in many different domains. That means the mechanisms used by the Cyan MOP to address the problems with metaprogramming (section 1.4) do no limit significantly the MOP's power. This chapter is organized as follows. Section 2.1 describes metaobject `grammarMethod` for the fast building of DSLs using method keywords and regular expressions; that is, using only regular Cyan syntax. Annotations of metaobject `concept` of section 2.2 take a DSL code for restricting the type parameters of a generic prototype. They check the correctness of an instantiation of a generic prototype instantiation before the actual instantiation. The result are better specified generic prototypes and clearer error messages. Section 2.3 presents some metaobjects able to interpret Cyan code at compile-time. The make it easy to build metaobjects because there is no need of creating a Java class or a Cyan prototype. The metaobject code is given with the annotation.

The last section shows the relationships among metaobjects and the language Cyan and its libraries. The language was designed simultaneously with the metaobjects it uses. Cyan would be very different without them. Package `cyan.lang`, imported automatically by every source file, depends heavily on metaobjects. If they were not used, many language features would have to be designed to support this library.

2.1 Grammar Methods

Annotation `grammarMethod` permits the definition of Domain Specific Languages (DSL) using message keywords. Although the DSL is limited to use message keywords, expressions, and a regular language, it is easy to define. The annotation should be attached to a method and it should have an attached DSL code. The example of Listing 2.1 shows a method `addAll:` annotated with `grammarMethod`. We say that `addAll:` is a *grammar method*.

The attached DSL code is a regular expression containing message keywords, types, parentheses, and regular expression operators. All of the usual operators are supported: `|` for alternative, `+` for one or more repetitions, `*` for zero or more repetitions, and `?` for optional regular expression. Because of the annotation `grammarMethod`, an object of `MyList` may receive a message with multiple `add:` keywords, as happens in line 3 of the following example.

Listing 2.1 – Grammar method `addAll:`

```
object MyList

    @grammarMethod{*
        (add: Int)+
    *}
    func addAll: Array<Int> array {
        for elem in array {
            list add: elem
        }
    }

    func getList -> Array<Int> = list;

    let Array<Int> list = Array<Int>();
end
```

```
    func myListTest {
        let myList = MyList();
        myList add: 0 add: 1 add: 2;
        var s = "";
        for elem in myList getList {
            s = s ++ elem
        }
        assert s == "012";
        "myListTest" println;
    }
```

There is no method `add:add:add:` in `MyList`. The metaobject `grammarMethod` intercepts messages for which there is no method. It then replaces the message passing by a message using keyword `addAll:` and an array as argument. Therefore,

```
myList add: 0 add: 1 add: 2;
```

is replaced by

```
myList addAll: [ 0, 1, 2 ];
```

In order to understand the following sections, it is important to distinguish two DSLs related to grammar methods. The code of the first one is attached to the annotation `grammarMethod`. It is a regular expression. Hence, the code itself is a grammar definition. A code of the second DSL is a message passing that should obey the regular expression. In the example above, it is "`add: 0 add: 1 add: 2`".

The grammar of the DSL for regular expressions is given in the next subsection. All code attached to an annotation `grammarMethod`, a regular expression, should be recognized by this grammar. The text attached to each annotation defines a regular expression that is also a grammar. The message passing should be recognized (or matched) by this regular expression in order to use the grammar method.

2.1.1 The Metaobject Class

The metaobject class of `grammarMethod` implements several Java interfaces of the MOP library. We explain two of them. The first is `IParseWithCyanCompiler_parsing`. The DSL code attached to the annotation is parsed with the help of the Cyan compiler. The `parsing_parse` of `IParseWithCyanCompiler_parsing` builds an Abstract Syntax Tree based on the DSL grammar that follows. KeywordGrammar is the start grammar rule. IdColon represents a message keyword which is an identifier attached to a colon, like "`add:`".

```
KeywordGrammar    ::= "(" KeywordUnitSeq ")" "*" |
                      "(" KeywordUnitSeq ")" "+" |
                      "(" KeywordUnitSeq ")" |
                      "(" KeywordUnitSeq ")" "?"
KeywordUnitSeq    ::= KeywordUnit |
                      KeywordUnit KeywordUnit { KeywordUnit } |
                      KeywordUnit KeywordUnit { "|" KeywordUnit }
KeywordUnit       ::= SelecGrammarElem |
                      KeywordGrammar
SelecGrammarElem  ::= IdColon |
                      IdColon Type1, Type2, ... Typen |
                      IdColon Type1 |
                      IdColon "(" Type ")" ( "*" | "+" )
```

When the compiler finds a message send for which there is no adequate method, it looks for a metaobject attached to the receiver type or its methods that can handle this "missing method error". That is, the compiler looks for a metaobject whose class implements `IActionMethodMissing_semAn`. If the receiver type is a prototype with a grammar method, the compiler calls method

```
semAn_missingKeywordMethod
```

of its attached metaobject. This method has access to the AST of the DSL code, which

is stored in a field of the metaobject. It then tries to match the AST with the message passing. Therefore, there is an attempt to do a pattern matching of the DSL code attached to the annotation, a regular expression, to the message passing. If they do not match, method `semAn_missingKeywordMethod` returns null and possibly other grammar methods may try a match. If no metaobject can handle this "method missing" problem, the compiler issues an error.

If there is a match, the `grammarMethod` metaobject has to replace the message passing by another message passing using the method to which the annotation is attached to. That is, a message passing with `add:` keywords to a `MyList` object is changed to a message passing to `addAll:`, the method to which the annotation is attached. All of the objects of the message passing are grouped into a single object which may be composed of arrays, tuples, and unions. The next Subsection explains how to calculate the type of this object, which is also the type of the single parameter of the annotated method.

2.1.2 Rules for the Method Parameter Type

The previous Subsection presented the grammar of the DSL of metaobject `grammarMethod`. The metaobject uses this grammar for two goals:

a) to build a single object from the arguments of all keywords of a message passing that would cause the "method missing" error. This object is passed to a call to the annotated method. In the example

```
myList add: 0 add: 1 add: 2
```

the object passed as parameter to `addAll:` is `[ 0, 1, 2 ]`.

b) to check the type of the single parameter of the method annotated with `grammarMethod`. This type should be equal to the type of the object described in item a). It should reflect the DSL code of the attached annotation.

The type of the parameter of the annotated method is calculated according to the rules of table 1. The type associated with the rule in the left is in the right. "`typeof(P)`" is the type associated with the grammar production `P`. If there are many occurrences of a rule name `R`, we use `Ri` for the `i`th occurrence of the rule. That is, when using the rule

```
A ::= B { B }
```

`B3` is the third occurrence of `B` in an expression. Note that, in the table, a type may itself be a union type like `Int|String`.

Table 2 displays some high-level examples of regular expressions and their types according to table 1. If the expression contains several symbols, as

```
R R ... R
```

`Ti` is the type associated with the `i`th occurrence of `R`. Table 3 shows some real examples

Table 1 – The types associated with rules of the `grammarMethod` DSL

Rule	Type
KeywordGrammar ::= “(” KeywordUnitSeq “)” “*”	`Array<typeof(KeywordUnitSeq)>`
KeywordGrammar ::= “(” KeywordUnitSeq “)” “+”	`Array<typeof(KeywordUnitSeq)>`
KeywordGrammar ::= “(” KeywordUnitSeq “)”	`typeof(KeywordUnitSeq)`
KeywordGrammar ::= “(” KeywordUnitSeq “)” “?”	`Union<some, typeof(KeywordUnitSeq), none, Any>`
KeywordUnitSeq ::= KeywordUnit	`typeof(KeywordUnit)`
two or more KeywordUnit KeywordUnitSeq ::= KeywordUnit KeywordUnit { KeywordUnit }	`Tuple<typeof(KeywordUnit1), ..., typeof(KeywordUnitn)`
KeywordUnitSeq ::= KeywordUnit | KeywordUnit { | KeywordUnit }	`Union<f1, typeof(KeywordUnit1), ..., fn, typeof(KeywordUnitn)>`
KeywordUnit ::= SelecGrammarElem	`typeof(SelecGrammarElem)`
KeywordUnit ::= KeywordGrammar	`typeof(KeywordGrammar)`
SelecGrammarElem ::= IdColon	`Any`
SelecGrammarElem ::= IdColon T1	`T1`
with n >= 2 SelecGrammarElem ::= IdColon T1, T2, ... Tn	`Tuple<T1, T2, ... Tn>`
SelecGrammarElem ::= IdColon “(” Type “)” (“*” | “+”)	`Array<typeof(Type)>`

of regular expressions with their types.

2.1.3 A More Complex Example

The example that follows uses all but the `*` operator. The parameter of the method attached to the `grammarMethod` operator should have a type that matches the regular expression according to rules given in Subsection 2.1.2. It is usually difficult to find the type by oneself. Thus, let the compiler help you. Just put the name of the parameter without the type. The metaobject issues an error and tells the correct type.

```
package main

object Player
```

Table 2 – High-level regular expressions and their types

Expression	type
`T1`	`T1`
`R R ... R`	`Tuple<T1, T2, ..., Tn>`
`Id ":" R R ... R`	`Tuple<T1, T2, ..., Tn>`
`Id ":"`	`Any`
`Id ":" T`	`T`, which must be a type
`Id ":" "(" T ")" "*"`	`Array<T>`
`Id ":" "(" T ")" "+"`	`Array<T>`
`"(" R ")"`	`typeof(R)`
`"(" R ")" "*"`	`Array<typeof(R)>`
`"(" R ")" "+"`	`Array<typeof(R)>`
`"(" R ")" "?"`	`Union<some, typeof(R), none, Any>`
`T1 "|" T2 "|" ... "|" Tn`	`Union<f1, T1, f2, T2, ..., fn, Tn>`
`R "|" R "|" ... "|" R`	`Union<f1, T1, f2, T2, ..., fn, Tn>`

```
@grammarMethod{*
    ( (playVideo: String (duration: Int)?) |
      playMusic: String |
      pause: Int |
      stop:
      )+
*}
func action:
       Array<
         Union<f1,
           Tuple<
              String, // video name
                   // duration
                 Union<some, Int, none, Any>
           >,
           f2, String, // music name
           f3, Int, // seconds to pause
           f4, Any // stop
         >
       >
       params {
    // code that plays everything
    // see the complete code at www.cyan-lang.org
}
```

Table 3 – Regular expressions and their types

`Int`	Int
`add: Int`	Int
`add: Int, String`	`Tuple<Int, String>`
`add: (Int)*`	`Array<Int>`
`add: (Int)+`	`Array<Int>`
`(add: Int)*`	`Array<Int>`
`(add: Int)+`	`Array<Int>`
`(add: Int | String)`	`Union<Int, String>`
`(add: (Int | String)+)`	`Array<Union<Int, String>>`
`(add: Int | add: String)`	`Union<f1, Int, f2, String>`
`key: Int value: Float`	`Tuple<Int, Float>`
`nameList: (String)* (size: Int)?`	`Tuple<Array<String>, Union<some, Int, none, Any>>`
`coke:`	Any
`coke: | guarana:`	`Union<f1, Any, f2, Any>`
`(coke: | guarana:)*`	`Array<Union<f1, Any, f2, Any>>`
`(coke: | guarana:)+`	`Array<Union<f1, Any, f2, Any>>`
`((coke: | guarana:)+)?`	`Union<some, Array<Union<f1, Any, f2, Any>>, none, Any>`
`((coke: | guarana:)?)+`	`Array<Union<some, Union<f1, Any, f2, Any>, none, Any>>`
`amount: (gas: Float | alcohol: Float)`	`Tuple<Any, Union<f1, Float, f2, Float>>`

`end`

The regular expression of the text attached to the annotation `grammarMethod` says that in a message passing there may be a repetition of following keywords: `playVideo:`, `playMusic:`, `pause:`, and `stop:`. After `playVideo:` there may appear a keyword `duration:` that takes an `Int` as parameter. `stop:` should be the last keyword in a message, a fact that is not checked. This prototype can be used as follows.

```
Player()
    playVideo: "Color demo" duration: 30
    playVideo: "Reef fish"
    pause: 10
    playMusic: "Bach BC Allegro"
    stop:;
```

The arguments to the message passing are grouped by the `grammarMethod` metaobject

into a single object passed as parameter to method `action:`. This object is in section E.6.

2.1.4 Additional Checks

A regular expression may not be enough for assuring the validity of a message passing. Additional checks may be needed, as in the example of prototype `Player` in which the last keyword in a message passing should be `stop:`. A more interesting example is that of a language-C like `printf` method of prototype `Out`:

```
@grammarMethod{*
    ( printf: (Any)+ )
    checkPrintf
*}
func printfAll: Array<Any> array {
   // elided
}
```

The regular expression allows a message passing with incorrect parameters like

```
Out printf: "%d is %s", "zero", 0;
```

To allow additional checks, a list of action functions can be given after the regular expression in the text attached to the annotation `grammarMethod`. In the above example, there is one action function, `checkPrintf`. Therefore, the source code of prototype `Out` should import a package with a metaobject that implements interface `IActionFunction` and that has name `checkPrintf`. It does because this action function belongs to package `cyan.lang`. The search for an action function of a given name is made with method `searchActionFunction` of the parameter of method `parsing_parse`, declared in `IParseWithCyanCompiler_parsing`.

Metaobject `grammarMethod` calls each action function of the list following the regular expression. It passes as argument a tuple consisting of the object that received the message and the AST object of the message. In the example, the receiver object is `Out` and the AST object would represent

```
"%d is %s", "zero", 0
```

The action function should return `null` if there is no error or a string that is an error message.

Note that an action function can be used to implement the parsing of a non-regular grammar. If the grammar use the keywords `kw1:`, `kw2:`, ... `kwn:`, the regular expression could be

```
( kw1: (Any)* | kw2: (Any)* ... kwn: (Any)* )+
```

This regular expression matches any expression that uses the cited keyword with any number of arguments (except `Nil`, of course). Now an action function can be used to reject some of the message passings accepted by the regular expression, thus implementing the syntactic and semantic analysis of a non-regular grammar.

2.1.5 Discussion

An embedded Domain Specific Language employs the same syntax and semantics of a host language. DSL code is just regular code of the host language but it expresses a domain more clearly. Embedded DSLs are supported by many object-oriented languages such as Groovy (DEARLE, 2010), Scala (**??**), and Ruby (FLANAGAN; MATSUMOTO, 2008).

In Cyan, embedded DSLs can be implemented with regular code, probably using anonymous functions and runtime metaprogramming. And they can also be implemented using grammar methods. The DSL grammar is that given in the text attached to the annotation `grammarMethod`. The code of the DSL is the message passing for which the compiler does not find an appropriate method.

The metaobject `grammarMethod` automatizes several tasks that should otherwise be made by the programmer:

a) it does the syntactical analysis of the message passing using the grammar of the regular expression. The lexical analysis was previously made by the compiler;

b) it automatically builds the AST of the message passing according to the attached regular expression. For that, it uses the AST object of the message passing built by the compiler.

The Cyan statements of a grammar method, the one to which the annotation is attached to, are responsible only for associating a semantics to the message passing. In the `MyList` example, the semantics would be to add the elements to the list. In the `Player` example, the semantics is to call methods for playing videos, playing music, pausing, and stopping everything.

The object passed as an argument to the grammar method is built from arrays, tuples, and unions. Therefore, to interpret it in the grammar method, it is necessary to scan arrays, access tuple elements, and introspect unions. That can be made with the `for` statement, methods `fi` (`i` a number) of prototypes `Tuple<...>`, and statement `type-case`.

Grammar methods resemble not too vaguely the macros of language Rust (Rust..., 2018) (KLABNIK; NICHOLS, 2022), which uses regular expressions for specification. However, Rust macros operate at the syntactic level only, the regular expressions cannot match the types of arguments of the macros. The `grammarMethod` metaobjects match both

the syntax, a sequence of message keywords and arguments, and the types of arguments and keyword parameters. Macros in Scala (BURMAKO, 2018) also consider the types, but they do not use regular expressions for pattern matching (in macros). Only `*` can be used, as in regular Scala code, to mean "zero ou more". For example, `c.Expr[String]*` means "zero ou more `String` expressions".

Polymorphism does apply to grammar methods. Thus, the regular expressions of two annotations may be `(add: Int)+` and `(add: String)+`. And a grammar method may be overridden in a subprototype as any other method.

2.2 Concepts for Generic Prototypes

Cyan supports generic prototypes, which are abstractions of prototypes over one or more type parameters (see section A.5). The language does not restrict type parameters in any way, type checking in generic prototypes is only made after they are instantiated with real types. Errors in the instantiated prototype are difficult to interpret because they are caused by code that was not made by the user of the generic prototype. And they may be deep in a stack of generic instantiations: a generic prototype may instantiate other generic prototype and the error may appear only in this last one. As an example, prototype `GroupList<T>` of Listing 2.2 assumes its parameter `T` has methods

```
func unit -> T
func * T -> T
func inverse -> T
```

because of the message passings of lines 9 (and 22), 14 (and 27), and 25 (and 27). There is an implicit requirement on the formal type parameter `T` that depends on the source code of `GroupList`.

If `GroupList` is instantiated with a type that does not declare one of the three methods, there is a compile error. Saci shows the error in a line of `GroupList`, breaking modularity. Users of a prototype have to know details of a prototype in order to use it. The interface is not enough. This problem is ameliorated by commenting the generic prototype parameters. The requirements for each of them is expressed in words. This is what most languages do. However, the compiler cannot check the demands expressed in the comments, this is not the ideal solution. There is one more problem with generic classes and prototypes: the error messages may be very clumsy.

Gregor et al. (GREGOR et al., 2006) mention the error caused by the use of generic function `sort` with incorrect arguments. The GNU C++ compiler, in 2006, reported more than two kilobytes of output with six different error messages. Generic classes and functions in C++ are called templates. An instantiation of a template class (function) is

Listing 2.2 – Generic prototype `GroupList`

```
1  package main
2
3  object GroupList<T>
4
5      func add: T elem { array add: elem }
6
7      func multAll -> T {
8          if array size == 0 {
9              return T unit;
10         }
11         else {
12             var p = array[0];
13             for n in 1..< array size {
14                 p = p * array[n]
15             }
16             return p
17         }
18     }
19
20     func inverseMultAll -> T {
21         if array size == 0 {
22             return T unit;
23         }
24         else {
25             var p = array[0] inverse;
26             for n in 1..< array size {
27                 p = p * array[n] inverse
28             }
29             return p
30         }
31     }
32
33     let Array<T> array = Array<T>();
34
35 end
```

made by creating a brand new copy of the class (method) with the instantiation arguments. Just like Cyan and unlike languages like Java and Scala, in which the same class is used for all instantiations. In these languages, the problems cited till now do not happen. But there is a price for that: there are many restrictions on the use of type parameters and instantiated generic classes. In Java, the type parameter of a generic class[1] cannot be a primitive type, new instances of it cannot be created, and it cannot be the type of a static field. There are also limitations of an instantiation `G<R>` of a generic class `G<T>`. Arrays of `G<R>` are illegal, objects of `G<R>` cannot be thrown by command `throw`, and there are limitations in the use of `G<R>` when overloading methods.

Concepts are predicates on template/generic parameters. They can be given as comments in the code or there may exist a *concept language* to express them. In the last case, the compiler has the power of restricting the legal parameters to a template class/function or generic prototype. In this text, *concepts* always refer to the language feature. They were proposed by Stroustrup (STROUSTRUP, 2003) for the language C++. However, they have not been adopted yet, although they may be in C++ 20 (SMITH, 2018). *Concepts* are able to check the real arguments to a template/generic prototype *before* the instantiation, issuing clear error messages. Concepts can also be used for compiling template classes and functions before they are instantiated with real arguments. The source code of a library need not be exposed to its users.

Cyan supports *concepts* through the metaobject `concept` whose annotations should be attached to prototypes. The text linked to the annotation should be code of a *concept language*. It can restrict the parameters and even the current prototype. Prototype `GroupList<T>` can restrict parameter `T` using an annotation `concept`.

```
@concept{*
    T has [
        func unit -> T
        func * T -> T
        func inverse -> T
    ]
*}
object GroupList<T>
    ...
end
```

In an instantiation `GroupList<Int>`, the metaobject `concept` associated with the above annotation issues two errors because `Int` does not define either `unit` or `inverse` methods. The error is pointed out in the place of instantiation, not in the prototype `GroupList`. Besides that, Saci gives the full stack of prototype instantiations. This is necessary

[1] https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html

because there may be a chain of prototype instantiations and the error may be in the last instantiated prototype. For example, suppose prototype `Test` declares a variable whose type is `A<Int>`. Generic Prototype `A<T>` declares a variable whose type is `GroupList<T>`. Therefore, there would be an error in `GroupList<Int>` and Saci shows the error message and a stack of prototype instantiations:

```
In file C:\Dropbox\Cyan\cyanTests\simple\main\--tmp\A(Int).cyan
(line 7 column 28)
object/interface main.A<Int>
A concept associated with generic prototype 'main.GroupList<Int>'
expected that method 'unit' were in prototype 'cyan.lang.Int'
Stack of generic prototype instantiations:
    main.A<Int> line 7 column 28
    main.Test line 364 column 13

    public func run  { var GroupList<Int> g    }
```

2.2.1 The Grammar of the Concept Language

Metaobject `concept` supports many kinds of statements. The complete grammar follows. In it, Id is a Cyan identifier, IdColon is an Id joined with a colon (as "`at:`"), CType is a Cyan type, including "typeof(expr)", which results in the type of `expr`. The construct { A }* is a repetition of zero or more A's separated by commas. LeftSeq is a left sequence of symbols whose definition is the same as the left sequence of symbols that delimits the text attached to an annotation. Idem for RightSeq. TEXT is any text. The initial grammar symbol is `CL`.

CL ::= { CStat ["`,`" CMessage] }*

CStat ::= CType "is" CType
| CType "implements" CType
| CType "subprototype" CType
| CType "superprototype" CType
| CType "interface"
| CType "noninterface"
| CType "identifier"
| CType "has" "`[`" { CSig } "`]`"
| CType "in" "`[`" { CType }* "]"
| Axiom
| CCall
| ! CStat

CSig	::= "`func`" (CUnarySig | CKeySig) ["`->`" CType] [CMessage]
CUnarySig	::= Id
CKey	::= IdColon { [CType [Id]] }*
CKeySig	::= CKey { CKey }
Axiom	::= "`axiom`" CKey LeftSeq TEXT RightSeq
CCall	::= Id { "`.`" Id } "`(`" { Id }* "`)`"
CMessage	::= LiteralString

Statement "`A implements B`" of the concept language demands that `A` implements interface `B`. `A` should be a non-interface prototype. Statement "`B subprototype A`" requires that `B` be a subprototype of `A`, the same requirement of "`A superprototype B`". Use `A interface` if `A` must be an interface and `A noninterface` if `A` must be a non-interface prototype. Statement `T interface` demands that `T` must be a *identifier parameter*, an identifier starting with a lowercase letter.

"`has`", as shown in the `GroupList` example, requires that a prototype declares a list of methods. Here "prototype" may be an interface or a non-interface prototype. Use "`A in [ ... ]`" if `A` should be one of the list elements. Operator `!` negates the following statement.

After each statement, there may appear an error message that is issued if the statement test fails. An error message can also follow each method signature in a "has" statement.

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

When the generic prototype is instantiated, the Cyan compiler replaces every formal type parameter, as `T`, by the real parameter in all DSL code attached to all annotations of the prototype. Thus, in `GroupList<Int>`, the first error message of this example becomes

```
"Int must have a method 'unit'"
```

Each metaobject class may choose if the formal parameter types are replaced by the real

types in an instantiation. The default behavior is to replace them. A metaobject class should override method

`getReplacementPolicy()`

if another replacement policy is desired.

"`axiom`" is used for generating test cases for the prototype attached to the annotation `concept`. The metaobject creates a prototype with a method with the signature given after "`axiom`" and with the code given between LeftSeq and RightSeq. This prototype is written in the test directory of the project. In the example that follows, the LeftSeq is `{%`. The test prototypes are only created if the annotation `concept` takes a parameter equal to "`test`".

```
@concept(test){*
    T has [
        func unit -> T "T must have a method 'unit'"
        func * T -> T
        func inverse -> T
    ], "T must have methods unit, *, and inverse"

    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
    %},

    axiom unitTest: T a, T b, T c {%

        if (a * a unit != a unit * a) ||
           (b * a unit != b unit * b) ||
           (a unit * b unit != c unit * c unit) {
            ^"The unit element of T is not an identity"
        }
        ^Nil
    %},

    axiom inverseTest: T a, T b, T c {%

        if (a * a inverse != b unit) ||
```

```
            (a unit != b inverse * b) ||
            (c inverse * c != T unit) {
             ^"The inverse operation is not working properly"
        }
        ^Nil
    %}

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

Code snippets of the concept language can be put in a file inside a directory called `--data` of a package. The file must have a name

```
filename(Id,Id,...Id).concept
```

There should be no space after the "," and the `Ids` may be used inside the file. In the `DATA` directory of package `cyan.lang`, there are several of such files: `lessThan(T).concept`, `addable(T, R).concept`, `arithmetic(T).concept`, etc. These concept files are used inside the DSL of an annotation `concept` as if they were functions of language C, but preceded by the package name:

```
cyan.lang.lessThan(R)
```

When the metaobject `concept` finds such a statement, it loads the file `lessThan(R).concept` and replaces all occurrences of identifier `T` by `R`. It then includes the transformed text into the DSL code of the annotation. Hence, this kind of "function" call works like a language-C `include` but with text replacement.

The text of a `concept` file should follow the grammar for the concept language. It may have every statement of the language, including axioms. Therefore, it is easy to generate test cases for prototypes that implement arithmetic and comparison operators. Just use the `concept` annotation.

```
@concept(test){*
    cyan.lang.arithmetic(MatrixElement),
    cyan.lang.comparison(MatrixElement)
*}
object MatrixElement
   ...
end
```

Common predicates on types and common test methods can be gathered in packages and shared. A prototype may check restriction of other types, even those of a library for

which the source code is not available.[2] Note that annotation `concept` is used in the above example in a non-generic prototype.

The compile-time Cyan function `typeof(expr)` can be used with a parameter that is either a prototype or a message passing to a prototype. This last form can be used to retrieve the return type of a method, which can then be used in some statement.

```
1  @concept{*
2      typeof(R get) in [ typeof(S at: 0), typeof(T get) ],
3      S has [ func at: typeof(R get) -> Array<R>
4              func grab: typeof(R get) -> typeof(R collect)
5             ],
6      typeof(T get) is Int,
7        // same as above
8      typeof(T get) is typeof(0+1),
9      typeof(T with: Array<R>) has [
10         func size -> Int
11         func add: S
12         ],
13     typeof(T with: (R collect)) is Array<S>,
14     typeof(T get) is typeof(R get)
15 *}
16 object Twist<R, S, T>
17    // elided
18
19 end
```

Function `typeof` is used in several situations in this concept code. Line 2 of this example demands that the return type of method `get` of type parameter `R` has a type equal to one element of the list that follows, between `[` and `]`. Line 14 requires that the return types of methods `get` of `T` and `R` be equal.

Currently, there is no way of referring to the type of a parameter of a method in the concept language. Therefore, it is not possible, for example, to demand that the parameter type has some methods. This is a planned feature.

Metaobject `concept` generates prototypes for testing the coverage of the DSL code attached to the annotation. The problem to be solved is that the DSL code may not be enough for assuring there will be no error when instantiating the generic prototype. For example, the DSL code may demand that a real type argument has only the method

[2] Currenty, Cyan does not support libraries of compiled code (as jar files in Java). The source code of a library must be available and it must be compiled with the program.

`unit`. But inside the generic prototype, there could be a message passing "`inverse`" to an expression whose type is the formal parameter.

When annotation `concept` takes a parameter `test`, the metaobject creates, in the test directory of the project, a subdirectory with several Cyan prototypes. These are not related to the statements "axiom" of the DSL code. One of the prototypes declares a variable whose type is the generic prototype with real parameters whose names are equal to the formal parameters. In the `GroupList` example, the declaration would be

```
var main.GroupList<T> testVar;
```

The metaobject also creates a prototype for each formal parameter. Inside it, methods are added to match the restrictions of the DSL code of annotation `concept`. Therefore, for the `GroupList` example, the metaobject creates the prototype that follows.

```
object T
    func unit -> T = T;
    func * T tmp1155 -> T = T;
    func inverse -> T = T;

end
```

This prototype was built based on all requirements on the parameter `T` of the generic prototype. These demands may be spread in the DSL code of the annotation. Thus, if the generic prototype has two formal parameters `T` and `R` and the DSL code demands that `T` is superprototype of `R`, the test prototype for `R` would have the inheritance of `T` by `R`.

All prototypes just cited should be compiled. If there is a compilation error, probably the *concept* code does not cover the uses of the type parameters inside the generic prototype. For example, suppose that we remove the need for method `inverse` in the DSL code of `GroupList concept` annotation. When the tests are compiled, there would be a compilation error when message `inverse` is sent to an expression of type `T`. This is a clear sign that the `concept` code is missing some restriction.

2.2.2 Concept Implementation

The metaobject class of `concept` implements the MOP library interface `IParseWithCyanCompiler_parsing` for parsing the DSL code attached to the annotation. An AST is built and stored in a field of the metaobject class. Method

```
bsa_checkProgramUnit
```

inherited from interface `ICheckProgramUnit_bsa` and implemented by the metaobject class, does the semantic analysis of the expressions used in the DSL code. After that, it checks if the demands of each statement are satisfied. Hence, any errors issued by the DSL code is in fact issued by this method.

The metaobject class of `concept` implements two other interfaces:

```
IActionProgramUnitLater_parsing
IListAfter_afterResTypes
```

A method of the first interface is called to insert the metaobject in a list that will later be used by a method of the second interface, `after_afterResTypes_action`. If the `concept` annotation has a parameter `test`, method `after_afterResTypes_action` produces test prototypes for all the instantiated prototypes and for the generic prototype.

2.3 Metaobjects Coded in Interpreted Cyan

Several metaobjects take a Cyan-like language called *Myan* as the DSL attached to the annotations. *Myan* is interpreted Cyan with the addition of parameterless methods declared with `func`. Let us see an example.

```
package main

object Test
    func run {
        @action_afterResTypes_semAn{*
            func semAn_codeToAdd {
                "semAn_codeToAdd" println;
                return "#semAn_codeToAdd println; getZero println; ";
            }
            func afterResTypes_codeToAdd {
                "afterResTypes_codeToAdd" println;
                return [. """ func getZero -> Int = 0;
                          """,
                          "func getZero -> Int" .]
            }
        *}
        assert self getZero == 0;
    }
end
```

The class of metaobject `action_afterResTypes_semAn` implements the following interfaces:

```
IParseWithCyanCompiler_parsing,
IAction_afterResTypes,
IActionNewPrototypes_afterResTypes,
IAction_semAn,
```

```
IActionNewPrototypes_semAn ,
ICommunicateInPrototype_afterResTypes_semAn
```

In the *Myan* code attached to the annotation, there may be a method for each method of these interfaces, except `IParseWithCyanCompiler_parsing`. In the example, there is a method `semAn_codeToAdd` of interface `IAction_semAn` and a method

`afterResTypes_codeToAdd`

of interface `IAction_afterResTypes`. No method has parameters. They are implicitly declared. Therefore, inside both methods there is a `compiler` variable. In the first method, it has type `ICompiler_semAn`. In the second one, `ICompiler_afterResTypes`. Variables `metaobject` and `env` are also implicitely declared. They refer to the metaobject associated to the annotation and to the environment.

Every method has an implicitly declared signature that is equal to the corresponding method of the Cyan interfaces for the MOP (See section 1.2). In particular, the return type is equal to the corresponding method in the Cyan MOP. Therefore, method

`afterResTypes_codeToAdd`

should return an object of `cyan.lang.Tuple<String, String>` and

`semAn_codeToAdd`

should return an object of `cyan.lang.String`.

In this example, the afterResTypes method add to prototype `Test` a method `getZero`. The semAn method adds to the current location two statements. Both methods print, at compile-time, their names.

The `self` object of any *Myan* code has two methods:

a) `call:` that takes at least one and at most 11 parameters. The first one is the name of an action metaobject that should have been imported by the current compilation unit. "`call:`" calls method `eval` of this action metaobject passing as parameter a tuple object with many information like the environment and the other parameters to `call:`. For more details, see the documentation for metaobject `action_afterResTypes_semAn`;

b) `runFile:` that takes at least one and at most 11 parameters. The first one is the name of a file of a `--data` directory with extension `myan`. The file name may be preceded by a package name, otherwise it is considered to be in the current package. This file is loaded and the arguments to `runFile:` replace the file arguments. Then the file statements are interpreted.

Given this introduction to *Myan*, we cite other metaobjects that also use this language.

a) `onDeclaration_afterResTypes_semAn_afterSemAn`, that accepts methods from

all interfaces of `action_afterResTypes_semAn` plus those of interfaces `ICheckDeclaration` and
`IActionMethodMissing_semAn`.

b) `onDeclaration_afterSemAn`, that accepts methods from all interfaces of `action_afterRes` plus those of interface `ICheckDeclaration_afterSemAn`.

c) `onFieldAccess`, that accepts methods from all interfaces of `action_afterResTypes_semAn` plus those of interface `IActionFieldAccess_semAn`.

d) `onFieldMissing`, that accepts methods from all interfaces of `action_afterResTypes_semA` plus those of interface `IActionFieldMissing_semAn`.

e) `onMessageSend_semAn`, that accepts methods from all interfaces of `action_afterResTypes` plus those of interface `IActionMessageSend_semAn`.

f) `onMethodMissing`, that accepts methods from all interfaces of `action_afterResTypes_sem` plus those of interface `IActionMethodMissing_semAn`.

g) `onOverride_afterResTypes_semAn`, that accepts methods from all interfaces of `action_afterResTypes_semAn` plus those of interfaces

```
IActionMethodMissing_semAn
ICheckOverride_afterSemAn
```

h) `onSubprototype_afterResTypes_semAn_afterSemAn`, that accepts methods from all interfaces of
`action_afterResTypes_semAn` plus those of interfaces

```
ICheckSubprototype_afterSemAn
IActionMethodMissing_semAn
```

i) `onMessageSend_afterSemAn`, that accepts methods from the following interfaces:

```
ICheckMessageSend_afterSemAn
IAction_afterResTypes
ICommunicateInPrototype_afterResTypes_semAn
```

j) `onOverride` that accepts a list of Cyan statements in the attached DSL. These statements are considered the body of method `afterSemAn_checkOverride` of `ICheckOverride_afterSemAn`.

k) `onSubprototype` that accepts a list of Cyan statements in the attached DSL. These statements are considered the body of method

```
afterSemAn_checkSubprototype
```

of `ICheckSubprototype_afterSemAn`

l) `onVariableDeclaration`, that accepts methods from all interfaces of
`action_afterResTypes_semAn`
plus those of interface `IActionVariableDeclaration_semAn`.

As an example, Listing 2.3 shows a prototype `Person`. Annotation `onSubprototype` to `Person` demands any subprototype overrides method `printData`. Using the implicitly declared variable `subPrototype`, the Cyan code attached to the annotation looks for a method with this name. If it is not found, an error is issued with method `addError:` of the metaobject. Variable `subPrototype` corresponds to a parameter to method `afterSemAn_checkSubprototype` of interface `ICheckSubprototype_afterSemAn` implemented by the class of `onSubprototype`.

Annotation `onOverride` is attached to method `printData`. It just calls the action metaobject `shouldCallSuperMethod` defined in package `cyan.lang`. This metaobject checks whether the first statement of the overridden subprototype method calls the superprototype method.

The two annotations in `Person` have the semantics "`printData` should be overridden in any subprototype of `Person` and the subprototype method should call the superprototype method".

Metaobject `runFile` works like `action_afterResTypes_semAn` but the *Myan* code is read from the file that is the first parameter, which is optionally preceded by a package name. The file should have extension `myan` and be in the `--data` directory of the package.

```
@runFile_afterResTypes_semAn("runFile_afterResTypes_semAn.
   afterResTypes_semAn_test",
      "with:1 do:1", "unary", 10)
```

The Myan file can have parameters in its name. They are textually replaced by the parameters of the annotation (all of them but the first, which is the file name). Therefore, in this example, there should be a file

`afterResTypes_semAn_test(MetSig,UMS,Ret).myan`

in directory `--data` of package `runFile_afterResTypes_semAn`. The parameters between `(` and `)` in the first name can have any names. But since the annotation has *four* parameters, the file should have *three* parameters. The annotation parameters are also available inside the *Myan* code through the expression

```
metaobject getAnnotation getJavaParameterList
```

which returns an `List<Object>` or through

```
metaobject getAnnotation getRealParameterList
```

which returns an object of `List<WrExprAnyLiteral>`.

Listing 2.3 – Annotations with interpreted Cyan statements

```
package main

@onSubprototype{*
    var Boolean found = false;
    var methodList = subPrototype getMethodDecList: env;
    var Int size = methodList size;
    for i in 0..< size {
        var method = methodList get: i;
        if (method getName) equals: "printData" {
            found = true;
            break;
        }
    }
    if !found {
        metaobject addError:
            "'" ++ subPrototype getName ++
            "' is a subprototype of '" ++
            metaobject getAttachedDeclaration getName ++
            "' It should define a method 'printData'." ++
            " It does not";
    }
*}
open
object Person
    func init {
        name = "noname"
    }
    func run {
    }

    @onOverride{*
        call: #shouldCallSuperMethod;
    *}
    func printData {
        "name = $name" println
    }

    @property var String name
end
```

Listing 2.4 – Example with metaobject `runPastCode`

```
package main
import cyan.io

object TestRunPastCode

    func run {
        self testImportProgram;
        self testPastCode;
    }

    func fat: Int n -> Int {
        if n == 0 { return 1; }
        else { return n*(fat: n - 1); }
    }

    func testImportProgram {
        var String color = favoriteColor;
        var TestRunPastCode past = self;
        var Int f5 = 0;
        @runPastCode(true){*
            import main
            let TestRunPastCode p = TestRunPastCode new;
            var Int fatorial5 = p fat: 5;
            "At compile-time, factorial of 5 = " print;
            fatorial5 println;
            return "f5 = " ++ fatorial5 ++ ";");
        *}
        if f5 == 120 {
            "runPastCode was called with 'true'" println
        }
        else {
            "runPastCode was called with 'false'" println
        }
    }

    func testPastCode {
        var TestRunPastCode past = self;
        favoriteColor = "cyan";

        @runPastCode(true, past){*
            "Favorite color = " print;
            past getFavoriteColor println;
        *}
    }

    @property var String favoriteColor = "blue";
end
```

Annotations of metaobject `runPastCode` take an attached Cyan code that:

a) may import packages of the current program that were previously compiled;

b) can save objects of the last program execution to be used in future compilations.

Listing 2.4 shows these two features. Annotation of lines 20-27 import the package `main` if the first parameter to `runPastCode` is `true`. This results in a compilation error the first time the program is compiled because there is no compiled `main` package yet. There should be a successful compilation with the first parameter equal to `false`. In this case the attached Cyan code to the annotation is ignored. In an succeeding compilation, the parameter can be `true` as in this example. Then line 21 can import package `main` and use prototype `TestRunPastCode` in line 22. At compile-time we are using a previous version of the current prototype. In line 23, method `fat:` of `TestRunPastCode` is called and an assignment "`f5` = 120" is inserted in the source code. The `if` statement of lines 28-33 shows what happens when the parameter to the annotation is `true` or `false`.

In lines 40-43, annotation `runPastCode` has a second parameter, a variable name. Any number of parameters are allowed, they may be local variables or prototype fields. If the first annotation parameter is `false`, the metaobject will generate, for each variable or field, code that stores the object it refers to, serialized, in a special file of directory `--tmp` of the prototype. If the first parameter is changed, later on, to `true`, this object is read from disk and it can be used in the attached Cyan code. In this example, initially the first parameter to `runPastCode` is set to `false` and the program compiled. When the program is run, an object of `TestRunPastCode` is stored in disk. Field `favoriteColor` of this object is `"cyan"` because of line 38. If the first parameter is changed to `true` and the program compiled again, variable `past` in line 42 will refer to the object saved in disk. Its field `favoriteColor` will contain `"cyan"`, this all at compile-time. It does not matter if, in line 38, `favoriteColor` is initialized with another string.

Another metaobject worth to discuss is `insertCode`. The DSL attached to an annotation of `insertCode` is interpreted Cyan. The `self` object has two methods: `insert:1` and `insert:2`. If the annotation is used outside methods and inside a prototype, method `insert:2` can be used to insert fields and methods to the prototype (in phase after-ResTypes). The metaobject of the annotation of lines 1-8 insert three methods in the current prototype:

```
    func red -> Int = 0;
    func green -> Int = 1;
    func blue -> Int = 2;

1   @insertCode{*
2       var Int n = 0;
3       for elem in [ "red", "green", "blue" ] {
```

```
4           var String s = " func $elem -> Int; ";
5           insert: s, " func $elem -> Int = " ++ n ++ ";" ++ '\n';
6           n = n + 1
7       }
8   *}
9   func insertCodeTest {
10      assert red == 0 && green == 1 && blue == 2;
11
12      var Int fat5;
13      @insertCode{*
14          var p = 2;
15          for n in 3..5 {
16              p = p*n
17          }
18          insert: " fat5 = $p;" ++ '\n';
19      *}
20      "The factorial of 5 is $fat5" println;
21  }
```

If the annotation is used inside a method, the code given as parameter to `insert:1` is inserted after the annotation in phase semAn. The metaobject associated to the annotation in lines 13-19 insert the following statement after the annotation:

```
fat5 = 120;
```

Discussion on Metaobjects in Cyan

Software documentation describes a great deal of semantic restrictions that are not expressed in the code. For example, the documentation of `Person` of Listing 2.3 would describe that method `printData` should be overridden in any subprototype and that the subprototype method should *extends* the superprototype method; that is, its first statement should be a call to the superprototype method. However, without metaprogramming it is not possible to express these restrictions in the code itself, as in the attached code of the annotations of `Person`. Metaobjects can check that:

a) the parameters to a message passing are related to each other in a way described by the method documentation. For example, the first parameter to method[3] `printf:` of prototype `Out` should be a literal string that matches the other parameters;

[3] It is a virtual method. The real method is `printfAll:`.

b) a field is accessed by only a selected set of methods or a private method is only called by some methods;

c) a Cyan interface is only implemented by subprototypes of a given prototype;[4]

d) a prototype is inherit only by some classes;

e) a method of a prototype is only overridden in some prototypes.

f) a object of a prototype is correctly initialized;

g) tests for a method or prototype are always generated;

h) in a subprototype some sets of methods are always overridden together: none or all of the set;

i) only some prototypes create objects of another prototypes;

j) a method of a prototype always calls another method of the same or another prototype;

k) a method of a prototype neves calls a given method of the same or another prototype;

l) no prototype of a given package import another specific package;

m) a test method should be added to a prototype after a day;

n) every prototype has a documentation with a minimum of `Np` words and that every public method has a documentation of at least `Nm` words.

Metaobjects can also generate code whose usefullness have been previously discussed. Checks and code generation are speeded up with the use of *Myan* and interpreted Cyan[5] in metaobjects. But their importance is greater than that. *Myan* and interpreted Cyan allows to fuse the program and the metaprogram in the same text file, making it easy to express the semantics of the program in itself, outside of the documentation.

2.4 Metaobjects in Cyan Libraries

Metaobjects are largely used in the package `cyan.lang` whose prototypes are tighly integrated with the language. This is unlike other languages, like C (KERNIGHAN, 1988), in which the compiler knows very little about the standard libraries. The compiler interacts with package `cyan.lang` because of several features of the language:

a) literals of a basic types such as `1` or `"Hi!"` are objects of prototypes of `cyan.lang`;

b) literal arrays have types that are instantiations of generic prototype `Array`, literal tuples are instantiations of `Tuple`, and literal maps of `HashMap`;

[4] Language Hack (HACK, 2020) has such a feature, but in the language itself.

[5] *Myan* allows the declaration of methods, interpreted Cyan allows only statements.

c) the `for` statement demands that the type of the expression following "`in`" implements an instantiation of interface `Iterable`;

d) there are no *shared* methods in Cyan, which would correspond to `static` methods of Java, C♯, and C++. They can be simulated with metaobject `prototypeCallOnly` that is in `cyan.lang`;

e) the exception handling system is implemented only with library objects. The prototype of an exception object should inherit from `CyException` and exception treatment is made with `catch:` methods of prototype `Function<Nil>`. All of these methods are in package `cyan.lang`, which also keeps some metaobjects used in `Function<Nil>` to check the real arguments of `catch:` message sends.

Metaobjects are used largely in prototypes of package `cyan.lang` and, since the language itself depends on this package, the language also depends on metaobjects. A dependency list follows.

a) Prototypes of basic types use metaobjects to communicate with their array types. The prototypes inject methods in their arrays. For example, prototype `Int` injects into `Array<Int>` the method

```
func sum -> Int
```

to sum the array elements.

b) A metaobject inserts a method `sort` into `Array<T>` if `T` defines the spaceship operator, `<=>`.

c) Metaobjects create methods for `Function<...>`, `Tuple<...>`, and `Union<...>` with any number of parameters. As an example, a method `==` is added to every `Tuple<...>` prototype based on the `==` methods of the tuple elements. These prototypes are not native to the language, they belong to package `cyan.lang`. A literal array, tuple, or union is a special object that inherits from the adequate prototype.

d) Methods `eq:` and `neq:` can only be defined in `Any` and the basic types. This is checked by a metaobject.

e) A do-not-use-it metaobject `javacode` is extensively used in package `cyan.lang` for injecting Java code inside Cyan. This was needed before the interoperability between the two languages.

f) Method `isA:` of prototype `Any` tests whether the receiver is an object of the argument, that should be a prototype. A metaobject checks if the argument is really a prototype. This method is the equivalent of `instanceof` of Java and `is` of other languages such as C♯ and Kotlin.

g) Three annotations are attached to method == of Any. One checks if the argument is compatible to the receiver. Hence, `0 == "zero"` results in a compile-time error. Other metaobject demands that, if == is overridden in a subprototype, `hashCode` is too. The metaobject of the third annotation generates test cases if the method is overridden and variable `testOverride` of the project is `true`.

h) Method `functionForMethod:` of Any takes the name of a method `M` as parameter, a string, and return an anonymous functions with the same parameters as `M`. The function calls `M`.

```
func functionForMethodTest {
      // the same as to call 'to: 0'
    assert (self functionForMethod: "to:1")
       eval: 0 == "n = 0";
}

func to: Int n -> String = "n = $n";
```

Message passings `functionForMethod:` are intercepted by a metaobject that create the functions at compile-time. The end result is that Cyan methods can be used as if they were regular functions.

i) Inline methods in Cyan are *simulated* with metaobject `replaceCallBy`, the language itself does not support it.

j) Annotation `genericPrototypeInstantiationInfo` is inserted at the beginning of every prototype instantiated from a generic prototype. Its parameters have information on the instantiation that caused the creation of the generic prototype instantiation. Therefore, if there is a compilation error in the newly-created prototype, the compiler points which caused it. For example, suppose a generic prototype `B` is instantiated in prototype `A`. `B` in its turn instantiates a prototype `C`. There is a compilation error in `C`. The compiler issues the error for the code in `C` and show the stack of generic prototype instantiations: `C`, `B`, and `A`. For each one, the file, line number, and column number of the instantiation are shown.

This section did not present any research material. But it showed that metaobjects are powerful enough to replace several constructs that otherwise should be part of the language (most of the items above).

3 Related Works

This chapter presents support for metaprogramming in several languages. Metacode can represent and generate base code in several ways, which is explained in section 3.1. Section 3.2 explains runtime metaprogramming and describes languages that support it. Section 3.3 is on metaprogramming made at compile-time, which can be made using Metaobject Protocols, Lisp-like quotations, and other mechanisms that are mainly employed by newer languages.

3.1 How Code is Generated and Represented

Metaprograms can handle and generated code using several representations (SMARAGDAKIS; BIBOUDIS; FOURTOUNIS, 2015), described next.

3.1.1 As text

Code is represented as strings. Therefore, any base code passed on to metacode is in string format, as is any code generated by metacode. Since strings do not keep any extra code information, the generated code may have lexical, syntactic, and semantic errors. This mechanism is used by Cyan for code generation as exemplified by metaobject `property2` of Appendix B.

3.1.2 Handling of the program Abstract Syntax Tree

Code is represented as AST objects if the compiler is implemented using an object-oriented language. Therefore, code generation is the creation of AST objects that represent it. There are some drawbacks of this approach. The developer has to understand countless AST classes (In Cyan, there are more than one hundred AST classes) and code generation is difficult because the mapping of code to AST creation is not trivial.

The benefit of this mechanism is that the generated code, an AST object, usually does not have syntactical errors.[1] However, they may be semantic errors that will be caught by the compiler later on. A drawback of this approach, besides its complexity, is that error messages are not clear because the compiler does not keep information about the metacode that created the generated code. Therefore, in all metaprogramming systems we are aware of, the compiler will not be able to pinpoint exactly who caused a problem. Cyan is able to give precise error messages because the compiler tracks which metaobject

[1] This is not true in Cyan: one can create an AST object representing a syntactical illegal expression with a unary `*`; for example, "`*2`".

added which code. If the generated code has an error, the compiler will issue the error and indicate the annotation associated to the metaobject that produced the ofending code. The message will cite the line, name, and file of the annotation that caused the error. In Cyan, errors are caught in the runtime of the metaprogram, which is the compile-time of the base program.

3.1.3 Quoting

Quoting is a syntax mechanism for automatically transforming text into AST objects. This construction is supported by many languages: Nemerle (NEMERLE, 2018), Scala (ODERSKY LEX SPOON, 2016) (and possibly in the new Scala version, Scala 3 (SCALA. . . , 2022)), Converge (TRATT, 2005), Elixir (ELIXIR, 2018), MetaOCaml (CZARNECKI et al., 2004), Template Haskell (SHEARD; JONES, 2002), and F♯ (PICKERING; EASON, 2016), to cite a few of them. Quotations in Lisp are detailed later in subsection 3.1.4 in the context of macros.

In F♯, quoted code is a text put between `<@` and `@>`. It evaluates to an AST object. Hence,

```
<@ n @>
```

evaluates to an AST object representing a variable `n`. An AST object can be inserted into a quoted text using %. This operation is called *splicing*. Therefore, variable `mixThem` is the AST of "`n + p`" (without the quotes!).

```
let n = 1
let p = 3
let quoteN = <@ n @>
let quoteP = <@ p @>
let mixThem = <@ %quoteN + %quoteP @>

printfn "%A" mixThem
```

The execution of this code produces

```
Call ( None, op_Addition,
       [ PropertyGet (None, n, []),
         PropertyGet (None, p, [])  ] )
```

This is the representation of the AST of the quoted text, "`n + p`". `Call` represents the call to function `+` and `PropertyGet` represents the getting of the values of local variables. `quoteN` and `quoteP` have types `Quotations.Expr<int>`. This permits that the compiler points any type errors when using quotes and splices.

```
let s = "ola"
```

```
let quoteS = <@ s @>
let er = <@ %quoteS * %quoteS @>
```

The last line causes an error:

```
/home/jdoodle.fs(39,24): error FS0001: The type
     'string' does not support the operator '*'
```

In F♯, untyped versions of quoted code exist, `<@@ ... @@>`. Untyped splicing is made with `%%expr`.

In the Python-based language Converge (TRATT, 2005), *quasi-quoted* code is given between `[|` and `|]` as in

```
[|  code   |]
```

Inside the text `code`, there may be annotation

```
${ otherCode }
```

In the generation of the AST object, `otherCode` is inserted into `code`. A text `anotherCode` may be evaluated at compile-time using annotation

```
$<  anotherCode >
```

It generates an AST object. It is as if the result of the evaluation were inserted in the source code. Variables are renamed to ensure that there is no unintended variable capture from the environment.

An example of quasi-quotes in Converge, taken from (LILIS; SAVIDIS, 2015), follows.

```
1    code := [| 5 |]
2    square := [| ${code} * ${code} |]
3    result := $<square>   // 25
```

In line 1, "`[| 5 |]`" creates the AST of `5` which is assigned to `code`. In line 2, `code` is embedded in a quasi-quote used to build the AST of `5 * 5` which is assigned to `square`. In line 3, `square` is evaluated at compile-time and the result, `25`, assigned to `result`. Therefore, the last line could be replaced by

```
result := 25
```

When the compiler finds `${code}`, it transforms `code` into a string that is inserted into the larger text, that inside `[| ... |]`.

Insertion like `${code}` is just string embedding in Cyan. The previous Converge example, in Cyan, is:

```
1    var String code = "5";
2    var String square = "return $code * $code";
3    var result = CyanInterpreter eval: square;
```

```
4     cast Int r = result {
5         r println;
6     }
```

Inside a literal Cyan string, there may appear a variable preceded by $. The variable value is inserted into the string at runtime. Therefore,

```
"return $code * $code"
```

is equivalent to

```
code ++ " * " ++ code
```

In line 3 of the Converge example, `square` is evaluated at compile-time. In Cyan, this demands the calling of the Cyan interpreter (line 3). The The return type of method `eval:` of prototype `CyanInterpreter` is an union type. The returned value needs to be cast in the last statement to `Int`. The above code could be in a base program or in a metaobject. The Cyan interpreter is called at runtime of the code. To evaluate a string at compile-time, use metaobject `eval`.

```
// the same as
// var value = 25;
var value = @eval("cyan.lang", "Int"){*
        return 5*5
    *};
```

Tratt shows an example in Converge, adapted from Template Haskell (CZARNECKI et al., 2004), that creates power functions at compile-time. Each function is tailored to a specific number of multiplications.

```
1  func expand_power(n, x):
2    if n == 0:
3      return [| 1 |]
4    else:
5      return [| ${x} * ${expand_power(n - 1, x)} |]
6
7  func mk_power(n):
8    return [|
9      func (x):
10       return ${expand_power(n, [| x |])}
11     |]
12
13 power3 := $<mk_power(3)>
```

`mk_power(3)` in the last line is evaluated at compile-time. The function return value is the AST object of the function of lines 9-10. Therefore, `power3` will refer to this function,

which is

```
func (x):
  return x * x * x * 1
```

`expand_power` and `mk_power` are normal functions that, because of `$< ... >` and `${ ... }`, are called at compile-time. Function `expand_power` is called in line 10 because of annotation `${ ... }`.

If the intention of the developer is to generate code, as in the `mk_power` example, she or he can use the `insertCode` metaobject for that.

```
@insertCode{*
    var body = " = x";
    var sig = " func power_5: Int x -> Int ";
    for n in 2..5 {
        body = body ++ "*x";
    }
    insert: sig ++ ";", sig ++ body ++ " ; ";
*}
func makePowerTest {
    assert power_5: 2 == 32;
}
```

The metaobject inserts into the current prototype the method

```
func power_5: Int x -> Int  = x*x*x*x*x ;
```

MetaOCaml is an extension of the functional language OCaml that supports *runtime* metaprogramming with quotations. Code may be produced at runtime and treated as data. A function, for example, can be built at runtime and inserted in the environment. Czarnecki et al. give a function `eval'` that generates quoted code for representations of First-Order Logic sentences. The call of this function returns quoted code that can be inserted in the current environment with the `.!` operator.

```
let eval expr env = .! (eval' expr env);
```

Quasi-quoted code may be ambiguous because the language may use the same grammar rule to mean different things. Using the Cyan syntax, a quasi-quote with the contents "`var Int n;`" would be ambiguous because it could represent the declaration of a field or a local variable. This is addressed by some metaprogramming systems by supplying several different kinds of quasi-quotes (BATORY; LOFASO; SMARAGDAKIS, 1998) (TRATT, 2008). There is no ambiguity in Cyan because the generated code is

represented as strings. Therefore, `"var Int n;"` becomes a field if it is inserted outside a method, in phase afterResTypes, or a local variable if it is inserted in phase semAn.

Quasi-quotes are, therefore, simulated with string handling in Cyan, which is much simpler for two reasons.

a) It uses regular Cyan string handling;

b) The base code and the metacode are clearly distinguished. The generated base code is always represented as strings.

However, errors in the generated code are only caught by the compiler in the next compilation phase. In a language supporting quase-quotation, some errors are caught when a quase-quote is evaluated at compile-time.

In Cyan, the code snippets produced by metaobject methods, which are just strings, are not checked when the method is running. Thus, the code below is perfectly valid in Cyan, even considering `dec` is returned as the code generated by the metaobject method.

```
var String partial = "var Int n =";
var String dec = partial ++ " 0;";
```

In languages that use quasi-quotes, the equivalent code would cause a parsing error in line 1 because the literal string would be represented using quasi-quotes and the compiler would check if this is a valid statement or expression. It is not because the expression assigned to `n` is missing.

3.1.4 Macros

Macros mean different concepts in different languages. In this paper, we assume the Lisp definition of macros: a function called at compile-time[2] whose return value, a code, replaces the macro call. The first high-level language to support macros was Lisp 1.5 (HART, 1963) in 1963. To explain Lisp macros, it is first necessary to give a general view of the language and explain how it support quotations.

Lisp (BARSKI, 2010) (SEIBEL, 2012) represents both data and programs using the same structure, lists. A list `(f a b c)` is the call of function `f` with arguments `a`, `b`, and `c`. Unless `f` is a special form which uses special syntax and evaluation rules.

```
(progn (print "everything")
       (print "is a list")
       (print "in Lisp"))
```

[2] Here, "compile-time" means "when the code is compiled". In Lisp, code that may contain macros may be created and executed at runtime.

In this example, `progn` is a special form that evaluates its arguments in the order they appear in the list. `print` is a function that prints its argument. A list that is not to be evaluated should be preceded by ' or `.

```
`(1 2 3)
```

Here, `1` is not considered a function. Operator ` is called *quasiquote* and ' is called *quote*.

A value is inserted in a quoted list using the comma, also called the *unquote* operator. This is not valid, in Common Lisp (BARSKI, 2010), if ' is used.

```
(let ((y 3)) (print `(+ ,y 5)))
```

This code prints

(+ 3 5)

because `3` is assigned to `y` and `,y` is replaced by `3`. After `,` there may be a list, which is evaluated.

```
(let ((y 3)) (print `(+ ,(+ y 2) 5)))
```

The expression printed is

```
(+ 5 5)
```

Lisp macros make extensive use of quasiquotes, ` , comma (,), and `,@` operators. The first three were explained in Subsection 3.2.1. Operator `,@` expands the list that comes after it. That is, the list is inserted in the quoted code without the parentheses.

```
(let ((bd `(BB CC))) (print `(AA ,@bd ,bd) ))
```

The above code prints

```
(AA BB CC (BB CC))
```

In the next example, a macro `repeat-until` takes a condition and a sequence of elements collected with `body`. `defmacro` defines a macro whose parameters are `condition` and a list `body`. A call to the macro is replaced by the quoted list in the second line.

```
(defmacro repeat-until (condition &rest body)
  `(loop while (not ,condition) do (progn ,@body) ) )
```

The last three statements of the following code print the values from `0` to `9`. `let` declares and initializes a list of local variables, `setf` assigns an expression to a variable, and `progn` takes a list of statements and executes them.

```
(let ((i 0))
    (repeat-until (>= i 10)
        (progn (print i) (setf i (+ i 1))) ) )
```

Currently, languages Nemerle (SKALSKI; MOSKAL; OLSZTA, 2005), Rust (KLABNIK; NICHOLS, 2022), and Scala (BURMAKO, 2013) supports macros that are more powerful than those of Lisp.

Skalski, Moskal, and Olszta (SKALSKI; MOSKAL; OLSZTA, 2005) give an example of a `for` statement added to Nemerle using a macro. The macro defines the syntax and how code is to be generated to a `for` statement. Quasi-quotes are used to express the generated code.

```
macro for (init, cond, change, body)
syntax ("for", "(", init, ";", cond, ";", change, ")", body)
{
    // generate code here using quasi-quotes
}
```

In Scala (BURMAKO, 2018), quotes are used in macros. A macro is implemented by a method that should return an expression whose type is an AST type.[3] The functional language Template Haskell allows code generation at compile-time using quotations and splicing. An example, given by Sheard and Jones (SHEARD; JONES, 2002), creates a `zip3` function:

```
zip3 = $(zipN 3)
```

`zipN` is a function that produces quoted code and splicing is made with `$( ... )` in Template Haskell. Code that calls `zip3` should be compiled after `zip3` and `zipN`. This is the usual requirement for statically-typed languages.

Macros are much more limited than Cyan metaobjects. They cannot :

a) intercept operations such as message passing, field access and inheritance. Or errors such as field and method missing. They can only produce code locally;

b) check a source file as metaobjects do;

c) be attached to classes, methods, and fields for producing code or check those entities;

d) rename methods.

Cyan does support *macros* which are metaobjects with most of the power of other kinds of metaobjects. Cyan macros can implement almost any interface that acts until phase semAn of the compilation. For example, a macro call can create new prototypes, add fields and methods to the prototype it is used, communicate with other metaobjects,

[3] Note that the AST type here may not be the same as the type of the Scala compiler. It may be an interface with a more limited view of the objects or another class that wraps the original compiler AST class.

and so on. A macro class cannot implement interfaces whose methods are called in phase afterSemAn. However, a macro metaobject can generate code that contains annotations that act in phase afterSemAn. Cyan macros are difficult to build compared with macros of other languages because there is no DSL for macro definition as in Rust (see Appendix D of (KLABNIK; NICHOLS, 2022)). Cyan macros are not further discussed in this text.

Macros of Nemerle, Rust, and Scala can be roughly simulated in Cyan by metaobjects whose prototypes implement interface

```
IActionMethodMissing_semAn
```

If the compiler cannot find an adequate method, a metaobject method is called. It can then apply any transformation to the message passing parameters and produce any code. Just like a macro whose syntax is that of a message passing. Metaobject `grammarMethod` presented in section 2.1 does just that.

3.1.5 Generic classes, functions, and prototypes

A generic prototype in Cyan, described in section A.5, is a prototype that accepts parameters and, for each set of parameters, a new prototype is created by the compiler. As an example, prototype `Stack` of package `cyan.lang` is a generic prototype that accepts one parameter. `Stack<Int>` and `Stack<String>` are called *instantiations* of `Stack` and they are two different prototypes that do not share any code. A generic prototype works as a compile-time function that returns a new prototype for each set of parameters. Therefore, generics are a code generation mechanism. This is not the case with generics in Java and Scala. All instantiations of a class share the same code.

Generics are called templates in C++ (STROUSTRUP, 2013). As in Cyan, a new class or function is created for each new set of parameters. C++ templates use a functional Turing-complete language for template generation (VELDHUIZEN, 2003). The same functionality is offered by metaobject `insertCode` of section 2.3. Code is generated by a subset of Cyan itself interpreted at compile-time. The benefit of using a metaobject like `insertCode` is that the language, a subset of Cyan, is much simpler than the C++ template language.

3.2 Runtime Metaprogramming

The definition of runtime metaprogramming (RTMP) that we use is "the handling of a program by itself" at runtime. RTMP is usually called *reflection* (DEMERS; MALENFANT, 1995) (BOBROW; GABRIEL; WHITE, 1993), the ability of a program treat itself as data in two different ways, introspection and intercession. To introspect is to observe itself, the program knows part of itself at runtime. Intercession is the ability of a program

to change part of itself at runtime. *Reification* is the encoding of the program as data that can be handled at runtime.

Most of the fundamental work on reflection was made by Smith (SMITH, 1984) in its seminal article which is properly explained by Herzeel, Costanza, and D'Hondt (HERZEEL; COSTANZA; D'HONDT, 2008). Smith categorizes reflection into *structural reflection* and *procedural reflection*, nowadays called *behavioral reflection* (ORTIN; REDONDO; PEREZ-SCHOFIELD, 2009). *Structural reflection* treats of the structure of the program. In an object-oriented program, it would be the inheritance hierarchy, the methods and fields of each class, the statements of each method, etc. *Behavioral reflection* is about to exam and change the execution of a program. For example, intercepting a message passing and making it call a method chosen at runtime, not the method that would normally be called.

There are four kinds of combinations between introspect/intercede and structural/behavioral. A Ruby (FLANAGAN; MATSUMOTO, 2008) program may add a field to a class at runtime (intercede/structural). In CLOS, the method dispatch mechanism[4] may be changed at runtime (intercede/behavioral). It is possible to know the methods of a Java class (introspect/structural) at runtime. In Ruby, it is possible to know if a message may be sent to an object without causing runtime errors (introspect/behavioral).

A list of object-oriented languages supporting *introspective reflection* includes Smalltalk (GOLDBERG; ROBSON, 1983), Self (UNGAR; SMITH, 1987), Java (GOSLING et al., 2014), C♯ (C#..., 2020), Groovy (KöNIG, 2007), Scala (**??**), Kotlin (JETBRAINS, 2022), Ruby (FLANAGAN; MATSUMOTO, 2008), EcmaScript (ECMA International, 2011), and Python (SUMMERFIELD, 2009). The introspective abilities of Java are shown in the following example.

```
1  public class Program {
2
3      final public void run() {
4          Method methodList[] = this.getClass()
5              .getDeclaredMethods();
6          for ( Method m : methodList ) {
7              System.out.print(m.getName() + " ");
8          }
9          System.out.println("");
10         Field fieldList[] = this.getClass()
11             .getDeclaredFields();
12         for ( Field f : fieldList ) {
13             System.out.print(f.getName() + " ");
```

[4] Which method is called for this message passing?

```
14          }
15      }
16
17      String to(int n) {
18          return "" + n;
19      }
20
21      int count = 0;
22      String name = "Program";
23
24  }
```

This code prints the name of all methods and all fields declared in the current class. `this` is the object that received the message. "`this.getClass()`" in lines 4 and 10 returns an object of class `Class`[5] that describes the class of the receiver, `Program` in this example. `Method` and `Field` are classes that describe a Java method and field respectively.

In lines 5 and 6 of the following example, a `Method` object representing method "`to`" is assigned to variable `method`. The object representing the parameter type, `int.class`, is passed as argument to `getDeclaredMethod`. In line 8, method "`to`" is called using method `invoke` of class `Method`.

```
1  public class Behavior {
2
3      final public void runBehavioral() {
4          try {
5              Method method = this.getClass()
6                  .getDeclaredMethod("to", int.class );
7              System.out.println(
8                      method.invoke(this, 10) );
9          }
10         /*
11             getDeclaredMethod and invoke can throw
12             a lot of exceptions
13         */
14         catch ( NoSuchMethodException | SecurityException |
15                 IllegalAccessException |
16                 IllegalArgumentException |
17                 InvocationTargetException e) {
18             e.printStackTrace();
```

[5] In reality, "`Class<? extends Program>`", since the receiver of message `getClass()` is `Program`.

```
19          }
20      }
21
22      String to(int n) {
23          return "" + n;
24      }
25  }
```

The next Subsections explain how runtime metaprogramming is supported by some other languages of academic interest.

3.2.1 Lisp and its Dialects

To our knowledge, runtime metaprogramming in a high-level language arose in Lisp with the `eval` function (MCCARTHY, 1981). `eval` is a Lisp interpreter that evaluates its argument, an expression.

```
(print (eval `(+ 22 11) ))
```

The backquote symbol was used to pass a list as argument to `eval`. Without the backquote, `33` would be the `eval` argument which would be returned. `eval` can be used for adding functions and even macros to the current environment.

Lisp function `apply` takes a quoted function identifier and a list of quoted arguments. `apply` calls the function with the arguments:

```
(let ( (plus '+) )
   (print (apply plus '(1 2))) )
```

This code associates the quoted `+` to variable `plus` and prints the call of this function with arguments `1 2`. The value printed is `3` as expected. This same goal was achieved with method `invoke` of class `Method` inside the Java class `Behavior` of page 107.

Language 3-Lisp (SMITH, 1984) (HERZEEL; COSTANZA; D'HONDT, 2008) is implemented with an infinite tower of interpreters. The user program is at level 0, interpreted by a Reflective Processor Program (RPP) (RIVIèRES; SMITH, 1984) at level 1. Each RPP at level n is interpreted by a RPP at level $n + 1$. There are interactions between the levels. Herzeel, Costanza, and D'Hondt (HERZEEL; COSTANZA; D'HONDT, 2008) give examples of reflection:

a) a function `print-infix` takes the internal representation of an expression and prints it in the infix form. The call to `print-infix` should use operator ' that returns the internal representation of an expression, which is at a level above the current level (structural reflection);

b) a function `advise-before` adds a statement, its second parameter, to the function that is its first parameter (structural reflection);

c) a macro-like function `when` that simulates an `if` expression without the `else` part (behavioral reflection).

3.2.2 Smalltalk

Smalltalk (GOLDBERG; ROBSON, 1983) (NIERSTRASZ; DUCASSE; POLLET, 2009) was the first object-oriented language to have an extensive support for runtime metaprogramming. Introspective reflection is implemented through metaclasses, the classes of classes. Almost everything in Smalltalk is an object, and every object is an instance of a class. A class is also an object, instance of its metaclass. For each class there is exactly one unnamed metaclass. There is an inheritance hierarchy among metaclasses that is parallel to the class hierarchy. That is, if `Student` inherits from `Person`, the metaclass of `Student` inherits from the metaclass of `Person`. That can be seen in the next example in which methods `super` and `superclass` are called. `Student class` is the sending of message `class` to object `Student` — the metaclass of `Student` is returned. `Student superclass` returns the superclass of `Student`, `Person`. Method `==` returns `True` if two objects are the same. This comparison returns `True`.

```
Student class superclass == Student superclass class
```

A metaclass is also an object, instance of class `Metaclass`. That means the object returned by the message send below is an instance of `Metaclass`.

```
Metaclass class
```

The Smalltalk class hierarchy employs several twists in order to accommodate the design goals of “classes are objects” and “classes have metaclasses”. One of them was just cited. The other is that the class of the top of the hierarchy is `Object`, which does not inherit from any other class. But the `Object` class inherits, indirectly, from `Object`. That is, the metaclass of `Object` inherits from `Object` and the following code results in `True`.

```
Object class inheritsFrom: Object
```

Method `inheritsFrom:` returns `True` if the receiver is a class that inherits from the argument. An excellent explanation of the Smalltalk hierarchy is given by Nierstrasz, Ducasse, and Pollet (NIERSTRASZ; DUCASSE; POLLET, 2009). Figure 11 show the inheritance hierarchy of classes `Person`, `Student`, and some fundamental classes for reflection. Figure 12 shows the instance-of relationships among the classes of Figure 11. In both drawings, classes are represented as rectangles and metaclasses as dashed rounded rectangles.

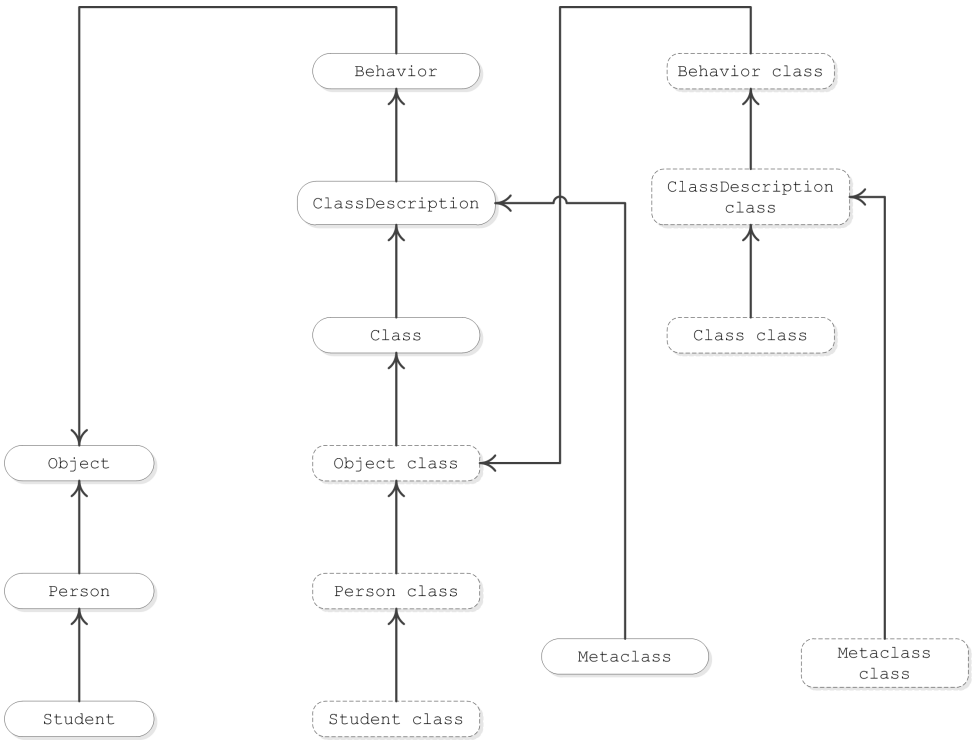

Figure 11 – Inheritance hierarchy in Smalltalk

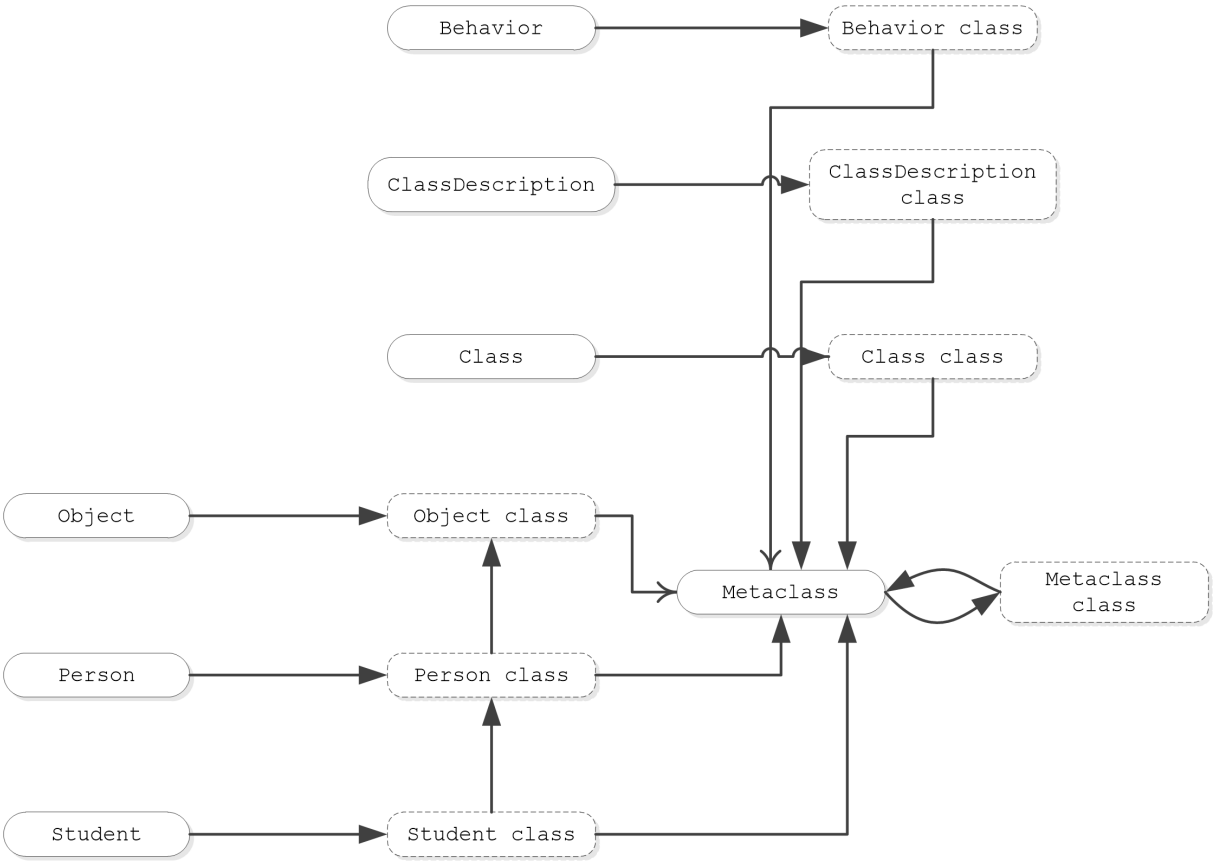

Figure 12 – Instance-of relationships in Smalltalk

Listing 3.1 – Insertion of methods in a class in Smalltalk

```
1  SmallInteger
2      compile:
3        'printInt [ Transcript show: ''an integer'' ]'.
4
5  5 printInt.
6
7     "can 5 understand message '+' ? true is printed."
8  (5 class canUnderstand: #+ ) printOn: stdout.
9
10    "prints 'true' "
11    "'perform' calls a method with the given"
12    " name and arguments"
13 ((5 perform: #+ with: 3) = 8 ) printOn: stdout.
14
15
16 SmallInteger removeSelector: #printInt.".
17
18   "runtime type error"
19 5 printInt.
```

The metaclass system of Smalltalk is for introspective reflection only. It is possible to discover the superclass of a class, its instance variable and methods, its metaclass, and so on. Intercession can be made with methods inherited from fundamental classes such as `Behavior`. For example, it is possible to compile a method given as string and insert it into a class using method `compile:` of `Behavior`. In the code of Listing 3.1, a method `printInt` is added to class `SmallInteger`.[6] Then it is removed, which causes a runtime type error in the last message passing. Quotes, `"` are used for comments and `#` starts a literal of `Symbol`, a subclass of `String`.

As another example, the following code changes the instance variables of a point through reflection. `instVarAt:put:` sets a new value for an instance variable whose number is given as the first parameter. That is, the instance variables are accessed as an array.

```
| p |                                "local variable declaration"
p := 5@5.                          "5@5 is a point"

p printOn: stdout.                 "print p in stdout"
Transcript show: ' '.              "print a white space"
p instVarAt: 1 put: 0.
p printOn: stdout.
Transcript show: ' '.
```

[6] This code was tested in site https://www.jdoodle.com/execute-smalltalk-online

```
p instVarAt: 2 put: 1.
p printOn: stdout.
```

This code prints

```
5@5 0@5 0@1
```

Method `doesNotUnderstand:` is defined in class `Object` and therefore is inherited by every Smalltalk class. It is called whenever a message is sent to an object that does not have an appropriate method. It can be redefined to implement several features (FOOTE; JOHNSON, 1989) such as dynamic fields, multiple views of an object (depending on the context, `doesNotUnderstand:` calls another method or not), simulation of the addition of methods at runtime, and persistent objects (`doesNotUnderstand:` does the persistent tasks before forwarding the message to another object or calling another method).

Method `doesNotUnderstand:` allows one to interfere in the method dispatch system (which method is called for a message?). Therefore it is one mechanism of the language for behavioral reflection. The class and metaclass hierarchy of Smalltalk and the many methods for introspection compose a Metaobject Protocol (MOP). A MOP is composed by the support classes like `Behavior` and the description of the methods that allow a program to view itself. Although a Smalltalk program can change itself (e.g. add a method to a class), these changes are not made by the MOP.

Many object-oriented languages offer support for runtime metaprogramming, some of them comparable to Smalltalk, like Groovy and Ruby. In these languages, objects can be inspected, methods may be added to classes at runtime (or objects), and there is a method with the same purpose as `doesNotUnderstand:` of Smalltalk. In particular, language Self (UNGAR; SMITH, 1987) was built to be highly dynamic. At runtime, almost everything is allowed, including changing the inheritance hierarchy.

3.2.3 3-KRS

Maes (MAES, 1987a) (MAES, 1987c) introduced the concept of runtime metaobjects in 3-KRS, a Lisp-based language. Each object is associated with a metaobject that not only has reflective information on it but also controls it. There is a one-to-one relationship between an object and its metaobject, although these are constructed in a lazy way, only when needed. This is necessary because classes, fields, methods, messages, and metaobjects are objects. Laziness is necessary in order to prevent an infinite regress.

A metaobject may interfere in the method dispatch algorithm and inheritance. Maes (MAES, 1987b) gives an example of a trace metaobject that can be attached to an object in order to intercept message passings. Actions may be added before and after the actual method is called. Another metaobject may be used for implementing multiple inheritance in 3-KRS (MAES, 1987c).

3-KRS separates explicitly the level of regular objects and the metalevel. There is a clear protocol for communication between metaobjects and objects. Some standard metaobjects are responsible for acting when no user-defined metaobject is available. These standard metaobjects implement the original semantics of the language.

The Metaobject Protocol of 3-KRS is composed by metaobjects that implement the original system behavior, their methods, and the interface between objects and metaobjects (how one can interact with the other).

In language Green (GUIMARÃES, 1998), a restricted kind of metaobject, only for intercepting message passing, can be efficiently implemented even in statically typed languages, with zero overhead. This is achieved by changing the method table of a single object. This table has pointers to the object methods and usually is shared among all objects of a class. But it does not need to be so. A copy of the class table may be changed, effectively introducing new methods or replacing existing methods.

3.2.4 The CLOS Metaobject Protocol

CLOS (KICZALES; RIVIèRES; BOBROW, 1991) (KICZALES et al., 1993) (PAEPCKE, 1993) (BOBROW; GABRIEL; WHITE, 1993) (DEMICHIEL; GABRIEL, 1987) is the extension of Common Lisp (SEIBEL, 2012) with features for object-oriented programming. The language has many particularities that are not commonly found in the most used object-oriented languages. We show the smallest level of details necessary to understand the CLOS Metaobject Protocol:

a) methods are defined outside classes. In the example of Listing 3.2, class `Person` is between lines 6-10 and method `pretty-print` is in lines 12-15. A method is called as a function, in line 26 method `pretty-print` is called with argument `meg`, a global variable defined in line 19. The usual syntax for line 26 would be "`meg.prettyPrint()`";

b) method dispatch depends on all method arguments. In most languages, the method is searched for in the class of the receiver of the message, which is the class of `meg` in "`meg.prettyPrint()`". In CLOS, the runtime system builds a list of method that may be called, based on all method arguments, order it (the most specific according to inheritance comes first), and calls the most specific method. The list of methods with the same name is called a *generic method*;

c) some methods are tagged "`before:`" or "`after:`" and are called before or after the more specific method in the list cited in the previous item.

Line 16-17 defines a method `make-instance` tagged "`after:`" whose first parameter has name `aClass` whose type is class `Traced-class`. Method `make-instance` creates a new object of its first parameter, it would be the "`new`" operator or method of most

languages. In the message passings

```
(make-instance 'Person)
```

of lines 19-22, the default method `make-instance` is called and, after it, the method of lines 16-17 is called. The reason is that `'Person` is a class and classes are objects in CLOS. The type of object `'Person` is class `Traced-class`, as specified in line 10 by the option `metaclass:`. `Traced-class` inherits from class `standard-class`, the superclass of all metaclasses and the default metaclass. Whenever a class is created without option `metaclass:`, its class will be `standard-class`. Methods like `make-instance` take a first argument of this class.

Method `make-instance`, in line 17, increments slot `numInstances` of object `Person`. Class fields are called *slots* in CLOS. In Java, line 17 would be

```
++aClass.numInstances;
```

`Person` is an object with a slot `numInstances` because its class, `Traced-class`, defines a slot with this name in lines 2-4. The initial value of this slot is `0` (line 3) and it has methods for getting and setting it (`numInstances` and `setf`). Method `numInstances` is called four times in lines 28-32. The argument is object `Person`, which is returned by method `find-class` and by `class-of`. The code of Listing 3.2 prints the following because of lines 26, 29, and 32.

```
name: Meg, age: 3
4
4
```

In a ficticious Java dialect that supports a Metaobject Protocol, classes `TracedClass` and `Person` would be as shown below.

```
class TracedClass extends StandardClass {
    public int numInstances = 0;
    // assume objects are created with makeInstances
    @Override
    public Object makeInstance(Object ...keys) {
        Object obj = super.makeInstance(keys);
        ++numInstances;
        return obj;
    }
}

class Person metaclass TracedClass {
       // the fields are public, no get
       // and set as in CLOS
    public String name = "Meg";
```

```
    public int age = 3;
    public void PrettyPrint() {
        System.out.println("name : " + name +
            " age: " + age);
    }
}
```

Listing 3.2 – Metaclass `traced-class` in CLOS

```
1  (defclass Traced-class (standard-class)
2     (( numInstances
3        :initform 0
4        :accessor numInstances)))
5
6  (defclass Person ()
7      ( (name :initform "Meg" :accessor name)
8        (age  :initform 3     :accessor age )
9      )
10     (:metaclass Traced-class))
11
12 (defmethod pretty-print ((aPerson Person))
13     (format t "name: ~a, age: ~a"
14         (name aPerson) (age aPerson))
15 )
16 (defmethod make-instance :after ((aClass Traced-class) &key)
17     (incf (slot-value aClass 'numInstances)))
18
19 (defvar meg  (make-instance 'Person))
20 (defvar doky (make-instance 'Person))
21 (setq meg  (make-instance 'Person))
22 (setq doky (make-instance 'Person))
23 (setf (name doky) "Doky")
24 (setf (age doky)  6)
25
26 (pretty-print meg)
27
28 (assert (= (numInstances (class-of  meg)) 4 ) )
29 (print (numInstances (class-of  meg)))
30
31 (assert (= (numInstances (find-class 'Person)) 4 ))
32 (print (numInstances (find-class 'Person)))
```

Listing 3.2 exercises the MOP of CLOS. Class `Person` has a tailor-made metaclass `Traced-class` for which method `make-instance` was specialized.[7] Therefore `Person` has a tailor-made instance allocation. By changing method `make-instance` of this example, one could implement other features such as logging the creating of objects, collecting them in a list, or initializing the object slots with non-default values.

Classes, methods, and generic methods are objects in CLOS whose classes (metaclasses) are, by default, `standard-class`, `standard-method`, and `standard-generic-function`. Methods are associated with every one of these metaclasses for creating instances (like `make-instance`), allocating memory, calculating the precedence of superclasses,[8] ordering the methods of a *generic method*, calling a method, and so on. Most of the classic book "The Art of Metaobject Protocol" (KICZALES; RIVIèRES; BOBROW, 1991) is dedicated to explaining what to redefine in order to achieve a given goal. A class like `Traced-class` may inherit from `standard-class` and be used to specialize a method like `make-instance`. Classes that explicitly declare their metaclasses, as `Person`, are changed by them. To change the behavior of a generic method or method one should create a subclass of `standard-generic-function` or `standard-method` and associate the generic method or method with that class. By doing that, message passings may be intercepted. An example is a metaclass that counts how many times a generic method was called.[9]

CLOS allows the changing of almost every aspect of the language. This was necessary because it was made to replace several Lisp dialects and the language should be able to simulate them. This was possible only because of the MOP.

As a last example, Listing 3.3 shows a metaclass `Get-print` for which method

```
slot-value-using-class
```

is specialized in lines 3-7. The first parameter to this method, named `aClass`, has type `Get-print`. In line 5, method `format` prints to the standard output `"getting "` followed by the slot name.

[7] method "specialized" is called, in most object-oriented languages, a "overridden" or "redefined" method.
[8] Since the language supports multiple inheritance, the superclasses are ordered according to a precedence list.
[9] See page 109 of the book (KICZALES; RIVIèRES; BOBROW, 1991).

Listing 3.3 – Intercepting slot access

```
1  (defclass Get-print (standard-class) () )
2
3  (defmethod slot-value-using-class (
4    (aClass Get-print) instance aSlot )
5    (format t "getting ~s~%" (slot-definition-name aSlot))
6    (call-next-method aClass instance aSlot)
7  )
8
9  (defclass Person ()
10    ((name :initform "Meg" :accessor name)
11    (age :initform 4 :accessor age) )
12    (:metaclass Get-print))
13
14 (defvar meg (make-instance 'Person))
15 (format t "~a~%" (slot-value meg 'name))
16 (format t "~a~%" (slot-value meg 'age))
```

The code of Listing 3.3 prints the following text because of lines 15 and 16.

```
getting NAME
Meg
getting AGE
4
```

3.3 Compile-Time Metaprogramming

Metaprogramming can be made at compile-time. A *metaprogram* can specify how to change a base program, called simply *program*, or do additional checks in it. The program is changed at compile-time and therefore there is a zero overhead at runtime, except, of course, for code that is added to the base program.

Metaprogramming at compile-time has been used for:

a) extending a language with new features;

b) implementing Domain Specific Languages which may employ a completely different syntax than the language;

c) generating boilerplate code, which is repetitive code or with little semantic contents;

d) doing additional checks in the program, even implementing a customized type checker for the language.

The next Subsections explain the compile-time metaprogramming features of several languages, starting with the old Lisp macros and ending with the new languages. Most of the examples of this section were tested with online compilers.

3.3.1 OpenC++

Chiba (CHIBA, 1995) proposed a Metaobject Protocol for C++, resulting in the language OpenC++. The compiler of OpenC++ first parses the code and builds an AST object for it. Then, for each *class* and each *method*, a metaobject is created from classes `Class` and `Function`. Method `CompileSelf` of each metaobject is called and given the opportunity of returning a modified AST. The default behavior of `Class` and `Function` is not change anything.

A subclass of class `Function` may override method `CompileSelf` to change the AST of a method `m`, given that `m` declares this `Function` subclass as its metaclass. Class `Class` declares several methods for intercepting method calls, variable declarations, creation of objects, and read and write of fields. Therefore, a subclass `ClassFunctionCall` of `Class` may redefine a method `CompileMemberFunctionCall` for intercepting method calls. If a class `MyClass` declares `ClassFunctionCall` as its metaclass, method `CompileMemberFunctionCall` is called whenever the compiler finds a method call whose receiver is `MyClass`. This metaclass method can then change or replace the AST of the method call.

The OpenC++ compiler supplies a function for producing AST object from a text. That makes it easy to produce new code as text that is then translated into an AST object. Chiba (CHIBA, 1995) presents a metaclass for persistent objects. The reading of a class field is translated into the loading of an object from disk. This metaclass is implemented by overriding some methods inherited from `Class`, among them the one that is called whenever a class field is read.

3.3.2 OpenJava

OpenJava (TATSUBORI et al., 2000) (TATSUBORI, 1999) is an extension of Java with a Metaobject Protocol somehow similar to OpenC++, although of a higher level. A class in OpenJava[10] is associated with a metaclass that may be user-defined. A metaclass in OpenJava must inherit from class `OJClass` that declares methods for introspection and for modifying the base class, the one associated with the metaclass. A class is associated with

[10] The language was renamed OJ, see http://openjava.sourceforge.net/. However, we use "OpenJava" because this is the name that appeared in the articles and Master dissertation that describe it.

a metaclass using keyword `instantiates` as shown in the following code, from Tatsubori (TATSUBORI et al., 2000).

```
class MyMenuListener
    instantiates ObserverClass
    extends MyObject
    implements MenuListener
{ ... }
```

At compile-time, an object of the metaclass `ObserverClass`, a metaobject, is created and it directs the compilation of `MyMenuListener`, the base class. Metaclass `ObserverClass` overrides method

`void translateDefinition()`

to insert methods in the base class. This method, shown in Listing 3.4, calls introspective methods from `OJClass` that are very similar to the ones of the Java library but with one difference: they return information regarding the base class, not the metaclass, which is the type of "`this`". Classes `OJMethod`, `OJField`, `OJConstructor`, and `OJModifier` are used for representing methods, fields, constructors, and field/method modifiers (like "public" or "abstract").

Listing 3.4 – Creating a new method. Source: (TATSUBORI et al., 2000)

```
1  void translateDefinition() {
2          // get the methods of the base class
3      OJMethod[] m = this.getMethods(this);
4      for (int i = 0; i < m.length; ++i) {
5          OJModifier modif = m[i].getModifiers();
6          if ( modif.isAbstract() ) {
7                  // if the method is abstract,
8                  // create a new method
9              OJMethod n = new OJMethod(this,
10                     m[i].getModifiers().removeAbstract(),
11                     m[i].getReturnType(),
12                     m[i].getName(),
```

This `translateDefinition` method scans all base class methods looking for abstract methods. When one is found, a new method is created in lines 9-15 with a single statement:

```
return ;
```

Then the new method is inserted in the base class in line 16.[11]

`OJClass` has methods for intercepting the allocation of objects of the base class (with `new`), arrays of objects, method calls, field read and write, and casts of expressions to the base class. For example, to intercept calls to methods of a base class, its metaclass should override the following method.

```
public Expression expandMethodCall(
    MethodCall expr, Environment env )
```

In OpenJava, metaclasses can only be associated with classes. A high-level library of AST classes is used, with classes like `OJField` and `OJMethod`, instead of the real AST classes.

3.3.3 Aspects

Teitelman (TEITELMAN, 1966) proposed, in 1966, advising a Lisp function with new functions that would be called at its entry and exit points. Hence, a function (or even a set of functions) would have a new behavior without editing it. These ideas were refined into a new programming paradigm called *Aspect-Oriented Programming* (AOP) (KICZALES et al., 1997). Code that implements an aspect of a program, like synchronization or error handling, is scattered in the source code. In a language that supports AOP, an aspect can be put in just one place, making it much more modular. But in order to be effective, the aspect code is introduced, by the compiler, in specific points of the generated code. This is called *weaving*.

The terminology of the paradigm is described in the items below using AspectJ (KICZALES et al., 2001) (THE..., 2020), a Java extension for AOP.

a) *Aspect* is a concern that cross cut several classes. In AspectJ, it is a class-like Java declaration that may declare methods, fields, *pointcuts*, *advices*, and *inter-type declarations*.

b) *Pointcuts* are declarations that specify or pick out certain points of the code, called *joint points*. For example, the *pointcut*

```
pointcut getting_a_Name:
    call( void Person.getName() );
```

picks out the calls to method `getName` of class `Person`. The *pointcut*

[11] It is supposed that the old abstract method is removed when the new method, that has the same name, is inserted by "`this.addMethod(n)`".

```
pointcut getting_anything:
    call( void Person.get*() );
```

selects all calls to methods of `Person` starting with `get`.

c) An *advice* is composed of a *pointcut* and code that should be run at it.

```
public aspect BeforeGetName {
      // start of an advice
    before(): getting_a_Name {
        System.out.println(
           "getting the name, of course");
    }
      // end of the advice
}
```

In this example, the string `"getting ..."` is printed before the `getName` method is called. `BeforeGetName` is an *aspect*. Both *pointcuts* and *advices* may have parameters, a feature that is not be explained here. There are several types of advices. The main ones are "before", shown in the above example, "after" (called after the join points), and "around" (before and after the joint point).

d) *Inter-type declarations* are used for changing the code at compile-time. An aspect may add fields, methods, and constructors to classes and make a class inherit from another or implement an interface.

```
public aspect AddBirthdate {
    public String Person.getBirthdate() {
        return birthdata;
    }
    public void Person.setBirthdate(String s) {
        birthdate = s;
    }

    private String Person.birthdate;

    declare parents: Person implements Cloneable;
}
```

This aspect adds two methods and one field to class `Person`. It also demands that `Person` implements interface `Cloneable`.

3.3.4 BSJ

The Metaobject Protocol of language OpenJava allows non-local changes, a metaclass may modify code in other source files. Two metaclasses may transform the same code region and the execution order is important for the end result. BSJ (PALMER; SMITH, 2011), Backstage Java, is a Java-based language with compile-time metaprogramming that limits the effects of non-local changes, detects conflicts between different metaprogram snippets, and directs the transformation order of the program by the metaprogram.

An example in BSJ follows. Between lines 6-10 there is a *metaprogram*, code run at compile-time and delimited by `[:` and `:]`. This metaprogram calls a method of class `Utils`[12] that inserts a `compareTo` method in class `Person`.

```
1  #import static com.example.bsj.Utils.*;
2  public class Person {
3      private String givenName;
4      private String middleName;
5      private String surname;
6      [:
7          generateComparedBy(context,
8            <:surname:>, <:givenName:>,
9            <:middleName:>);
10     :]
11 }
```

Method `generateComparedBy` is not shown. It produces code using quosiquotes delimited by `<:` and `:>`, which are also used in the above example in lines 8 and 9. The code is inserted in the current class using a method of parameter `context`, which is the compilation environment. A metaprogram can only change the declaration in which it is. For example, the metaprogram of the example can introduce a method and add an interface to class `Person`. But it cannot change other source files or even create a class outside `Person` in this same source file.

Metaprograms are run on the original AST, not changed by any other metaprograms, unless there are declared dependences on other metaprograms. BSJ use directives `#target` and `#depends` for creating a dependency graph with the metaprograms. A metaprogram that is in a target runs before its dependent metaprograms which can see the AST changed by the target. The BSJ compiler uses a sophisticated algorithm to detect conflicts. This is not simple because a metaprogram may produce code with metaprograms. These may be marked as *targets* or they may *depend* on other metaprograms.

[12] Class `Utils` is imported using `#import static`, which means the static methods of the class are inserted in the name space of class `Person`. Therefore one can use `generateComparedBy` instead of `com.example.bsj.Utils.generateComparedBy`.

Java-like annotations are associated with classes that implement the interface `BsjMetaprogramMetaAnnotation`. Method `execute` of the class is called passing a context object as parameter, the environment of the annotation. Then, this method can insert code in the AST using quasiquotes.

3.3.5 Xtend

Xtend (XTEND, 2020) is a Java extension that supports *active annotations* that have some similarities with Cyan annotations. For example, the Xtend library annotation `@Data`, when associated with a class, creates several methods, a constructor, and changes all fields to `final`.[13]

```
@Data
class Person {
    String name;
    int age;
}
```

The declaration of `Data` is itself annotated.

```
@Target(ElementType.TYPE)
@Active(DataProcessor)
@Documented
@GwtCompatible
annotation Data {
}
```

The annotation target, to where it may be attached to, is specified using `@Target` with a value of the Java enumeration `ElementType`.[14] The parameter to `@Active` is the *processor class*, a class with methods that transform the class annotated with `@Data`. The other annotations to `Data` are not important here.

A *processor class* must implement one or more of four Xtend interfaces:

a) `RegisterGlobalsParticipant` for registering a new type. A method of this interface should be implemented by the *processor class* to create a new empty type such as an Xtend class or interface;

b) `TransformationParticipant` for adding, removing, or changing fields and methods of any class or interface. In particular, the class attached to the annotation, as `Person` of the previous example, can be changed;

[13] After a value is assigned, the fields cannot change their values.
[14] Defined at https://docs.oracle.com/javase/7/docs/api/java/lang/annotation/ElementType.html

c) `ValidationParticipant` for checks only. The code cannot be changed anymore and all types are resolved;

d) `CodeGenerationParticipant` for generating additional code such as XML files.

Each interface defines exactly one method. The *processor class* of `Data`, for example, overrides method `doTransform` of `TransformationParticipant` in order to insert methods into the associated class. It is important to note that an active annotation can change not only its attached declaration but also any class/interface it can find through a `findClass` or `findInterface` method.

Another example of active annotation is `@Extract` (XTEND, 2020). When attached to a class `MyClass`, it creates an interface `MyClassInterface` with its method signatures. Listing 3.5 presents the *processor class* of the active annotation `@Extract`. This code was taken from the Xtend site (XTEND, 2020) and transformed in order to become more Java-like. The superclass of class `ExtractProcessor` is

`AbstractClassProcessor`

which implements all the interfaces cited previously. Method `doRegisterGlobals` in lines 4-8 overrides a method defined in interface `RegisterGlobalsParticipant`. It creates an empty interface with method `registerInterface`. The interface name is that returned by method `getInterfaceName`. Parameter `annotatedClass` is what it says, the class that is annotated, `MyClass` in the example.

Method `doTransform` overrides a method of interface

`TransformationParticipant`

In lines 17-19, it finds the interface created by `doRegisterGlobals`. Then, this interface is added to the list of implemented interfaces of the annotated class (e.g. `MyClass`) in lines 23-25. A new method signature is created in lines 33-42 and inserted into the interface. An anonymous function, between `[` and `]`, is the second parameter to method `addMethod`. It is executed in `addMethod` after `this` being redirected to the newly created method. Therefore, the new bodyless method will have, for example, the same return type as the original method of the annotated class. The `ExtractProcessor` is a *processor class* that changes a type (an interface) that is not the annotated class.

The code of Listing 3.5 handles the AST or a simpler version of the AST. That can be seen in lines 23-25 (add an interface to a list of implemented interfaces) and in lines 33-42 (insert a method into an interface). There is another way of doing that using code in text format. An example of that, from the Xtend site, is given in Listing 3.6. The text between the `'''` is a *template expression*, a multiline string that may have embedded commands like `IF` and `FOR`. The variables between french quotes, « and », are replaced by their values. Note that the AST is still used to add the method.

Listing 3.5 – Processor class for `Extract` annotation

```
class ExtractProcessor extends AbstractClassProcessor {

  override void
  doRegisterGlobals(ClassDeclaration annotatedClass,
                    RegisterGlobalsContext context) {
    context.registerInterface(
        this.getInterfaceName(annotatedClass) )
  }

  def String getInterfaceName(ClassDeclaration
                                annotatedClass) {
    return annotatedClass.qualifiedName + "Interface"
  }

  override void
  doTransform( MutableClassDeclaration annotatedClass,
       extension TransformationContext context) {
    MutableInterfaceDeclaration interfaceType =
        context.findInterface(
        this.getInterfaceName(annotatedClass) )

    // add interface interfaceType to the list of
    // implemented interfaces
    annotatedClass.implementedInterfaces =
        annotatedClass.implementedInterfaces +
         #[ interfaceType.newTypeReference ]

    // add the public methods to the interface
    for ( method : annotatedClass.declaredMethods ) {
      if (method.visibility == Visibility.PUBLIC) {
          // pass as parameter to addMethod
          // a name, method.simpleName, and
          // the new method data
        interfaceType.addMethod(
            method.simpleName, [
              docComment = method.docComment
              returnType = method.returnType
              for (p : method.parameters) {
                addParameter(p.simpleName, p.type)
              }
              exceptions = method.exceptions
              ]
            ) // closing addMethod
      }
    }
  }
}
```

Listing 3.6 – Transforming string into an AST object in Xtend

```
1
2     observableType.addMethod('set' +
3             fieldName.toFirstUpper) [
4       addParameter(fieldName, fieldType)
5       body = '''
6         «fieldType» _oldValue = this.«fieldName»;
7         this.«fieldName» = «fieldName»;
8         _propertyChangeSupport.firePropertyChange(
9             "«fieldName»", _oldValue, «fieldName»);
10      '''
11    ]
```

3.3.6 Groovy

Language Groovy (KöNIG, 2007) is a Java-based language that supports compile-time metaprogramming through *AST transformations* (GROOVY, 2018). There are two flavors of them: local and global. Local AST transformations are applied to annotated elements such as classes and interfaces. Global AST transformations are applied to every source code in the compilation and they do not demand annotations.

The Groovy compiler has nine compilation phases: initialization, parsing, conversion, semantic analysis, canonicalization, instruction selection, class generation, output, and finalization.[15] The AST is created in phase *conversion* and most of the checks are made in phase *semantic analysis*. Phase *class generation* creates the bytecodes of each class (in memory).

A local AST transformation is associated with an annotation name using the syntax of the next example, taken from the Groovy site (GROOVY, 2018). Annotation `Retention` uses the retention policy of Java.[16] In this example, `SOURCE` means that the annotation is used at compile-time only. `Target` takes as parameters the kinds of elements to which the annotation can be attached to. In this example, it can be attached to a method.

```
// imports go here
@Retention( RetentionPolicy.SOURCE )
@Target( [ElementType.METHOD] )
@GroovyASTTransformationClass(
    [ "gep.WithLoggingASTTransformation" ] )
public @interface WithLogging {
}
```

[15] Unfortunately, the Groovy documentation does not give enough details to a complete understanding of the compiler phases.

[16] See https://docs.oracle.com/javase/7/docs/api/java/lang/annotation/RetentionPolicy.html

Listing 3.7 – Groovy class for WithLogging annotation

```
1  @GroovyASTTransformation(
2      phase=CompilePhase.SEMANTIC_ANALYSIS)
3  class WithLoggingASTTransformation
4            implements ASTTransformation {
5
6      @Override
7      void visit(ASTNode[] nodes, SourceUnit sourceUnit) {
8          MethodNode method = (MethodNode) nodes[1]
9          def startMessage = createPrintlnAst(
10               "Starting $method.name")
11         def endMessage = createPrintlnAst(
12               "Ending $method.name")
13
14         def existingStatements = ((BlockStatement)
15                   method.code).statements
16         existingStatements.add(0, startMessage)
17         existingStatements.add(endMessage)
18     }
19
20     private static Statement
21     createPrintlnAst(String message) {
22         ...  // create an AST object
23     }
```

The last annotation, `GroovyASTTransformationClass` takes as parameters an array of strings, each a Groovy class name. A method `visit` of each of these classes is called during compilation to handle the annotation `WithLogging`.

The class specified in the declaration of annotation `WithLogging` is in Listing 3.7. Annotation `GroovyASTTransformation` stipulates that method `visit` of lines 7-18 should be called in phase *semantic analysis*. An annotation may be associated with transformation classes that are used in several phases, although each class is used in exactly one phase.

Method `visit` takes a first parameter `nodes` that is an array with two AST objects. The first is the AST object of the annotation itself. Through it, method `visit` can extract the annotation parameters. The second array element is the AST object of the annotated element. It would be the AST of `sendMessage` of the following example.

```
@WithLogging
void sendMessage(String msg) {
   // elided
}
```

The second `visit` parameter, `sourceUnit`, is the AST object of the whole file in which

the annotation is.

Annotations may change code they have access to by using AST methods. For example, in lines 16-17 of Listing 3.7, two statements are inserted in the beginning and end of the annotated method (e.g. `sendMessage`). AST objects may be created using several strategies.

a) Creating objects from the AST classes:[17]

```
new ReturnStatement(
    new ConstructorCallExpression(
        classHelper.make(Date),
        ArgumentListExpression.EMPTY_ARGUMENTS
    )
)
```

b) With helper methods:

```
returnS(ctorX(make(Date)))
```

c) Using class `AstBuilder` from a specification:

```
new AstBuilder().buildFromSpec {
    returnStatement {
        constructorCall(Date) {
            argumentList {}
        }
    }
}
```

This example uses a construction of Groovy called *builder*. Method `buildFromSpec` takes an anonymous function as parameter given between `{` and `}`. The parameter is not surrounded by `(` and `)`. Method `buildFromSpec` changes the `this` of the anonymous function in order to call methods like `returnStatement` that return an AST object. This method also takes an anonymous function as argument.

d) Using class `AstBuilder` from a string:

```
new AstBuilder().buildFromString('new Date()')
```

e) Using class `AstBuilder` from code:

```
new AstBuilder().buildFromCode {
    new Date()
```

[17] These examples were taken from a presentation of Paul King available in https://www.slideshare.net/SpringCentral/groovy-asttransforms-paulkingsep2014b.

```
}
```

This same result can be obtained from methods called `macro` that can be used in a method if it is annotated with `@Macro`. There is also a class `MacroClass` for creating new classes.

The AST builders and macro methods of Groovy make it easy to create AST objects. But they have many limitations. For example, some language constructs are not supported and it is difficult to make a reference to an existent AST object. These limitations are not discussed here.

Groovy supports also what is called *global transformations*. They are defined by AST transformation classes that inherit from a class

`ASTTransformation`

Global transformations are not associated with annotations. Method `visit` of a global AST transformation class is called for every source file of the compilation. The compiler knows about these classes through a specification file.

3.3.7 Nemerle

Nemerle (NEMERLE, 2018) (SKALSKI, 2005) is a C♯ based language with support for two kinds of compile-time metaprogramming. The first one is an advanced form of Lisp-like macros, functions that are called at compile-time to produce code that replaces the macro call. Related features have already been discussed in subsection 3.1.4.

The second kind is related to annotations of Xtend, Groovy, and Cyan, although it is also called *macros*. An annotation-like macro `Serializable` is attached to a class `S` with the following syntax.[18]

```
[Serializable]
class S {
        // constructor
    public this (v : int, m : S) { a = v; my = m; }
        // fields
    my : S;
    a : int;
}
```

At compile-time, the macro function `Serializable` of Listing 3.8 is called to transform its attached declaration (class `S` in the example). A macro method may be called in three compilation stages, described below.

[18] This and the following examples were taken from https://github.com/rsdn/nemerle/wiki/Macros-tutorial.

Listing 3.8 – Nemerle macro `Serializable`

```
1  [Nemerle.MacroUsage( Nemerle.MacroPhase.WithTypedMembers,
2                       Nemerle.MacroTargets.Class,
3                       Inherited = true )]
4  macro Serializable (t : TypeBuilder)
5  {
6    /// here we list its fields and choose only those, which
7    /// are not derived or static
8    def fields = t.GetFields (BindingFlags.Instance |
9                              BindingFlags.Public |
10                             BindingFlags.NonPublic |
11                             BindingFlags.DeclaredOnly);
12
13   mutable serializers = [];
14
15   /// code elided
16
17   serializers = <[
18       printf ("<%s>", $(x.Name : string));
19       System.Console.Write ($(nm : name));
20       printf ("</%s>\n", $(x.Name : string));
21       ]>
22       :: serializers   /// :: is "add to the array"
23   /// code elided
24
25 }
```

a) *BeforeInheritance*, which just after parsing. There is no type information, the compiler has not yet related classes and interfaces regarding inheritance and implementation. A macro called in this stage can change the inheritance of the annotated type and the implemented interfaces. It can also create new classes and interfaces and add fields and methods to an annotated type. However, all of this should be made with few type information.

b) *BeforeTypedMembers*, a stage after the compiler calculates the inheritance and implementation hierarchies of all classes and interfaces. Although inheritance information is available, the types of parameters, return value of methods, and fields are not accessible. In this stage, a macro may add fields and methods to classes and interfaces.

c) *WithTypedMembers*, the stage after the compiler has associated types to fields, method parameters, and method return value. Inheritance and implementation should not be changed but the macro may add fields and methods.

Listing 3.8 shows a sketch of macro `Serializable` with many missing lines. This

macro is active in phase *WithTypedMembers* (line 1) and it should annotate a class (line 2). Macros can annotate types (classes, interfaces, and the like), methods, fields, parameters, and other elements that are specific to Nemerle. Option `Inherited` is turned on in line 3 because the macro should be applied to subclasses too. The parameter to the macro function is `TypeBuilder`, the AST object of the annotated class (line 4). If the macro should be attached to a field, for example, the parameters will be a `TypeBuilder`, the class in which the field is, and a `ClassMemberField`, the AST class of a field.

A macro function handles the AST in order to do checks and insert members. For example, using parameter `t` of `Serializable` we could insert a field `count` by calling method `Define`:

```
t.Define( <[
      decl:  mutable count : int;
  ]> )
```

`mutable` is the Nemerle keyword for non-read-only fields and `<[` and `]>` delimits a quoted code. Code may be spliced with `$( ... )` inside quoted code. It is even possible to use pattern matching with quoted code and handle the AST objects in a high-level way. For example, the following code adds statements `printf` before and after a method body.

```
m.Body = <[ printf("before\n");
            $(m.Body)
            printf("after\n"); ]>;
```

Using AST objects, the macro may get information on the annotated element, as is made in lines 8-11 of Listing 3.8. To `fields` is assigned a list of fields with the given properties.

4 Comparison and Future Works

This chapter compares Cyan with other languages and presents future works. The features of the Cyan MOP are compared with features of other metaprogramming systems in section 4.1. Cyan has some strong points when compared with other programming systems with relation to usability and simplicity. However, these benefits are, for the most part, a side-effect of the main goal of the design of the Cyan MOP, which was to address the list of problems with metaprogramming presented in **??**. The comparison of Cyan with other languages in relation to these problems is presented in section 4.2. Metaobject `concept` of Cyan is important enough to have its own comparative section 4.3. Cyan supports user-defined literals of numbers and strings, pluggable types, language-oriented programming, and Codegs. These features and metaobject kinds are compared to similar mechanisms of other languages and environments in section 4.4. Future works are described in section 4.5.

The Cyan Metaobject Protocol may be improved in a number of ways, employed in several Computer Science areas, and give rise to new research. All of this is presented in section 4.5.

4.1 Comparison of Metaprogramming features

This thesis is about the *compile-time* metaobject protocol of Cyan. But section 3.2 on runtime metaprogramming was included because many metaprogramming features can be implemented either way. And because some benefits can only be achieved with runtime metaprogramming, cited below.

a) Add fields and methods to a class or object at runtime. It may be impossible to foresee beforehand which fields and methods will be necessary at runtime. And only some objects may need to be changed, not all objects of a class. As seen in section A.4, this can be *simulated* in Cyan.

b) Replace methods at runtime. Method `print` of an object `out` used for printing in the standard output may be replaced by another that sends the data to a printer, for example.

c) Intercept message passings at runtime, as done with metaobjects of 3-KRS described in subsection 3.2.3. Method `doesNotUnderstand:` of Smalltalk is able to intercept a message passing for which no method is found (subsection 3.2.2). In Cyan, a method `doesNotUnderstand:` can be defined with the same purpose as that of Smalltalk. But message passings cannot be intercepted at runtime as

in 3-KRS.

d) Call a method whose name is only known at runtime, as is made with method `perform:` of Smalltalk. Operator ` of Cyan does exactly that.

e) Discover the fields and methods supported by an object at runtime. That is useful, for example, for a method that produces XML code from its parameter object. The method would ask the object for a list of its field names and values and produce the file based on them. Cyan does support this through methods of prototype `Any`, the top-level prototype.

f) Load classes at runtime. This feature is used in the Cyan compiler to load metaobject classes from the imported packages of a source code (file). These classes were not known, in the general case, when the compiler was itself compiled. Cyan can use the Java libraries to load prototypes at runtime.

g) Create and evaluate code at runtime. A simple example is to read an expression from the keyboard and plot the graph of it in a Graphical User Interface. Or create and execute a SQL command. A more sophisticated example is to create a whole class based on database data. Cyan may support the evaluation of statements and expressions created at runtime. This could be made by interpreting the statements and expressions, as is made in metaobject `action_afterResTypes_semAn` of section 2.3. Code may be created and compiled at runtime in any language. It is just a matter of producing text (the source code) and calling the compiler. The question is how easy is to do that and if the code produced can interact with the current program.

Runtime metaprogramming is a useful tool but it does have some weaknesses. It brings a performance penalty, although that can be alleviated with an optimized compiler as is made with languages Self (CHAMBERS; UNGAR, 1991) and CLOS (KICZALES; RIVIèRES; BOBROW, 1991). Programs that change themselves at runtime can be extremely difficult to understand, debug, and maintain. Whenever there is an option, this mechanism should not be used. Code produced at runtime may have syntactic and semantic errors, adding one more error source to the program.

Some tasks cannot be made with runtime metaprogramming:

a) introduce new syntax in the language, as do Lisp macros and annotations with DSL code of Cyan;

b) check the code at compile-time. A library, for example, cannot check, at compile-time, if its classes and methods are being used correctly.

Some procedures can be made both at compile and at runtime if the language is powerful enough. For example, message passing, object creation, and field access interception, inheritance change, method and field addition to classes, and all introspective

reflection features. In CLOS and 3-KRS, most of this can be made at runtime. In Cyan, they are all made at compile-time except inheritance change.

In the next subsections, we compare features of Cyan with those of languages that offer compile-time metaprogramming. Before that, it is important to note that there is a clear division between the languages compared. On one side, that we call *group MOP*, there are languages with a full Metaobject Protocol: CLOS (KICZALES; RIVIèRES; BOBROW, 1991), OpenC++ (CHIBA, 1995), and OpenJava (TATSUBORI et al., 2000). On the other side, *group MetaProg*, there are languages with support for compile-time metaprogramming but without a proper MOP. These include BSJ (PALMER; SMITH, 2011), Xtend (XTEND, 2020), Groovy (GROOVY, 2018), and Nemerle (NEMERLE, 2018). The third group is composed of languages that use generics and patterns for code generation — they have not been described before. The forth group is composed only of AspectJ (THE..., 2020).

4.1.1 Languages with a Metaobject Protocol

Group MOP contains languages with a full Metaobject Protocol. Articles on them were published before the year 2005. Classes of languages of this group are associated with metaclasses that control their semantics and code generation. The same is true for some other declarations in CLOS and OpenC++. There is a default metaclass with the standard algorithms defined by the language.

Each class, method, generic method, etc of CLOS has a default metaclass that specifies the default semantics. A single alternative metaclass may be specified and it may change the default behavior. Two metaclasses cannot be associated with the same declaration (class, method, etc). To achieve the equivalent result, a new metaclass that composes the behavior of the two has to be designed.

In Cyan, the default behavior is given by the compiler itself, not by a metaobject. And more than one annotation may be attached to a declaration, making it easy to change different aspects of a declaration, which is more difficult in CLOS. A metaclass in CLOS may replace the default language semantics. In Cyan, metaobjects may add code to prototypes but they cannot change the way the compiler generates code. And metaobjects can do additional checks, they are not able to prevent the compiler from doing the checks the language demands.

In Cyan, the AST is available to metaobjects, although it cannot be changed. To our knowledge, the AST is not available for handling, introspection or intercession, in CLOS. Cyan metaobjects may introspect any AST objects they can grab. The current prototype and current method are always available. Their AST objects may be introspected, although subject to security limitations as explained in section 1.4 (MessWithOthers).

In Cyan, metaobjects *ask* the compiler to add code, rename a method, and replace a statement (using method `replaceStatementByCode` of class `ICompiler_semAn` or methods of interface `IActionMessageSend_semAn`).[1] These are the three forms of changing code in Cyan. Code to be added is always returned by methods of metaobjects that override some interface method. This is unlike CLOS, in which code may be added to a program using the metaprogramming facilities of Lisp. A class or method in this language can be created like any other object. Thus, a conflict may occur and there are no clues on how to solve it.

Cyan adds code as strings that may not be valid Cyan code. That is discovered only afterward when the source code is compiled again. In CLOS, errors are caught only at runtime. Annotations in Cyan can have parameters and an attached DSL code that has the purpose of configuring the metaobject. Metaclasses in CLOS cannot be configured — no parameters and no DSL code. And they cannot help to build DSLs whose code is checked at compile-time (the language is dynamically-typed and, besides that, all metaprogramming is made at runtime).

Feigl (FEIGL, 2011) describes a Scheme-based MOP that was built to be more efficient than that of CLOS. However, the main structure of the protocol is the same, and therefore the comparison of Cyan with CLOS applies, for the most part, to this MOP.

A Cyan metaobject class/prototype may implement several Java/Cyan interfaces that are activated in several compiler phases and may change and check prototypes, fields, and methods. In CLOS, the same result is obtained by using several metaclasses, breaking modularity. To achieve the same goal, not only several metaclasses have to be built but they also have to be associated with several declarations.

The CLOS MOP covers every aspect of the language, everything can be changed, including some features not supported by Cyan such as method combination and multiple inheritance. Cyan is more restricted by choice. Radical changes, as to replace the superprototype or change the type of a parameter, were ruled out since the beginning. However, Cyan metaobjects have some powers that CLOS metaclasses do not have. They can be activated when a prototype is inherited and when a method is overridden in a subprototype. There is a protocol for metaobject communication in Cyan.

The MOP of OpenC++ and OpenJava allow the interception of method calls, field access, object creation, and so on using metaclasses. A metaobject is an instance of a metaclass that is created at compile-time. A metaobject may change the code and do checks in one compiler phase only. In Cyan, there are four *main* occasions in which metaobjects may act: phases parsing, afterResTypes, semAn, and afterSemAn. Metaobjects may add fields and methods only in phases parsing[2] and afterResTypes. That means metaobjects active in later phases know for sure that fields and methods of the current prototype will

[1] Note: expressions, like message passings, are considered statements in Cyan.
[2] This will soon be deprecated. Phase parsing will be for checking and parsing of the attached DSL code

not change.

OpenC++ and OpenJava have just one compiler phase, which makes the MOP simpler than that of Cyan. However, in this last language the division of the compilation in phases helps to clarify the design of metaobjects and prevent some errors such as different views of the code. Most of the comparison between Cyan and CLOS also applies to a comparison between Cyan and OpenC++/OpenJava. In particular, these two languages have a default metaclass for each declaration, only a single metaclass may be associated with a declaration, and metaclasses cannot have parameters.

In OpenC++, methods of metaclasses should return AST objects but there is a function that takes strings and produces these objects. Metaclass methods return code that replaces method calls, field access, and so on. Thus, the AST is not changed directly, a very positive aspect.

Metaclasses of OpenJava may change several aspects of any AST object accessible to them. Methods and fields may be added to classes, inheritance may be changed, method bodies may be replaced, parameter types may be changed, and so on. That is all made by calling methods of the AST classes. That means metaprogramming is more error-prone than in Cyan, in which code is returned by methods and the AST is not changed directly. That also means that non-local changes are possible, a non-modular characteristic. Therefore, a metaobject associated with a class `A` may replace the method body of a class that is parameter to one of the methods of `A`.

Python 3 (RAMALHO, 2015) features a MOP and other programming constructs that support metaprogramming. There is a metaclass associated to each class that can modify it. For example, the metaclass can add fields and methods to the class. Metaprogramming in Python 3 and Cyan have very different characteristics.

a) There can be just one metaclass per class in Python, which limits the complexity of metaprogramming. However it also limits its usefulness. The equivalent restriction in Cyan would be to prohibit more than one annotation per prototype. In this case, some of the problems described in **??** would not exist (OrderMatters, InfiniteMetaLoop, and CircularDependency). The conflict and interrelationships among metacode is the source of several of the problems.

b) An annotation can be an expression in Cyan, as `@lineNumber`. In Python, there is no such feature.

c) Cyan metaobjects have access to the current prototype AST. In Python, the AST of a class is not readily available to metaclasses. The obtain the AST, the metaclass has to access the class bytecodes or source code and, using it, build the AST.

d) In Cyan, metaobject methods are called in inheritance or method overridden.

Python does not have a similar functionality.

e) Metaprogramming in Python is done at runtime and, in Cyan, at compile-time. Therefore, Python offers more flexibility since metaclasses can use information that is only available at runtime. The downside is that errors in the generated code are not discovered at compile-time.

Classes and objects of Iguana/J (REDMOND; CAHILL, 2002) can be dynamically adapted through the use of *protocols*. The operations that can be intercepted are method call, method execution, field access, object creation and deletion, and method dispatch. The Java extension Reflex (TANTER et al., 2003) allows the modification of classes at loading time, thus supporting behavioral reflection. Languages Iguana/J and Reflex[3] do not support structural changes like the addition of methods and classes. They are runtime metaprogramming systems.

4.1.2 Languages with Metaprogramming Features

Languages Xtend and, mainly, Groovy, are responsible for the recent development in metaprogramming. These languages, BSJ and Nemerle (group MetaProg), do not support a MOP, they do not use metaclasses. The default behavior is given by the compiler itself, it cannot be replaced. Any changes have to be made by modifying the AST. The metaprogramming mechanism is similar in all of them but BSJ: methods of classes associated with annotations are called in some specific compilation phases stated in the declarations of the annotations. These methods can change the AST of the annotated element or other AST objects that are accessible. Therefore non-local changes are possible — any class or method can be changed by a class associated with an annotation. And almost all modifications are allowed because the AST is handled directly. BSJ stands out for allowing only local changes and for assigning exactly one meaning to the whole metaprogram.

The metaprogramming support, in group MetaProg, is low-level when compared with Cyan and languages of group MOP, requiring AST handling for any code insertion and checks. In Cyan, unless method `replaceStatementByCode` (see item 1.2.4) is used, the code to be added must be returned by a metaobject method that overrides a Java interface method of the MOP. All code handling is made with strings, in contrast with the languages of group MetaProg that use either AST objects or some mechanisms to build AST objects from text (including quotations).

The MOPs of languages of group MOP are carefully described in academic articles, dissertations, and books. The metaprogramming features of languages of group MetaProg are loosely specified with the exception of Nemerle (SKALSKI, 2005) (SKALSKI; MOSKAL;

[3] Reflex was cited in this section although it is an *open* architecture, it does not impose a specific metaobject protocol.

OLSZTA, 2005) and BSJ. More information on these languages would be necessary for a detailed comparison between them and Cyan. We compared with Cyan only the specification of the metaprogramming constructs given in sites, books, and articles. The use of test cases with the compiler would eliminate some of our doubts on the languages. But the semantics given by the compilers could change in future versions while still matching the specification of the languages.

Message passing, object creation, field read and write, inheritance, and method overridden are operations that cannot be intercepted in languages of group MetaProg. That is only possible by using *global AST transformations* of Groovy. With them, the whole AST of a program is available to one class and therefore everything is possible. However, this can hardly be considered a good practice: it slows down the compiler and allows changes in the whole program.

In languages of group MetaProg, there is also no *standard* way of objects of the metaprogram[4] to communicate with each other. In Cyan, there is a well-defined protocol for communication that prevents non-determinism: metaobjects of a prototype supply information that is put in a pool that is shared in a later step. Metaprogramming in language BSJ is done in just one compilation step, unlike other group languages. BSJ relies on a dependency graph to prevent conflicts that are either not solved in other languages or are solved by using different compiler phases.

In Cyan, annotations can be considered statements or expressions. Annotations of languages of group MetaProg cannot be statements or expressions. In Cyan, an annotation that is a statement may do checks and produce code after it, as is made by metaobject `action_afterResTypes_semAn` of section 2.3. Annotations of many metaobject classes of package `cyan.lang` are expressions: `lineNumber`, `compilationInfo`, all literal numbers and literal string annotations. Cyan is more expressive by allowing the possibility of annotations to be expressions.

In languages of groups MOP and MetaProg, there is no direct support for DSLs attached to annotations or metaclasses. However, DSL code may be inside a literal string passed as a parameter to an annotation. Some of the languages of groups MOP and MetaProg are statically-typed: Xtend, Nemerle, OpenJava, BSJ, and Xtend. In none of them, it is possible to parse and interpret code of the own language at compile-time as in Cyan. Hence, an annotation like `action_afterResTypes_semAn`, section 2.3, is at least difficult to implement in these languages.[5]

The Cyan MOP offers the compiler to metaobjects that need to parse or do lexical

[4] That is, objects of classes associated with code transformations like AST transformation classes of Groovy. They are in fact *metaobjects* but they are not called by this name.

[5] It could be achieved with great pain. The compiler could call itself to compile the code. But, in this case, the code could not use objects available to the compiler. An interpreter for the AST could be made. To our knowledge, this has not been done.

analysis in DSL code. The AST built by the compiler for a metaobject may undergo semantic analysis if the metaobject chooses this option. Language Converge (TRATT, 2008) provides the Converge Parser Kit for parsing code based on a grammar. Cyan only supports Cyan-like code.

4.1.3 Languages that Use Generics and Patterns

Generic classes of the C♯ extension language Genoupe (DRAHEIM; LUTTEROTH; WEBER, 2005b) (DRAHEIM; LUTTEROTH; WEBER, 2005a) can use a language for code generation. This language supports statements `foreach` and `if` used to generate code. In the following example, adapted from (DRAHEIM; LUTTEROTH; WEBER, 2005b), `foreach` is used for scanning the fields of `S`, the type parameter to the generic class `C`. For each `S` field, a new field with the same name and type is added to `C`.

```
1 class C(Type S) {
2     @foreach(F in S.GetFields()) {
3         @F.FieldType@  @F.FieldName@;
4     }
5 }
```

Genoupe cannot add code to existing classes. The language offers a high degree of type safety at compile-time. However, it does not guarantee the generated code is correct.

SafeGen (HUANG; ZOOK; SMARAGDAKIS, 2005) is a metalanguage for Java that supports *Generators* for generating well-formed Java code. The language employs a theorem prover that is fed with first-order logical sentences, each one represents a property of the generated code. Either the prover assures that the generated code is well-formed or an error is issued. SafeGen supports statements `#foreach` and `#when` that play a role similar to `foreach` and `if` of Genoupe.

The metaprogramming system CTR (FäHNDRICH; CARBIN; LARUS, 2006) is an C♯ extension with a mechanism, called *transformers*, that combine patterns and generation templates. The generation template is applied whenever the associated pattern matches a code (like a class). A transformer can, for example, create new classes or add methods to a class. The generated code is checked first by CTR and, later, by the compiler.

MorphJ (HUANG; SMARAGDAKIS, 2011) uses a technique called *morphing* for building classes based on the fields and methods of their type parameters (which are generic classes). An example, taken from (HUANG; SMARAGDAKIS, 2011), creates a class `AddGetter<X>` with `get` methods for each field of class `X`.

```
class AddGetter<class X> extends X {
    <F>[f] for ( F f : X.fields )
```

```
    F get#f () { return super.f; }
}
```

In line 2, there is a pattern that matches the public fields of `X`. `<F>` means that `F` is a type variable and `[f]` means that `f` is a name variable. The `for` statement iteracts over all public fields of `X`.

As can be seen in this example, classes instantiated from the same generic class may have different structures. The MorphJ compiler detects not well-formed code in generic classes before instantiations. Patterns can be positive, as in the previous example, or negative. Since MorphJ creates code by instantiating generic classes, it cannot add code to existing classes.

Model MTJ (REPPY; TURON, 2007) offers *trait functions* composed of `requires` and `provides` clauses. Trait `PropT` below, taken from (REPPY; TURON, 2007), has only the `provides` clause.

```
trait PropT ($f, $g, $s, T)
    provides {
        private T $f;
        public void $s (T x) { $f = x; }
        public T $g () { return $f; }
    }
```

This trait is used twice in class `Point2`, with different arguments.

```
class Point2 {
    use PropT (x, getX, setX, int);
    use PropT (y, getY, setY, int);
    Point2 () { x = 0; y = 0; }
}
```

The result is that fields `x` and `y` of type `int` and get and set methods for them are added to class `Point2`. Everything declared in the `provides` clause is added to the class. This trait can be implemented in Cyan as a metaobject. The `use` clause would be just annotations:

```
object Point2 {
    @propT(x, getX, setX, int)
    @propT(y, getY, setY, int)
    Point2 () { x = 0; y = 0; }
}
```

Restrictions on real arguments are declared in clause `requires`. PTFJ (MIAO; SIEK, 2012) is a MTJ extension employing patterns borrowed from MorphJ. Both MTJ and PTFJ cannot add statements to methods — they are able to add only whole methods. This restriction is lifted in an extension of PTFJ (MIAO; SIEK, 2014). In this language extension, pattern matching is also used for generating statements. For example, if a class has a given method, a statement is generated inside a class method.

The Java extension cJ (HUANG; ZOOK; SMARAGDAKIS, 2007) supports predicates on the type parameters of generic classes. As an example, if a generic class parameter `X` is subclass of class `Y`, then a method is added to the generic class. Therefore, the predicate works as the expression of a *static if* that only generates code if the expression is true. It resembles language D's *static if* (ALEXANDRESCU, 2010), although, in this language, regular functions can be called at compile-time. A D interpreter is used to evaluate them. Returning to cJ, the type-checking of a generic class is made before any instantiations.

Cyan can generate code with lexical, syntactical, and semantic errors. This is unlike Genoupe, SafeGen, CTR, MorphJ, MTJ, and cJ. In Cyan, metaobjects can add code in instantiations of generic prototypes, thus emulating the creation of classes in the above languages. The easier way of doing that is using metaobject `insertCode`.

```
object MyList<T>
    @insertCode{*
        /*
            the code below, elided, tests if T declares a binary + method
            that returns a T value. If yes, method 'sumAll' is added
            to MyList<T>. This method is
                func sumAll -> T { ... }
        */
        ... // elided
    *}
    ...
end
```

The statements of `insertCode`, which are interpreted at compile-time, have access to information about `T` such as its superprototype and its declared methods. The statements test if `T` declares a binary `+` method that returns a `T` value. If yes, method `sumAll` is added to `MyList<T>`. This method is

```
func sumAll -> T { ... }
```

This mechanism is largely used in the Cyan libraries. There are specific metaobjects for generating code for prototype `Tuple` and `Array`, among others, for example. There is an interesting metaobject `addCodeFromMetaobject` that asks for the first generic type parameter which code it should add to the generic prototype. An annotation of this metaobject is at-

tached to the generic prototype `Array<T>`. The code to be added to the generic prototype is given as the attached DSL code of an annotation `addCodeToGenericPrototype` attached to a type `R`. In an instantiation `Array<R>`, `addCodeFromMetaobject` and `addCodeToGenericPrototype` talk to each other and the first metaobject adds to the generic prototype the code supplied by the latter.

For example, prototype `Int` is declared as

```
@addCodeToGenericPrototype(Array, "func sum -> Int"){*

    func sum -> Int {
        var Int s = 0;
        for elem in self {
            s = s + elem
        }
        return s
    }
*}
...
object Int
   ...
end
```

The first parameter to the annotation is the generic prototype instantiation to which the code should be added. The second parameter is the signature of the method given in the attached DSL code. Therefore, prototype `Array<Int>` has a `sum` method that is not added to any other `Array<T>` prototype. The generic prototype `Array<T>` does not know, by itself, of method `sum`.

There is another way of configuring a generic prototype in Cyan. Identifiers starting with a lowercase letter are not considered types when passed as a parameter to a generic prototype. Therefore they can be used to give information to metaobjects. For example, a metaobject whose annotation is attached to prototype `MyList<T>` could create a list optimized for space when instantiated with identifier `space` as in

```
MyList<Int, space>
```

Hence, different instantiations of `MyList` could have different source code and even different fields and methods.

In language MetaFJig* (SERVETTO; ZUCCA, 2013), classes are combined by a set of composition operator for supporting *active libraries*. A class can be created by composing other classes and calling methods returning classes. A customized version of a library is created when the newly created class has nested classes. Metacode that has

already been compiled never causes errors in MetaFJig⋆. The Cyan features that are comparable to class creation in MetaFJig⋆ are generic prototypes (with metaobjects that generate code) and metaobjects that create new prototypes. In both cases, metaobjects are used for customizing the new prototype. However, in Cyan, there is no guarantee that the generate code or the generic prototype, before instantiated, is free of errors. Neither there is a DSL that helps to generate the code, although one could be created and code of it could appear in the attached DSL code of an annotation.

4.1.4 AspectJ

AspectJ is taken as a representative of the aspect languages. In AspectJ, it is possible to specify a set of method calls (and object creation) that should be intercepted, using `*`. In Cyan, an annotation attached to a method or prototype can only intercept message passings related to that prototype.[6] There is an offset between locality and flexibility among the two languages. The behavior is more foreseeable in Cyan because of the locality of the reach of the annotation. But AspectJ offers more power.

One of the design goals of the Cyan MOP is to restrict metaobjects to change only the compilation unit in which their annotations are, with exceptions described in the next paragraphs. Even metaobject of annotations of the *project file* (subsection 1.3.1) follow this rule, let us see why. There are two kinds of metaobject classes whose annotations are used in the *project file*: those who implement interface `IAction_dpp` and those who do not. The former ones act only in the packages or the program. The later ones are applied to every prototype of the package (or the program). They are individually applied to every prototype, there is no communication between the prototypes using the metaobjects. Therefore, to use such kind of annotation is just a shortcut of attaching them to every prototype of a package (or the program). Therefore every annotation only does changes in its prototype.

If the metaobject class implements interfaces

`IActionMessageSend_semAn`

`IActionMethodMissing_semAn`

then code is changed outside the annotation. But the change is expected if the type of the message receiver expression has attached annotations either to itself, to its superprototypes, or to its methods. That is, the non-local changes are described in the documentation of the prototype or the method. They are made to fulfil it.

Inter-type member declarations of AspectJ may introduce fields and methods to classes. The declarations are in an aspect file and the classes are in separate files without being *explicitly* related to the aspect file. By looking to a class, there is no way of saying

[6] An annotation attached to a method can intercept message passings to other methods of the same prototype.

who will change it. That is different in Cyan: fields and methods can only be added by annotations in the prototype or in the project file. We know, by looking at the source file, who can change it. A message passing in a prototype can be replaced by metaobjects of other prototypes. But this replacement is local and it only fulfils the semantics of the message passings.

AspectJ has an *aspect language* for defining aspects, pointcuts, advices, and inter-type declarations. There is no such language in Cyan, metaprogramming is made either in Java or in interpreted Cyan.

4.2 Metaprogramming Systems and their Problems

This subsection discusses how the other languages described in this thesis deal with the problems presented in **??**. For Cyan, this discussion has already been made in section 1.4. Some languages have fewer problems because of their lack of power:

a) if a language uses class patters or generics for code generation, it cannot have any of the problems. Usually, the language does have some problems because it uses patterns or generics and some other mechanisms for code generation;

b) languages that support a single metaclass for each class cannot have the following problems: WhoDidWhat, OrderMatters, InfiniteMetaLoop, and CircularDependency. These problems only exist if there are conflicts among metacode that act on the same base code;

c) some of the problems are associated with compilation (WhoDependsOnWho, Compiler-Interactions, and CircularDependency). Therefore, languages supporting runtime metaprogramming cannot have them;

d) some languages do not allow the addition of code although they permit the interception of operations like object creation and message passing. Some problems, such as MessWithOthers and OrderMatters, cannot occur in these languages.

The problem's description and a discussion on how they affect Cyan and other languages follow. **boldface** is employed for the problem name and *italics* for a short problem description.

MessWithOthers *A metacode in a file changes another source file.*

A non-local change to a file is an AST modification made by a metacode associated with a different file. Non-local changes made by AST handling are allowed in languages OJ, Groovy, Nermerle, and Xtend. Metaclasses of CLOS and OpenC++ can only change the class they are associated with. BSJ supports a security mechanism that does not

allow non-local changes. In Cyan, a metaobject can change an external file just to replace a message passing with an expression. Therefore, the change is expected and problem MessWithOthers does not happens.

In AspectJ, *aspect language* source files can change other files. They keep the cross-cutting concerns of the program and, therefore, these changes are not only *expected* as they are clearly specified using a *static* DSL language. This is not the case with regular metaprogramming. The changes a metacode in one file can do in other file are decided at metacode runtime. These changes are not clearly specified in the annotation that links the source code to the metacode. Therefore, the developer cannot guess, by looking at the source code, that an annotation will cause changes in another source file.

WhoDependsOnWho *Metacode are not taken into account when the compiler builds the dependency graph among source files.*

Metacode associated with an OpenC++ class does not know other classes. In all other languages supporting metaprogramming features, the compiler does not know the dependencies caused by metacode. In Cyan, the compiler builds a table with entries describing the dependency. For each file, there is a list of files it depends on. Each time a metaobject associated with file `X` asks information on another file `Y`, the dependency of `X` from `Y` is added to the table.

KnowsFriendsSecrets *Metacode in one source file know private information of another file.*

Languages with limited metaprogramming powers do not allow a metacode associated to a file to know private details of other files. With the exception of Cyan, more powerfull languages, those that supply the complete AST to metacode, do not limit the visibility of AST objects

Compiler-Interactions *Metacode interact with compiler low-level structures.*

Metaprogramming made with low-level mechanism, such as compiler plugins and AST handling, strongly depend on internal compiler details. Therefore, all compiler-interactions problems occur with languages with such characteristics. The MOP of language OJ permits changes in the AST. However, the language supplies a simplified AST to the metacode that prevents the Compiler-Interactions problem from occurring. Cyan addresses this problem because metaobjects have access to restricted and read-only compiler data-structures (including the AST).

WhoDidWhat *The compiler does not link an inserted code to the metacode that made the insertion.*

Only Cyan and Converge link the metacode and the changes they made in the base program. Converge goes beyond Cyan because even bytecodes[7] know their origin. Thus, even runtime error messages can specify exactly the metacode that may have caused the error. In Converge, two or more locations can be associated with an AST node, a feature that is not supported by Cyan.

OrderMatters *The order metacode is called inside a source file changes metacode behavior.*

In languages that allow direct handling of the AST, a metacode views the modifications made by other metacode that run before it. Thus, changes in the calling order of metacode change the view of each metacode. Some reasons that cause problem OrderMatters in group MetaProg follow.

a) the textual order of annotations in a source file may matter. Changes in this order may introduce compilation errors or changes in the semantics. The example of field `thrashFood` of the previous subsection applies here. Another example is exemplified using language Xtend. Active annotations of this language are processed in the order they appear in the source code (a file). Thus, if the first annotation of a file is `@aaa`, all annotations of this kind in the source file are processed before any others.[8] This can introduce errors:

```
@bbb class First { }
@aaa class Second { String log; }
@bbb @aaa class Third { }
```

Active annotation `@aaa` assumes that the annotated class has a field `log`. Annotation `@bbb` introduces `log` to class `Third` and therefore there is no error in this code because `@bbb` appears in line 1. Now suppose we add an annotation `@aaa` to a class `Zero`.

```
1  @aaa class Zero { public String log; }
2  @bbb class First { }
3  @aaa class Second { public String log; }
4  @bbb @aaa class Third { }
```

This introduces an error in class `Third` because `@aaa` is activated before `@bbb`. It will not find a field `log`;

b) in most languages of groups MetaProg and MOP, an annotation or metaclass in one file may change a class in another file. Therefore, the order of the compilation defines the semantic of the metaprogram, a very undesirable feature;

[7] Intructions of the Converge Virtual Machine.

[8] This information is in a discussion group, https://groups.google.com/forum/#!topic/xtend-lang/_-RTAYBSTLMU, not in the Xtend manual.

c) usually the AST can always be altered, it cannot be transformed into a read-only AST in the last compilation phases. Therefore, checks made by a metacode can be invalidated by changes made by other metacode later on.

There is a keyword, in AspectJ, for specifying the execution order of metacode. Dependencies among metacode, fixing the calling order, may be declared in BSJ. A metacode with a `#target label` clause is executed before a metacode with a `#depends label` clause. The metacode that runs later on can view the changes made by the previous metacode.

Cyan addresses partially this problem. There are two cases in which the annotation order do matter:

a) when metaobjects add code at the start of base methods. The order of the annotations is the order of code insertion;

b) in phase semAn, metaobjects associated to annotations inside a method body have different views of the method statements. If a metaobject associated to annotation `aaa` comes before annotation `bbb`, the metaobject associated with the latter knows all the types of expressions that come before itself. The metaobject associated with annotation `aaa` does not know the types of expressions between itself and `bbb`.

InfiniteMetaLoop *Metacode can generate metacode that, in its turn, generate metacode, and so on.*

This problem happens when metacode generates code (with annotations) that is processed in the same compilation phase. This occurs in CLOS,[9] OpenC++, and every sufficiently powerful metaprogramming. In Cyan, annotations inside code generated by metaobjects are only active in the next compilation phase. Therefore, Cyan addresses this problem.

Nontermination *Metacode may not finish its computation.*

The termination of code generation is ensured by languages SafeGen, MorphJ, and Meta-traits (SERVETTO; ZUCCA, 2013). No language, but Cyan, that allow unrestriced metacode guarantees the termination of metacode execution. This is not the case of languages that generate code using patterns and a few kinds of statements. Cyan addresses this problem by using threads and time limits. If a metaobject method takes longer than a fixed time limit, the compiler is finished.

Nondeterminism *Metacode is nondeterministic.*

[9] Although metaclasses act at runtime, the generated code is also processed at runtime and, therefore, an infinite loop may occur.

Nondeterminism occurs in all metaprogramming systems that allow any contact with the external world (files, system information such as time, Internet, etc). This problem does not occurs in some systems that use patterns for code generation, as C++ templates.

In language Genoupe (DRAHEIM; LUTTEROTH; WEBER, 2005b), two identical expressions in a metacode always return the same value because the language uses memoization to evaluate the expressions. However, nondeterminism may occurs in Genoupe because metacode can access the external word. Thus, the same metacode may generate two different base code in two different compilations.

NoGeneratedCodeGuarantees *Metacode may generate defective code.*

Languages Genoupe, SafeGen, CTR, and MorphJ offer, at compile-time, a high degree of safety in the generated code. They are all pattern-based. The Java extension DynJava (OIWA; MASUHARA; YONEZAWA, 2001) supports special quasi-quotes and rules that assure that runtime generated code is type-safe. Quasi-quoted code in this language keep information on the context in which they should be used. The context includes the local variables, fields and methods, the base class name, and so on. Therefore, when a code `X` is spliced into a quasi-quoted code `Y`, the compiler is able to discover if the context expected by `X` is supplied by `Y`. The Cyan language does not offer any guarantees in relation to the generated code. If the code is not correct according to the language, the compiler will issue an error. If the code is correct but it is not what was intended, a metaobject can check it in phase afterSemAn.

NoContracts *The contract between the metacode and the base code is explicitly stated.*

In SafeGen, predicates are used for restricting the arguments to metacode. As an example, a metacode may accept, as the first argument, only a non-abstract class. The pattern in a CTR *transformer* works as a contract between meta and base code because it limits the classes the transformer can match. The best solution to the NoContracts problem is that of Model MTJ. In it, a trait function has `requires` and `provides` clauses. The former imposes constraints on real arguments and the latter supplies the code added to a class. Cyan does not addresses this problem, although any metaobject can demand a context from its annotation environment and check its generated code. There is even a metaobject with these goals: `concept` of section 2.2.

CircularDependency *Metacode may depend on information produce or changed by other metacode. This dependency relation may be circular.*

This problem occurs in languages that support metaprogramming features and compiler plugins. BSJ solves partially the problem because the metacode execution order may be specified. There are cases, like that of example with metaobject `addFieldInfo` in

Listing 4.1 – Factorial function using C++ templates

```
1  #include <iostream>
2  #include <cstdlib>
3
4  template<unsigned n>
5  struct factorial {
6      static const unsigned value = n*factorial<n-1>::value;
7  };
8
9  template<>
10 struct factorial<0> {
11     static const unsigned value = 1;
12 };
13
14 int main() {
15     std::cout << factorial<5>::value;
16 }
```

subsection 1.2.3, in which there is no correct execution order. In Cyan, this metaobject can be correctly implemented because of algorithm FixMeta of Listing 1.5. However, there are still some cases in which there is no solution at all and cases in which the solution would be demand the extension of FixMeta to handle several prototypes at once.

4.3 Comparison of Concept Features for Generic Programming

A *concept* is a predicate on a generic parameter, which includes restrictions on its syntax and semantics. If the parameter is a type, the common case, a concept represents a family of types. For example, a *concept* may represent all types that have a method `<` that compares two objects. The syntax restriction of this concept would be the need of the `<` method in the type. The semantic restriction would be that this method is transitive: if `a < b` and `b < c`, then `a < c`. Usually, semantic restrictions can only be enforced through tests. A type that obeys a concept is said to *models* the concept.

Concepts arose in the C++ Standard Template Library (STL) (PLAUGER et al., 2000), an old and largely-used library of generic algoritms and classes. Since C++ does not support *concepts*, these are described using comments, test files, and some compile-time language features. C++ templates can be used for compile-time programming (VELDHUIZEN, 2003), a non-planned characteristic that was discovered after the design of templates. As an example, a factorial function is implemented in Listing 4.1 using a `struct` template. The expression `n-1` and the multiplication of line 6 are calculated at compile-time. A literal, `0`, is parameter to the template in line 10. The calculated values

Listing 4.2 – Use of concepts in C++

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <type_traits>
4
5  template<typename T>
6  void printFirst(T elem) {
7      static_assert( std::rank<T>::value == 1,
8          "Parameter must be an array of rank 1" );
9      std::cout << elem[0] << "\n";
10 }
11
12
13 int main()
14 {
15     int array1[] = { 1, 2, 3 };
16     int array2[][3] = { { 4, 5, 6 }, { 7, 8, 9 } };
17     printFirst<int[]>(array1);
18     // error if uncommented
19     //printFirst<int[][3]>(array2);
20 }
```

are always as struct or class fields, as `value` in this case.

Using compile-time programming, some basic functions and values on types are available. Listing 4.2 is a C++ source code that uses header `type_traits`.[10] In lines 7-8, `static_assert` checks, at compile-time, if its first argument is `true`, outputting the second argument, a string, if not. In this example, this function is used to implement *concepts*. It restricts the parameter of `printFirst` to be arrays of rank 1, which is tested by the first argument to `static_assert`. We sketch how this works without giving all the details — they are many.

`value` is a field of struct `rank` of Listing 4.3, inherited from struct `integral_constant` (not shown). `integral_constant` is a struct with a type parameter and a literal value that can be retrieved from field `value`. The following expression is `true`.

```
std::integral_constant<int, 2>::value == 2
```

In lines 8 and 14, there is a sum of struct field `value` with one. This is made at compile-time. The three definitions of `rank` work like the recursive definitions of a function in a functional language:

```
f(0) = 0
f(1) = 1
```

[10] https://en.cppreference.com/w/cpp/header/type_traits

Listing 4.3 – `rank` "function" in C++

```
template<class T>
struct rank : public
    std::integral_constant<std::size_t, 0> {};

template<class T>
struct rank<T[]> : public
    std::integral_constant<std::size_t,
                           rank<T>::value + 1> {
};

template<class T, std::size_t N>
struct rank<T[N]> : public
    std::integral_constant<std::size_t,
                           rank<T>::value + 1> {
};
```

```
f(n+1) = f(n) + 1
```

In an instantiation `std::rank<int [][10]>`, the C++ compiler finds a match of the type `int [][10]` with the third definition, making `rank` inherit from `integral_constant` with parameters `size_t` and `rank<int[]> + 1`. Type `rank<int[]>` matches with the second `rank` definition and so on. At the end, `std::rank<int [][10]>::value` has value 2.

Concepts as a language feature for C++ were proposed by Stroustrup (STROUSTRUP, 2003). However, they have not been adopted yet, although they may be in C++ 20 (SMITH, 2018). To explain C++ *concepts*, we use the definitions and examples from Gregor et al. (GREGOR et al., 2006). Function `min` that follows demands that its type parameter supports operation `<` through a `where` declaration that refers to a `concept` declaration.

```
concept LessThanComparable<typename T> {
    bool operator<(T x, T y);
}

template<typename T>
where LessThanComparable<T>
const T& min(const T& x, const T& y) {
    return x < y? x : y;
}
```

Only types that *model* the concept can be used with function `min`. Concepts, as a language feature, demand a *concept language* defining what is valid in a `concept` declaration and

after a `where` keyword.

A type that does not have a `<` method may be adapted to use `min` using a *concept map*. In the following example, a function `<` is defined in a *concept map* for adapting class `complex` to the concept `LessThanComparable`.

```
concept_map LessThanComparable<complex> {
    bool operator < (complex a, complex b) {
        return abs(a) < abs(b);
    };
}

int f(complex a, dcomplex b) {
    complex& x = min(a, b);
    // ...
}
```

Languages Java (GOSLING et al., 2014), C♯ (C#..., 2020), Rust (Rust..., 2018), Swift (SWIFT, 2022), and Kotlin (JETBRAINS, 2022) support constrains in types of generic classes based on other types, the called "constraints-are-types" approach (BELYAKOVA, 2016). As an example, class `TestVisitor` in Java restricts its parameter `T` to be a subtype of `ast.ASTNode`.

```
class TestVisitor<T extends ast.ASTNode > {
    ...
}
```

This approach does not offer all functionalities of a concept language. The only valid predicates on types are those that can be expressed as subtype relationships. The keyword `extends` in the example means "subtype". If `ast.ASTNode` is an interface, the real parameter `T` may be an interface that inherits from `ast.ASTNode` or a class that implements this interface.

Concepts are implemented as language mechanisms in some versions of C++ (SMITH, 2018) (VOUFO; ZALEWSKI; LUMSDAINE, 2011), Genus (ZHANG et al., 2015), G (SIEK; LUMSDAINE, 2011), Magnolia (BAGGE, 2009), and JavaGI (WEHR; LäMMEL; THIEMANN, 2007). Genus and JavaGI are Java extensions, G and Magnolia are based on C++.

Sutton and Stroustrup (SUTTON; STROUSTRUP, 2011) consider *concepts* as restrictions plus axioms, which are semantic specifications of types. An example follows in which concept `Ordered` demands that its parameter `T` obeys many axioms, among them `Greater`.

```
concept Ordered<Regular T> {
    requires constraint Less<T>;
    requires axiom Strict_total_order<less<T>, T>;
    requires axiom Greater<T>;
    requires axiom Less_equal<T>;
    requires axiom Greater_equal<T>;
}

template<typename T>
axiom Greater(T x, T y) {
    (x>y) == (y<x);
}
// other axioms and constraints elided
```

Axioms are difficult or impossible to check statically.[11] But the compiler can generate test cases based on them, as demonstrated by Bagge, David, and Haveraaen (BAGGE; DAVID; HAVERAAEN, 2009).

Language support for generic programming may be offered by many features such as concepts, axioms, interface-based restrictions (as in Java), and other language mechanisms (as in C++ without *concepts*). The support may be minimal, as in Java and C♯, to almost complete, as in the languages G (SIEK; LUMSDAINE, 2011) and Magnolia (BAGGE, 2009). Belyakova (BELYAKOVA, 2016) compares the support of some desired properties by twelve languages. Among these properties, we can cite:

a) may a type retroactively obey the restrictions of a concept? That demands adaptation of the type to the concept without its source code;

b) can two or more types be used in the same *concept*? For example, may a concept demand that two types have the same supertype?

c) can concepts be refined? That is, is there some way of concept inheritance?

d) restrictions on associated types. For example, can we restrict the return value of a method of a type parameter?

e) are concept maps supported? A type models a concept if it obeys its restrictions. How it obeys the restrictions is called a *model*.[12] A simple example is a concept that demands that its type has the comparison operator `<`. Type `String` may

[11] In the general case, they are impossible to check because of the Rice theorem (ODIFREDDI, 1992). This theorem asserts that, apart from trivial properties, runtime properties are uncomputable.

[12] The word "model" is used in two meanings in the literature. A type *models* a concept if it obeys its restrictions. And a *model* is the way a type obeys the restrictions. Some languages have language features that allow the adaptation of a type to a concept.

model this concept in two ways: < may be the regular string comparison operator or it may compare the sizes of the two strings.

To these properties, we add *modular type checking*. A language supports *modular type checking* of a generic element (class, interface, function or other abstraction) if it can be type checked before any instantiation. That is, if the generic element can be type checked only with the information of the concept restrictions. Languages that do not duplicate the code of the generic element, like Java, Scala, and G (SIEK; LUMSDAINE, 2011) have this property, which is difficult to implement in languages that duplicate the code of the generic element, like C++ and Cyan.

Metaobject `concept` in Cyan supports many of properties of concepts of other languages:

a) any number of types can be compared in a single declaration.

```
T subprototype R,
T is typeof(R get)
```

b) some types associated with a generic parameter can be restricted:

```
typeof(T at: Int) subprototype Person
```

The compile-time function `typeof` return the type of an expression. It can be used in any statement of metaobject `concept`. However, it is not possible yet to retrieve the type of a method parameter. We cannot demand, for example, that the parameter of method `at:` is `Int`;

c) axioms may be specified. From them tests are automatically generated;

d) restrictions and axioms can be put in a file and reused. This is the Cyan way of concept refinement. In the following example, file `arithmetic.concept` of package `cyan.lang` is being imported and used with type `T`.

```
@concept(test){*
    cyan.lang.arithmetic(T),
    cyan.lang.comparison(T)
*}
object MyMatrix<T> ... end
```

Language Cyan supports *concepts* using metaobjects, which is a radically different approach from all other languages we know. Therefore, Cyan concepts are not a language feature, which has many benefits. Anyone can copy, paste, and change the metaobject class of `concept`. A new restriction may be implemented in hours or even minutes. Different *concept* metaobjects may coexist, they are imported like any other library resource. Such flexibility is impossible with language-supported concepts.

For each restriction, a specific error message may be given after it:

```
@concept(test){*
    cyan.lang.arithmetic(T),
        "T should have the arithmetic methods"
    cyan.lang.comparison(T)
*}
object MyMatrix<T> ... end
```

When compared with concepts of equivalent power, as those of G (SIEK; LUMSDAINE, 2011) and Magnolia (BAGGE, 2009), metaobject `concept` is simpler. The metaobject class and the helper classes were implemented in only 2000 lines of code.

A future work will be to eliminate the concept language of metaobject `concept`. In its place, Cyan itself will be used, which will be interpreted by the same interpreter used in metaobject `action_afterResTypes_semAn` (section 2.3). The `self` object would belong to a Java class with methods corresponding to the statements of the concept language. Hence, there would be a method `has:` because there is a statement `has` in the current concept language. In the example that follows, besides `has:`, we use a method `call:1 errorMessage:1` to run statements from a file.

```
@concept(test){*
      // this is a message send to self
    call: "cyan.lang.arithmetic(T)"
        errorMessage: "T should have the arithmetic method";

      // this also is a message send to self
    has: T methods: [
        "func unit -> T",
        "func * T -> T",
        "func inverse -> T" ];

      // this is regular Cyan code
    for elem in [ "red", "green", "blue" ] {
        has: R methods: [
            "func " ++ elem ++ " -> Int" ]
    }
*}
object MyMatrix<T> ... end
```

Another future work is to design a DSL for code and project management. Metaobjects whose annotations use this DSL could do the checks cited in the final of section 2.3,

page 92. Any checks can be made with *Myan* or interpreted Cyan, but a DSL just for that could make the job easier.

4.4 Comparison of Other Metaobject Kinds

In this section, we compare the metaobject kinds described in section 1.3 with other languages and systems.

4.4.1 User-defined Number and String Literals

A literal number ending with an identifier, like `101bin`, and a literal string prefixed by an identifier, as `r"a*b"`, are considered annotations in Cyan. Their metaobjects have most of the power of regular metaobjects with a few restrictions. They cannot do checkings after phase semAn for example. The new version of C++ (STROUSTRUP, 2013) supports *user-defined literals* of four kinds: integer, floating-point, string, and character literals. A suffix `i` may appear after a double number if a *literal operator* is defined:

```
    // imaginary literal
constexpr complex<double> operator"" i(long double d)
{
return {0,d}; // complex is a literal type
}
```

Then `3.1415i` is replaced by an object of `complex<double>`. That would be made at runtime but in this case the function is marked as `constexpr`, which enables evaluation at compile-time. User-defined string and character literals take a suffix: `"a*b"regex` and `'f'_runic`. Cyan does not support user-defined character literals.

Non-standard string literals in language Julia (JULIA, 2018) are processed by a macro definition. The literal `r"a*b"` is passed as parameter, without the `r`, to the macro that follows. The macro name is the prefix name followed by `_str`.

```
macro r_str(p)
    Regex(p)
end
```

The return value of the macro is `Regex(p)`. Then, `r"a*b"` is replaced by `Regex("a*b")`.

Both C++ and Julia user-defined literal strings work well in practice. Usually one does not need metaobject power to process them, which is offered by Cyan.

4.4.2 Pluggable Types

Annotations attached to types in Cyan (`Char@letter`) were based on the Checker Framework for Java (THE..., 2018) (PAPI et al., 2008). The original idea was called *pluggable types* and it was proposed by Bracha (BRACHA, 2004). Many of the annotations of the Checker Framework, described in (THE..., 2018), are available in the Cyan libraries. Some are not needed, as `@NonNull`, for types whose variables cannot hold the `null` value. The Checker Framework makes use of flow information for checkings and it may introduce, automatically, annotations in the code. In the next example, it issues a warning in the second line because the type of `s` is considered as annotated with `@NonNull` and, therefore, the test is redundant.

```
String s = "I am something";
if ( s != null ) { k = 1; }
```

In Cyan, annotations are never automatically introduced by metaobjects like in this example.

Class `WrEnv` is the type of several parameters of methods of Java interfaces of the MOP library. If not passed as paramter, an object of it can be retrieved by a method `getEnv` of several of the compiler interfaces (like `ICompiler_afterResTypes`). Class `WrEnv` defines a method

```
WrLocalVarInfo getLocalVariableInfo(
        WrStatementLocalVariableDec varDec)
```

that returns an object with information on a local variable at a given program point. If the variable is initialized only with literals, all of them can be retrieved.

```
var Int k;
if a < b {
    k = 0
}
else {
    k = 1
}
@checkVar
```

Metaobject `checkVar` knows, through method `getLocalVariableInfo`, that `k` may hold values `0` and `1`. It is also possible to discover if an expression was used to initialize the variable. Other kinds of flow information are not available yet.

JavaCOP (MARKSTRUM et al., 2010) (ANDREAE et al., 2006) is a framework for pluggable type system in Java. It features a declarative language for constraints and

Listing 4.4 – Rule `checkNonNull` in JavaCOP

```
1  rule checkNonNull(Assign a) {
2      where ( requiresNonNull(a.lhs) ) {
3          require (definitelyNotNull(a.rhs)):
4              error (a,"Possible null assignment"
5              + " to @NonNull");
6      }
7  }
```

an API for user-defined flow analyses. The declarative code (ANDREAE et al., 2006) of a rule `checkNonNull` is in Listing 4.4.

This rule is checked against all AST nodes of a program whose type is `Assign`, which represents assignments. It uses two predicates: `requiresNonNull` and `definitelyNotNull`. The first is true if its argument has a type with annotation `@NonNull`. In the example, the argument is the left-hand side (lhs) of the assignment. For short, rule `checkNonNull` is only applied to assignments (`Assign` is the parameter type) whose left-hand side has a type annotated with `@NonNull`. The second predicate is true if the argument can never be `null`. Its real argument in the example is the right-hand side of the assignment (rhs). If the boolean expression of `require` is false, an error is issued.

Pluggable types in Cyan are made by subclassing a Java class of the MOP library and overriding some methods. No other help is offered to the metaprogrammer than Java itself. JavaCOP, on the other side, offers a sophisticated declarative language with pattern matching and some high-level statements. Users may define their own dataflow analysis algorithms to be used in JavaCOP (an interface should be implemented).

The Cyan support for pluggable types has two advantages over the Checker Framework and JavaCOP. First, a DSL may be attached to the annotation:

```
var Int@restrictTo{* self >= 100 && self prime *} n;
    // runtime error now
    // a compile-time error in a near future
n = 11;
```

Second, the annotations are associated with metaobjects with all the power they offer. They can create auxiliary methods and fields, create new prototypes, add checks, etc. However, in Cyan there is no declarative language and no advanced dataflow information.

Language Python is dynamically typed but there is a static type checker for it that acts in the whole program, called Mypy[13]. Annotations are not necessary. This is unlile the Cyan pluggable types, which have a local effect. Only variables, parameters,

[13] http://mypy-lang.org/

and expressions of the annotatype type are checked by the metaobject associated with the annotation. An interesting companion tool to Mypy is PyAnnotate[14] that inserts types in Python code based on the runtime type of parameters and method return values.

4.4.3 Language-Oriented Programming

Language-oriented programming (LOP) (DMITRIEV, 2004) is a programming paradigm first proposed by Ward (WARD, 1995). In this paradigm, a program is divided into domains and each one is programmed in its own language, many of which may be Domain Specific Languages (DSLs). In our opinion, the language with better support for LOP is Racket (FELLEISEN et al., 2018) (FELLEISEN et al., 2015). At the start of a source file, one may write

```
#lang racket
```

to specify that the language used is Racket. One can use another identifier after `#lang` to specify another language, defined by the language syntax extension system. A language may be created by adding or removing features from an existing language (defined in Racket, of course). The meaning of some constructs, such as function application, may be redefined too. This is tantamount to intercept method calls in Cyan or function calls in CLOS.

Compared with Cyan, Racket offers much more tools for LOP. However, it should be noted that:

a) the language specification in Racket, with `#lang`, is equivalent of using a file in directory DSL, described in subsection 1.3.2. This directory should contain files whose extensions are metaobject names. The metaobject class is responsible for compiling the file and generating from it one or more Cyan prototypes, as described in subsection 1.3.2. The Cyan compiler may be used to implement a restricted version of Cyan or even an extension of it. Unlike Racket, the implementation of non-trivial Cyan dialects would be very difficult;

b) external tools can be used for implementing DSLs.

Language Converge (TRATT, 2008) permits the embedding of *DSL blocks* with the following syntax.

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

`dslCode` is the code of a DSL that is implemented by function `compilerFunc`. Since Converge is a Python-based language, `dslCode` should be indented in relation to `$<<`. Function `compilerFunc` is called at compile-time with the `dslCode` as parameter, a string,

[14] https://github.com/dropbox/pyannotate

and returns an AST object that replaces the *DSL block*. A Converge Parser Kit (CPK) is supplied for building parsers from grammars. An AST object is also built during parsing. Function `compilerFunc` may use the CPK for compiling the DSL code and quotations for building the AST object that is returned. The grammar given to the CPK can use a rule representing a Converge expression, which makes it easy to integrate the DSL with regular Converge code.

The equivalent of Converge *DSL block* in Cyan is user-defined string literals and attached DSL code in "`@`" annotations. In the first case, there is no help from the MOP for generating Cyan code from the DSL code (the string). In the second case, the Cyan compiler can be used for compiling the DSL code. The compiler has methods for compiling Cyan expressions, statements, types, and literals. The Converge Parser Kit can compile only Converge expressions. In Cyan, the AST produced from the DSL code may undergo semantic analysis as a regular Cyan code — this is the default behavior that may be turned off. Cyan does not offer any other tool for DSL parsing than the Cyan compiler. Of course, an external tool can always be used.

An important difference between Cyan and Converge is that DSLs are implemented in Cyan by metaobject annotations. Therefore, a metaobject is associated with each DSL code and it has all the power of it. Hence, the metaobject may create new prototypes and add code to the program. It has access to all information that the compiler makes available.

According to Erdweg et al. (ERDWEG et al., 2015), Language Workbenches (LW) are environments for simplifying the creation and use of computer languages (which include DSLs). An LW allows the definition of a language and supporting tools such as IDEs, compilers, and debuggers. One example of Language Workbench is Spoofax (KATS; VISSER, 2010) (VOELTER et al., 2013) which uses SDF (HEERING et al., 1989) for defining the syntax of the language using context-free grammars. Spoofax makes use of Stratego (VISSER, 2001) for transforming the AST, which includes generating code. Spoofax is able to build not only a whole compiler from specifications but also the language IDE.

SugarJ (ERDWEG et al., 2011) is a Java-based language that allows syntax extensions and, therefore, DSL embedding. It also uses SDF and Stratego for grammar specifications and transformations. New syntax can be added to a source file by importing a library. The language allows non-trivial syntax to be added such as tuples delimited by parentheses.[15] SugarJ does not support a MOP as Cyan. Its syntax extension feature is so complete that SugarJ is even considered a Language Workbench by Erdweg et al. (ERDWEG et al., 2015).

[15] It is non-trivial because there is no keyword in the start of the construction as a Cyan macro.

Cyan offers only the basic support for implementing DSLs, very far away from the features offered by Language Workbenches. The two can only be lightly compared because their goals are very different. The only positive side of Cyan is that DSLs are integrated with the language, supported by metaobjects.

Polyglot (NYSTROM; CLARKSON; MYERS, 2003) and ExtendJ[16] (formerly JastAddJ) (EKMAN; HEDIN, 2007) are examples of *extensible compilers* for Java. Both can be used to extend the Java syntax, including the support for DSLs. There is no such thing in Cyan, Saci is not extensible, grammar rules cannot be added to the language. Metaobjects in Cyan can add new checks in addition to those made by the compiler, they cannot prevent the compiler from doing checks or generating code.

4.4.4 Codegs

Codegs are metaobjects that, at editing time, may activate a Graphical User Interface (GUI), as described in subsection 1.3.7. They are not directly related to any language construct that we know of but there are some related works.

Codea (CODEA, 2018) is an IOs App for developing games in Lua (IERUSALIMSCHY, 2013) that have at least one feature that works like Codeg `color`, described in subsection 1.3.7. According to the Codea site, images and sounds can also be chosen visually. The documentation on Codea does not claim that these features can be user-made as Cyan Codegs. And they are not part of the Lua language, they are an editor feature.

Language injection (IDEA, 2018) of the IntelliJ IDEA IDE of Jetbrains brings code assistance for languages inside string literals. The following Java example declares a variable whose type is annotated with `@Language`. During editing time, the IDE supplies syntax and error highlighting and code completion. A code may be injected using IDE menus too.

```
@Language("HTML")
String meg = "<dog> Meg </dog>";
```

The IDE assumes that the literal string is in HTML because of the annotation parameter. The literal can be also edited in a separate *fragment editor*.

Method parameters may be annotated with `@Language`. When typing, in the editor, a method call with a literal string as argument, the IDE offers help to type in the string. Instead of using an annotation, the IDE menus can be used to associate a literal or a parameter with a language. IntelliJ IDEA supports the injection of several languages through plugins. One of them is a regular expression language named "`RegExp`". In a

[16] https://extendj.org/

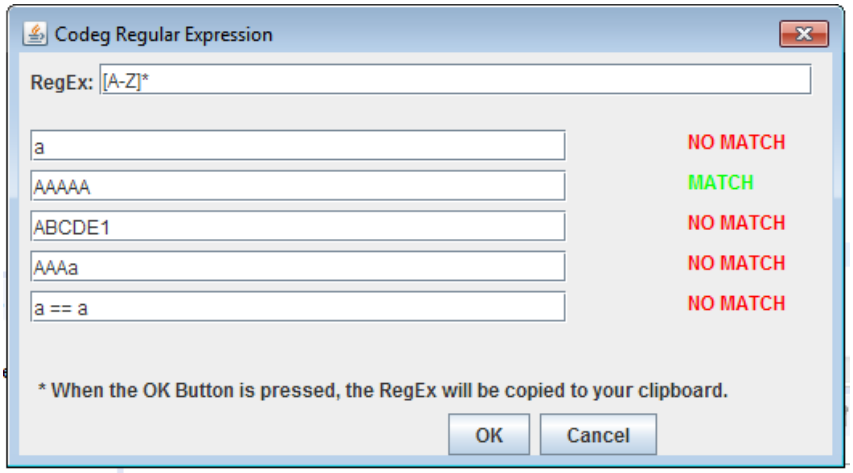

Figure 13 – Graphical interface of Codeg `re` for regular expressions

fragment editor, the regular expression may be typed and matched against a sample string. An annotation can also work as a pluggable type:[17]

```
@Pattern("[a-zA-Z]+")
public String getID() {
    return "no match because of spaces";
}
```

The editor would sign an error because the return string does not match the pattern.

Cyan Codegs link annotations in the Cyan code with GUI for gathering user data. *Language injections* of IntelliJ IDEA link annotations to IDE help for typing in data. And some injections, as for regular expressions, can test the data. Codeg `re` (ROCHA, 2017) is very similar to the language "`RegExp`" of IntelliJ IDEA. Figure 13 shows the window of this Codeg. The regular expression is tested against five strings and the user can test if it matches the ones it should and does not match the ones it should not.

Language injections of IntelliJ IDEA are activated by IDE plugins and they cannot generate code or do checks in the program, their help is in editing time only. Codegs are made available at editing time by importing a library and they can generate code and do checks in the program. Codegs have the full power of metaobjects. On the other side, Codegs do not offer, by themselves, any editor support like language injections.

Codegs are also related to other IDE tools that gather data visually, as tools for generating GUI themselves (WINDOWBUILDER, 2018). A Codeg, yet to be made, could generate code that builds a GUI. Some of the Codegs made by Cassulino (SOUZA, 2017) could be supplied as IDE plugins: `batch` (execute DOS commands and show the result),

[17] This example was taken from https://blog.jetbrains.com/idea/2007/01/custom-languages-and-regular-expressions-in-intellij-idea/

`cmd` (a command prompt), `foundFile` (select a disk file), and `sound` (record and play sound messages). However, Codegs have an advantage over plugins: there may be many different occurrences of Codeg annotations in the same or in different source file, each one keeping different data.

Codegs are a not-to-distant relative of *visual programming languages* (ZHANG, 2007) (BURNETT, 1999), which allow programming using visual means. Codegs are more limited because they just gather data. But Codegs could be built for a visual language. The generated code would be Cyan.

4.5 Future Works

Future works on the Cyan Metaobject Protocol can be grouped into two categories: improvements and increments in the MOP itself and use of metaobjects in other research areas.

several categories, each one composed by one or more Subsections. The first category contains works that demand changes in the compilation process, either to make it faster or to allow separate compilation. Category two contains works whose goals are to make metaprogramming easier and to allow more powerful metaobjects. The third category contains future works on new metaobjects and algorithms.

4.5.1 Reduce the Number of Compilations

Currently, the Cyan compiler does ten compilation phases: parsing (1), resTypes (1), afterResTypes (1), parsing (2), afterResTypes (2), semAn (1), parsing (3), afterResTypes (3), semAn (2), code generation. Metaobjects act only in phases labelled (1). The repetition of phases is caused by the addition of code either by metaobjects or by the compiler. Whenever code is added, compiler has to start from parsing again. Since the compiler always adds code in phase afterResTypes and semAn[18] the code has always to be parsed again after phases afterSemAn and semAn.

A future goal is to make one parsing and one semantic analysis (phases resTypes and semAn). Code, as strings, produced by metaobjects are currently added to the source code in memory, which should be parsed again. Our goal is to parse the code to be inserted producing AST objects. Then, these objects would be inserted in the existing AST.

[18] At least, the compiler adds method `prototype` for every prototype and an inner object for each anonymous function. Inner objects are prototypes declared inside another prototype. They are not allowed in Cyan but are legal after phase semAn if added by the compiler.

4.5.2 Add the Option of Changing the Original Code

Metaobjects may add code as texts to prototypes and create new prototypes but all the changes are made in memory only. Programmers should have the option of making the changes permanent. Therefore, metaobject `insertCode` (section 2.3), for example, could produce code that is incorporated in the source file. The choice for temporary or permanent changes could be made either using options in annotations or a special syntax, like preceding the annotation name by `@@` instead of `@`.

Refactorings (OPDYKE, 1992) are code transformations that do not change its semantics. For example, the renaming of a variable, method, or field.[19] The goal is to improve legibility and maintainability of software.

Refactorings are usually implemented by tools such as IDEs but they could also be specified by a metaobject `refactor` in Cyan. The text attached to the annotations of this metaobject would be code of a yet-to-be-made *refactoring language*, a DSL. This DSL can be Cyan itself with some refactoring methods in object `self`. The example exemplifies the idea. Note that it demands changes in the MOP because currently it does not allow to change prototype names.

```
@refactor{*
      // rename 'length' to 'size'
    rename: MyArray, "func length -> Int",
            "func size -> Int";
      // move 'size' from MyArray to MyCollection
    moveFrom: MyArray
          to: MyCollection
        what: "func size -> Int";
*}
```

4.5.3 Research Works using the Cyan MOP

The Cyan Metaobject Protocol can originate new research or be employed in works of related research areas. This subsection describes some of the possibilities. There are a myriad of them since metaprogramming can be applied in every Computer Science field.

4.5.3.1 Support Ownership

Language Rust (KLABNIK; NICHOLS, 2022) is based on the idea of *ownership*, whose rules are:

a) each value is owned by a variable;

[19] The semantics may change if reflection is used to call the method, access the field, or in a dynamically-typed language. We ignore these cases.

b) each value has only one owner;

c) the value can be disposed of by the runtime system when the owner goes out of scope.

Ownership applies to everything in Rust: memory management (no traditional garbage collector), files (closed when the owner goes out of scope), sockets, etc.

Ownership may be supported in Cyan either by the creation of a new Java interface for the Cyan MOP or by the addition of new methods to the Java interface `IActionAttachedType_semAn`. This interface supports annotations attached to types as `Char@letter`. In the example that follows, file `f` would be automatically closed, at runtime, at the end of the execution of the anonymous function.

```
var fileText = "";
{
    var f = InTextFile("readThis.txt");
    cast s = f readFile {
        fileText = s
    }
} eval;
```

An annotation `@owner(close)` would be attached to prototype `InTextFile`. An alternative would be annotate variable `f`. A kind of ownership, that of anonymous function, is already described in the Cyan manual as future work. This feature allows efficient implementation of functions but it was dropped because the Java Virtual Machine cannot support it.

The implementation of Rust ownership in Cyan is a promising topic of research because of its wide-scale impact on software construction. It makes programming easy because the finalization of resources is automatic. A much bolder project would be add to Cyan the concurrency features of Pony (Pony. . . , 2019) (CLEBSCH et al., 2017). This language uses the *actor model* and it is of a much higher-level than Rust.

4.5.3.2 Adapt Features of the Cyan MOP to Runtime Metaprogramming

The Cyan MOP addresses total or partially many problems that also occur with runtime metaprogramming: MessWithOthers, WhoDependsOnWhol, KnowsFriendsSecrets, WhoDidWhat, OrderMatters, InfiniteMetaLoop, Nontermination, and CircularDependency. The Cyan solutions could be adapted for runtime metaprogramming. The idea is to concentrate all metaprogramming activities in execution points, called MetaPoints, in which only metaprogramming is allowed. In each point, there could be a phase for creating new prototypes and phases afterResType, semAn, and afterSemAn. The compiler would be invoked, for each MetaPoints, to assure that the code produced and added to the base program is free of compilation errors. There should be an entry point, a new `main` method

that is called after each MetaPoint execution. This new main method would be responsible for calling the newly added code. The system would resemble the Cyan MOP, although executed at runtime. Therefore, it would support the same features as the Cyan MOP.

4.5.3.3 Software Restrictions

In page 92 of section 2.3, there is a list of checks that metaobjects can do. Metaobjects can enforce restrictions that usually are only in documentation. In a Software Project, restrictions on methods, types (classes, prototypes, and interfaces), inheritance, and other entities that compose a program are documented but not checked. Some restrictions and requirements follow.

a) Whenever method `equals` of Java (or an equivalent method of other language) is overridden, method `hashCode` should be overridden too. This is already demanded by a metaobject in prototype `Any` of the Cyan package `cyan.lang`.

b) An inherited method annotated with `@shouldBeExtended` should be extended in the subprototype. That is, the overridden subprototype method should call the superprototype method.[20]

c) Metaobjects could limit which packages the prototypes of a package could import. That is, package visibility could be limited by metaobjects. For example, access to the file system could be limited to a single package of the program.

A future work would be to design a *rule language* based on the *concept language* (section 4.3) for expressing restrictions and requirements of prototypes and packages.

4.5.3.4 Software Testing

Software testing benefits from tools for automatically producing code that calls other code and checks the results. Code generation for testing can be made using metaobjects. There are several possibilities for that, described next.

Metaobject `callUnaryMethods` take a regular expression as parameter and produces code that calls all unary methods of the current prototype that matches the expression. It is used for calling the unary methods of a prototype built for testing. Metaobject `insertCode` produces code and it is ideal for generating large amounts of similar test code. Metaobjects can be used for replacing some non-deterministic method calls by deterministic ones. For example, calls to a method that returns the current time can be replaced by an expression that is fixed. In this way, the expected test result becomes deterministic and can be checked. Metaobjects can generate *mock* classes and methods, entities that mimic the real ones, just for testing. Metaobject `concept` (section 2.2) is able

[20] This has been implemented in Cyan.

to generate test cases, called *axioms*, each time the prototype is compiled. This metaobject can be used with any prototype, generic or not.

4.5.3.5 Pluggable Type Systems

New type systems, local to one declaration, are introduced in Cyan by using annotations attached to types (subsection 1.3.6). This is a large area of research and a booming one. Metaobjects can check new restrictions and even replace some language features. One example is an annotation that, when attached to a method, allow that method to be used only in the package of the prototype. Another example is to support a limited form of *structural typing*. In a language that employs *structural typing*, a type is a supertype of any other type that has at least the same methods it has, even when the subtype does not inherits from the supertype. This could be supported, in a limited way, by an annotation `st` attached to type `Dyn`, as shown in the example.

```
var Dyn@st{* func + Int -> Int } elem;
elem = 0; // ok, 0 has a + method
```

To `elem` can be assigned objects of any type that has a `+` method as described in the text attached to annotation `st`. Hence, `0` can be assigned to `elem` because `Int` as a method as described. The MOP should be improved to allow metaobject `st` to check whether messages sent to `elem` match the methods of the annotation text.

4.5.3.6 Live Programming

Codeg `cyan`, subsection 1.3.7, offers a partial support for live programming (TANIMOTO, 2013) when mode "live" is on. The support is partial because *interpreted* Cyan is used, not Cyan. Metaobject `runPastCode` is able to store objects of the last program execution in files and use them during the current compilation. This is a planned feature for Codeg `cyan`. Therefore, the programmer will be able to test methods of the current prototype at editing time.

4.5.3.7 Programming Education

Every AST object has a `visit` method that implements the *Visitor* Design Pattern (GAMMA et al., 1995). The AST objects of a program, package, prototype, and so on can be visited given a top-level AST object. A metaobject for teaching programming may limit the kinds of statements a program can have and demand some program characteristics. A DSL would specify what is expected from the program (the maximum number of prototypes, no import of certain packages, etc) and the restricions of the student code. For example, it cannot use `while` or `repeat-until`. The metaobject annotation would be

attached to the program, in the project file. The student project file would be replaced by the teacher project file when correcting the assignments.

4.5.3.8 Distributed Programming

Ugliara, Vieira e Guimarães (UGLIARA; VIEIRA; GUIMARÃES, 2017) (UGLIARA; VIEIRA; GUIMARãES, 2020) used Cyan metaobjects for software replication, automatizing some repetitive tasks. Several copies of a root object are run in several computers. The chalenge is to keep their state equal. For that, methods of the objects are annotated. The associated metaobjects take care of replication by adding code to communicate to the other objects, in different computers, that the state has changed and how it was changed. This work has not finished. Non-determinism detection needs to be improved, for example.

5 Conclusion

The main contributions of this thesis are the Cyan MOP and its features, a list of problems with metaprogramming, and the explanation on how Cyan addresses most of these problems. Some minor contributions are interesting metaobjects for some areas of research, as `concept` and `grammarMethod`. The Metaobject Protocol of Cyan is composed of Java classes and interfaces or Cyan prototypes and interfaces, the association of them with the compilation phases, and the semantics of the interactions between the compiled code (the program with annotations) and the metaprogram (the metaobject classes). Cyan combines ideas of full MOPs like that of CLOS with more recent work on metaprogramming, like that of Groovy. And it has some unique characteristics that make the building of metaobjects easier and safer. Java or Cyan interfaces are associated with compilation phases. Each phase plays a well-defined role in the protocol, as adding fields and methods in phase afterResTypes or adding statements and expressions, in phase semAn. What is known in each phase is clear to the programmer: no types and no structures are known during parsing, information and types of everything outside method bodies are known in phases afterResTypes and semAn, and the complete picture in phases semAn and afterSemAn. The metaprogrammer knows for sure that some changes, as adding a field, are not possible after some phases. This not only prevents non-determinism but also simplifies the building of metaobject classes. In phase afterSemAn, metaobjects can do checks without fearing any changes will be made by other metaobjects.

The metaprogram uses the Abstract Syntax Tree (AST) for reading only. Metaobjects change the program by return value of methods, which is required when a method of an interface of the MOP library is overridden in a metaobject class or prototype. Other forms of metaprogramming use AST handling for adding code, which is a decision made by running code. In Cyan, the metaprogrammer decides in the *design* of a metaobject class which code to add and where it is added. It is a static decision, before compile-time. Other mechanisms employ a runtime decision with an explicit handling of the AST. In Cyan, code is produced and combined using strings, which is much easier to do and understand than changing the AST. Instead of modifying an AST object, the metaobject may transform part of it in a string and use string concatenation to produce new code. A method `asString` of every AST statement or expression returns it as a string. The text of the source file of the current prototype is available too. Code produced by metaobject methods is returned and therefore metaobjects are *passive* when changing the program, the greater responsibility is on the compiler, a much better tested software than metaobjects. There may be errors in the code generated by metaobjects. They are pointed out during the next compilation. The errors are shown with the context, the prototype in which

the code was inserted, and with a clear message. The compiler displays the name and information on the annotation that produced the code with errors. This is possible because the compiler *tracks* the changes made by metaobjects since it is the compiler itself that does the changes. Other languages allow direct AST handling, which leaves no traces. In our experience, code as strings and the context of errors make it easy to discover and correct errors.

There is a security mechanism in the AST that prevents one prototype code from changing another prototype code or even visiting non-visible methods. The view that a metaobject of a prototype `P` has of external prototypes is exatly the same as the view of `P` code. If a method of another prototype is not visible in `P` code, its AST object is not visible to metaobjects in `P` either. That means the compilation order or prototypes, in a future compiler, will not be dependent on the metaprogram.

Metaobjects act in multiple compilation phases by implementing multiple interfaces. The job of each interface is clearly defined and so is the role of methods declared in them. This is unlike metaprogramming in other languages in which a method of the metaprogram can do everything. As a result, in Cyan, code of metaobject classes are naturally modularized, each method plays a single role that is deduced from its name and the interface declaring it. That makes the metaprogram easier to understand compared with solutions in which a method does it all. A metaobject class may implement multiple interfaces with few restrictions. Consequently, the metaobject may generate code and do checks in several compilation phases, each task assigned to a different metaobject method.

Annotation parameters and attached DSL code permit parametrization of metaobjects. All kinds of Cyan literals are allowed as parameters to annotations. Thus, a metaobject may play different roles according to the annotation parameters and DSL code. Metaobject fields keep data on previous compilation phases since parsing, a largely-used feature. Hence, past information, as the AST of the attached DSL code, flows into the current phase. Metaobjects may generate annotations that will be activated later, which is also a form of communication with future phases.

Metaobjects in the same prototype can communicate with each other, an ideal feature for preventing conflicts. Information needed in metaobject methods may come from their parameters or from the AST. The great majority of metaobjects built so far only use the information given by parameters, which is fortunate because the parameters are more stable than the AST. The AST used by the Cyan MOP was built based on the language, not based on internal details of the compiler. Therefore the AST of the MOP only changes with the language, which means it is very stable.

Metaobjects in Cyan can never bypass compiler checks. Metaobjects may produce code and do additional checks, they can never prevent the compiler from doing its regular work. Therefore, it is not possible for a metaobject to introduce any errors, including

generating bad code, that will not be caught by the compiler. The AST of the MOP is also read-only, which means it cannot be damaged by the metaprogram.

The Cyan Metaobject Protocol was not designed for radical changes in the program. Metaobjects can only act locally, in the prototype in which they are. As a consequence, the user or the programmer of a prototype is sure that its semantics is what the documentation says, there is no surprise. The documentation is made by the prototype programmer and incorporates any changes brought by its own metaobjects. It should not take into account changes and checkings caused by external metaobjects (they cannot act in the prototype). The emphasis of the Cyan MOP is on adding code and checks, not replace them. Therefore, modifications that go against programmer choices are not allowed. For example, to change the superprototype of a prototype, change the type of a variable or parameter, and remove fields and methods. There is a fixed order in which metaobject methods are called in each compilation phase: the textual order of their annotations in the code. Therefore, the call order is deterministic and the programmer is responsible for choosing the order to minimize conflicts. It should be noted that the design choice of local and predictable changes contrasts with the main languages for metaprogramming cited in this thesis.

Annotations may take an attached text and therefore the language supports embedded DSL code. If the DSL is close enough to language Cyan, its code can be parsed by the Cyan compiler itself instead of a tailor-made parser. The parser produces AST objects from the Cyan MOP, which are subject to semantic analysis made by the Cyan compiler during phase semAn. This feature makes it easy to integrate DSLs and Cyan. Metaobject `concept`, for example, uses types of the Cyan program which are checked by the Cyan compiler.

Packages and the program have several special directories whose files are accessed, in a standard way, using parameters passed to metaobject methods. These directories keep Domain Specific Language code, tests, information on previous compilations, temporary files, and code that is transformed into Cyan prototypes. Hence, metaobjects have a standard way of sharing DSL code with other metaobjects and of producing program tests.

Annotations are allowed in the project file of a Cyan program. They can set global compilation variables (accessible from metaobjects). These annotations can also be applied to all prototypes of a package or the program. As a result, the metaobjects associated to them can check the whole program, change prototypes, enforce code style and requirements, and so on.

Metaobjects are used for testing the Cyan compiler itself. Annotations in test files tell the compiler that it should issues some errors. If it does not, the compiler accuses itself.

The Cyan Metaobject Protocol combines the powers of CLOS-like MOPs and

metaprogramming features of new languages. Hence, operations like message passing and field access are intercepted as in CLOS. Code can be inserted in several places and the AST nodes can be visited (new metaprogramming languages). Many different kinds of metaobjects, seven altogether, are united into a single concept. Although there are restrictions, most kinds of metaobjects share the same power. That includes even macros and literal numbers and strings, which are considered metaobjects. Editing time activities and Graphical User Interfaces are integrated with metaprogramming through Codegs. Tasks that are easier to do visually can be made at editing time and use the full power of metaobjects at compile-time.

The greatest majority of characteristics of the Cyan MOP described in the previous paragraphs are not found in metaprogramming systems of other languages. The combination of them makes it easy to design and use metaobjects. In our personal experience, it is easy to code a new metaobject class or prototype. Unless the annotation uses a sophisticated DSL for the attached code, a metaobject class can be made and tested in a few hours, most of them in less than one hour. This is because the Metaobject Protocol supplies almost everything that is needed for metaprogramming and metaobject classes are small. They usually have just one responsibility.

The addition of one more Java/Cyan interface to the MOP library, like those of section 1.2, is easy too. That can be made in around one hour. Besides the creation of the interface, the compiler has to be changed to recognize the interface and use the return value of its methods. Since the infrastructure for that is already made, supporting a new MOP library interface is mostly copy-and-paste programming. More interfaces were not added to the MOP in order to keep it simple. In fact, the Metaobject Protocol underwent several big changes until it stabilized in the current format. Most of the changes had the goal of increasing the available information to metaobjects and of reducing the complexity of the protocol.

The MOP of Cyan is fully functional, available for download at www.cyan-lang.org. Around one hundred metaobject classes were built in a large variety of areas: supply metadata, testing, code style, documentation, generate boilerplate code, support embedded DSL code, code optimization, implement Design Patterns, and checking. Although the metaobjects just touch the surface of each area, their existence evidences that the Metaobject Protocol is easy to use and fits several domains. The major challenges for the protocol were the constructions of the `grammarMethod` and `concept` metaobjects, the most complex ones. The addition of similar functionalities, like *concepts*, to other languages demands heavy changes in the compiler.

The Cyan compiler has around 61,000 of non-blank and non-commented lines of code (SLOC-P). The library of metaobjects, mostly of package `cyan.lang`, has around 12,000 SLOC-P. The Cyan libraries have 5,700 SLOC-P and the compiler has been tested

with 13.000 SLOC-P. All code just cited was made by the thesis author. There are some more Codegs implemented by students. The work on Cyan has not finished yet. There are many missing features in the language and others have to be improved.

Programming languages are used in every area of Computer Science and therefore language features matter in each of them. Metaprogramming brings the possibility of adapting a language to an area and using DSLs for that domain. In particular, Cyan has good metaprogramming features that interrelate with many research areas. They can be used to adapt Cyan to new domains thus generating new research works.

Acknowledgments: this project was financed by FAPESP under Process number 2014/01817-3.

Bibliography

ALEXANDRESCU, A. *The D Programming Language*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2010. ISBN 0321635361, 9780321635365. One citation in page 142.

ANDREAE, C. et al. A framework for implementing pluggable type systems. In: *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 2006. (OOPSLA '06), p. 57–74. ISBN 1-59593-348-4. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/1167473.1167479>. 2 citations in pages 158 and 159.

BAGGE, A. H. *Constructs & Concepts: Language Design for Flexibility and Reliability*. Tese (Doutorado) — University of Bergen, PB 7803, 5020 Bergen, Norway, 2009. Available from Internet: <http://www.ii.uib.no/~anya/phd/>. 3 citations in pages 153, 154, and 156.

BAGGE, A. H.; DAVID, V.; HAVERAAEN, M. The axioms strike back: Testing with concepts and axioms in c++. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 45, n. 2, p. 15–24, out. 2009. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/1837852.1621612>. One citation in page 154.

BARSKI, C. *Land of Lisp: Learn to Program in Lisp, One Game at a Time!* 1st. ed. San Francisco, CA, USA: No Starch Press, 2010. ISBN 1593272812, 9781593272814. 2 citations in pages 102 and 103.

BATORY, D.; LOFASO, B.; SMARAGDAKIS, Y. Jts: Tools for implementing domain-specific languages. In: *Proceedings of the 5th International Conference on Software Reuse*. USA: IEEE Computer Society, 1998. (ICSR '98), p. 143. ISBN 0818683775. One citation in page 101.

BATRA, R. *SQL Primer: An Accelerated Introduction to SQL Basics*. 1st. ed. Berkely, CA, USA: Apress, 2018. ISBN 1484235754, 9781484235751. One citation in page 51.

BELYAKOVA, J. Language support for generic programming in object-oriented languages: Peculiarities, drawbacks, ways of improvement. In: CASTOR, F.; LIU, Y. D. (Ed.). *Programming Languages*. Cham: Springer International Publishing, 2016. p. 1–15. ISBN 978-3-319-45279-1. 2 citations in pages 153 and 154.

BLOCH, J. *Effective Java*. 3. ed. Boston, MA: Addison-Wesley, 2018. ISBN 978-0-13-468599-1. One citation in page 41.

BOBROW, D. G.; GABRIEL, R. P.; WHITE, J. L. Object-oriented programming. In: PAEPCKE, A. (Ed.). Cambridge, MA, USA: MIT Press, 1993. cap. CLOS in Context: The Shape of the Design Space, p. 29–61. ISBN 0-262-16136-2. 2 citations in pages 105 and 113.

BRACHA, G. Pluggable type systems. In: *In OOPSLA'04 Workshop on Revival of Dynamic Languages*. [S.l.: s.n.], 2004. One citation in page 158.

BURMAKO, E. Scala macros: Let our powers combine! on how rich syntax and static types work with metaprogramming. In: *Proceedings of the 4th Workshop on Scala*. New York, NY, USA: Association for Computing Machinery, 2013. (SCALA '13). ISBN 9781450320641. One citation in page 104.

BURMAKO, E. *Def Macros*. 2018. Available from Internet: <https://docs.scala-lang.org/overviews/macros/overview.html>. 2 citations in pages 76 and 104.

BURNETT, M. M. Visual languages. In: *Encyclopedia of Eletrical and Eletronics Engineering*. New York: John Wiley & Sons Inc., 1999. p. 275–283. One citation in page 164.

C# Language Specification. 2020. Available from Internet: <https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/introduction>. 4 citations in pages 106, 153, 189, and 205.

CHAMBERS, C.; UNGAR, D. Making pure object-oriented languages practical. In: *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 1991. (OOPSLA '91), p. 1–15. ISBN 0-201-55417-8. Available from Internet: <http://doi.acm.org/10.1145/117954.117955>. One citation in page 134.

CHIBA, S. A metaobject protocol for c++. In: *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*. New York, NY, USA: ACM, 1995. (OOPSLA '95), p. 285–299. ISBN 0-89791-703-0. 2 citations in pages 118 and 135.

CLEBSCH, S. et al. Orca: Gc and type system co-design for actor languages. *Proc. ACM Program. Lang.*, ACM, New York, NY, USA, v. 1, n. OOPSLA, p. 72:1–72:28, out. 2017. ISSN 2475-1421. Available from Internet: <http://doi.acm.org/10.1145/3133896>. One citation in page 166.

CODEA. *Codea*. 2018. Available from Internet: <https://twolivesleft.com/>. One citation in page 162.

CZARNECKI, K. et al. Dsl implementation in metaocaml, template haskell, and c++. In: _____. [s.n.], 2004. ISBN 978-3-540-25935-0. Available from Internet: <https://doi.org/10.1007/978-3-540-25935-0_4>. 2 citations in pages 98 and 100.

DEARLE, F. *Groovy for Domain-Specific Languages*. 1st. ed. [S.l.]: Packt Publishing, 2010. ISBN 184719690X, 9781847196903. One citation in page 75.

DEMERS, F.-N.; MALENFANT, J. Reflection in logic, functional and object-oriented programming: a short comparative study. In: *In IJCAI '95 Workshop on Reflection and Metalevel Architectures and their Applications in AI*. [S.l.: s.n.], 1995. p. 29–38. One citation in page 105.

DEMICHIEL, L. G.; GABRIEL, R. P. The common lisp object system: An overview. In: *European Conference on Object-oriented Programming on ECOOP '87*. Berlin, Heidelberg: Springer-Verlag, 1987. p. 151–170. ISBN 0-387-18353-1. One citation in page 113.

DMITRIEV, S. *Language Oriented Programming: The Next Programming Paradigm*. 2004. Available from Internet: <http://www.onboard.jetbrains.com/is1/articles/04/10/lop/mps.pdf>. One citation in page 160.

DRAHEIM, D.; LUTTEROTH, C.; WEBER, G. Generative programming for c#. *ACM SIGPLAN Notices*, Association for Computing Machinery, v. 40, n. 8, p. 29–33, 8 2005. ISSN 1523-2867. One citation in page 140.

DRAHEIM, D.; LUTTEROTH, C.; WEBER, G. A type system for reflective program generators. In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Berlin, Heidelberg: Springer-Verlag, 2005. (GPCE'05), p. 327–341. ISBN 3540291385. 2 citations in pages 140 and 149.

ECLIPSE. *The Eclipse IDE*. 2018. Available from Internet: <https://www.eclipse.org/>. One citation in page 52.

ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1. ed. [s.n.], 2011. Available from Internet: <http://www.ecma-international.org/publications/standards/Ecma-262.htm>. One citation in page 106.

EKMAN, T.; HEDIN, G. The jastadd extensible java compiler. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 42, n. 10, p. 1–18, out. 2007. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/1297105.1297029>. One citation in page 162.

ELIXIR. 2018. Available from Internet: <https://elixir-lang.org/>. One citation in page 98.

ERDWEG, S. et al. Sugarj: Library-based syntactic language extensibility. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 46, n. 10, p. 391–406, out. 2011. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/2076021.2048099>. One citation in page 161.

ERDWEG, S. et al. Evaluating and comparing language workbenches. *Comput. Lang. Syst. Struct.*, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, v. 44, n. PA, p. 24–47, dez. 2015. ISSN 1477-8424. Available from Internet: <http://dx.doi.org/10.1016/j.cl.2015.08.007>. 2 citations in pages 45 and 161.

FäHNDRICH, M.; CARBIN, M.; LARUS, J. R. Reflective program generation with patterns. In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. New York, NY, USA: Association for Computing Machinery, 2006. (GPCE '06), p. 275–284. ISBN 1595932372. One citation in page 140.

FEIGL, P. *An Efficient Meta-Object Protocol for Scheme*. Tese (Doutorado) — Johannes Kepler Universitat, 2011. One citation in page 136.

FELLEISEN, M. et al. The racket manifesto. In: *SNAPL*. [S.l.]: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. (LIPIcs, v. 32), p. 113–128. One citation in page 160.

FELLEISEN, M. et al. A programmable programming language. *Commun. ACM*, ACM, New York, NY, USA, v. 61, n. 3, p. 62–71, fev. 2018. ISSN 0001-0782. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/3127323>. One citation in page 160.

FLANAGAN, D.; MATSUMOTO, Y. *The Ruby Programming Language*. First. [S.l.]: O'Reilly, 2008. ISBN 9780596516178. 2 citations in pages 75 and 106.

FOOTE, B.; JOHNSON, R. E. Reflective facilities in smalltalk-80. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 24, n. 10, p. 327–335, set. 1989. ISSN 0362-1340. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/74878.74911>. One citation in page 112.

GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2. 2 citations in pages 39 and 168.

GOLDBERG, A.; ROBSON, D. *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN 0-201-11371-6. 4 citations in pages 106, 109, 189, and 207.

GOSLING, J. et al. *The Java Language Specification, Java SE 8 Edition*. 1st. ed. [S.l.]: Addison-Wesley Professional, 2014. ISBN 013390069X, 9780133900699. 4 citations in pages 106, 153, 189, and 207.

GREGOR, D. et al. Concepts: Linguistic support for generic programming in c++. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 41, n. 10, p. 291–310, out. 2006. ISSN 0362-1340. 4 citations in pages 63, 76, 152, and 205.

GROOVY. *Runtime and compile-time metaprogramming*. 2018. Available from Internet: <http://groovy-lang.org/metaprogramming.html>. 2 citations in pages 126 and 135.

GUIMARÃES, J. d. O. Reflection for statically typed languages. In: JUL, E. (Ed.). *ECOOP*. Springer, 1998. (Lecture Notes in Computer Science, v. 1445), p. 440–461. ISBN 3-540-64737-6. Available from Internet: <http://dblp.uni-trier.de/db/conf/ecoop/ecoop98.html#Guimaraes98>. One citation in page 113.

GUIMARÃES, J. d. O. *The Green Language*. 2013. Available from Internet: <http://www.cyan-lang.org/jose/green/green.htm>. One citation in page 209.

GUIMARÃES, J. d. O. The cyan language. 2020. Available from Internet: <http://cyan-lang.org/articles/>. One citation in page 189.

HACK. *The Hack Programming Language*. 2020. Available from Internet: <http://hacklang.org/>. 2 citations in pages 41 and 93.

HART, T. P. *MACRO Definitions for LISP*. Cambridge, MA, USA, 1963. One citation in page 102.

HEERING, J. et al. The syntax definition formalism sdf; reference manual. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 24, n. 11, p. 43–75, nov. 1989. ISSN 0362-1340. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/71605.71607>. One citation in page 161.

HERZEEL, C.; COSTANZA, P.; D'HONDT, T. Self-sustaining systems. In: HIRSCHFELD, R.; ROSE, K. (Ed.). Berlin, Heidelberg: Springer-Verlag, 2008. cap. Reflection for the Masses, p. 87–122. ISBN 978-3-540-89274-8. Available from Internet: <http://dx.doi-org.ez31.periodicos.capes.gov.br/10.1007/978-3-540-89275-5_6>. 2 citations in pages 106 and 108.

HUANG, S. S.; SMARAGDAKIS, Y. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst.*, Association for Computing Machinery, New York, NY, USA, v. 33, n. 2, fev. 2011. ISSN 0164-0925. One citation in page 140.

HUANG, S. S.; ZOOK, D.; SMARAGDAKIS, Y. Statically safe program generation with safegen. In: *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. Berlin, Heidelberg: Springer-Verlag, 2005. (GPCE'05), p. 309–326. ISBN 3540291385. One citation in page 140.

HUANG, S. S.; ZOOK, D.; SMARAGDAKIS, Y. Cj: Enhancing java with safe type conditions. In: *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*. New York, NY, USA: Association for Computing Machinery, 2007. (AOSD '07), p. 185–198. ISBN 1595936157. One citation in page 142.

IDEA. *Using Language Injections*. 2018. Available from Internet: <https://www.jetbrains.com/help/idea/using-language-injections.html>. One citation in page 162.

IERUSALIMSCHY, R. *Programming in Lua, Third Edition*. 3rd. ed. [S.l.]: Lua.Org, 2013. ISBN 859037985X, 9788590379850. One citation in page 162.

JETBRAINS. *The Kotlin Language*. 2022. Available from Internet: <https://kotlinlang.org/>. 3 citations in pages 106, 153, and 207.

JULIA. *The Julia Language*. 2018. Available from Internet: <https://juliacomputing.com/docs/JuliaDocumentation.pdf>. One citation in page 157.

KATS, L. C.; VISSER, E. The spoofax language workbench. In: *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. New York, NY, USA: ACM, 2010. (OOPSLA '10), p. 237–238. ISBN 978-1-4503-0240-1. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/1869542.1869592>. One citation in page 161.

KERNIGHAN, B. W. *The C Programming Language*. 2nd. ed. [S.l.]: Prentice Hall Professional Technical Reference, 1988. ISBN 0131103709. 2 citations in pages 93 and 200.

KICZALES, G. et al. Object-oriented programming: The clos perspective. In: _____. *Object-oriented Programming: The CLOS Perspective*. Cambridge, MA, USA: MIT Press, 1993. cap. Metaobject protocols: Why we want them and what else they can do, p. 101–118. ISBN 0-262-16136-2. One citation in page 113.

KICZALES, G. et al. An overview of aspectj. In: KNUDSEN, J. L. (Ed.). *ECOOP*. [S.l.]: Springer, 2001. (Lecture Notes in Computer Science, v. 2072), p. 327–353. One citation in page 120.

KICZALES, G. et al. Aspect-oriented programming. In: AKŞIT, M.; MATSUOKA, S. (Ed.). *ECOOP'97 — Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997. p. 220–242. ISBN 978-3-540-69127-3. One citation in page 120.

KICZALES, G.; RIVIèRES, J. des; BOBROW, D. G. *The Art of Metaobject Protocol*. Cambridge, MA, USA: MIT Press, 1991. ISBN 0-262-61074-4. 4 citations in pages 113, 116, 134, and 135.

KING, G. *The Ceylon Language*. 2022. Available from Internet: <https://ceylon-lang.org>. One citation in page 199.

KLABNIK, S.; NICHOLS, C. *The Rust Programming Language*. second. No Starch Press, 2022. ISBN 978-1-59327-828-1. Available from Internet: <https://doc.rust-lang.org/book/>. 4 citations in pages 75, 104, 105, and 165.

KöNIG, D. *Groovy in Action*. New York: Manning, 2007. ISBN 978-1-932394-84-9. 3 citations in pages 106, 126, and 223.

LILIS, Y.; SAVIDIS, A. An integrated implementation framework for compile-time metaprogramming. *Softw. Pract. Exper.*, John Wiley & Sons, Inc., USA, v. 45, n. 6, p. 727–763, jun. 2015. ISSN 0038-0644. One citation in page 99.

MAES, P. Computational reflection. In: *Proceedings of the 11th German Workshop on Artificial Intelligence*. London, UK, UK: Springer-Verlag, 1987. (GWAI '87), p. 251–265. ISBN 3-540-18388-4. Available from Internet: <http://dl-acm-org.ez31.periodicos.capes.gov.br/citation.cfm?id=647607.732995>. One citation in page 112.

MAES, P. *Computational Reflection*. Tese (Doutorado) — Laboratory of Artificial Intelligence, Vrije Universiteit, Brussel, 1 1987. One citation in page 112.

MAES, P. Concepts and experiments in computational reflection. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 22, n. 12, p. 147–155, dez. 1987. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/38807.38821>. One citation in page 112.

MARKSTRUM, S. et al. Javacop: Declarative pluggable types for java. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 32, n. 2, p. 4:1–4:37, fev. 2010. ISSN 0164-0925. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/1667048.1667049>. One citation in page 158.

MCCARTHY, J. History of programming languages i. In: WEXELBLAT, R. L. (Ed.). New York, NY, USA: ACM, 1981. cap. History of LISP, p. 173–185. ISBN 0-12-745040-8. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/800025.1198360>. One citation in page 108.

MIAO, W.; SIEK, J. Pattern-based traits. In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. New York, NY, USA: Association for Computing Machinery, 2012. (SAC '12), p. 1729–1736. ISBN 9781450308571. One citation in page 142.

MIAO, W.; SIEK, J. Compile-time reflection and metaprogramming for java. In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. New York, NY, USA: Association for Computing Machinery, 2014. (PEPM '14), p. 27–37. ISBN 9781450326193. One citation in page 142.

NEMERLE. *The Nemerle Programming Language*. 2018. Available from Internet: <http://nemerle.org>. 3 citations in pages 98, 129, and 135.

NIERSTRASZ, O.; DUCASSE, S.; POLLET, D. *Squeak by Example*. [S.l.]: Square Bracket Associates, 2009. ISBN 3952334103, 9783952334102. One citation in page 109.

NYSTROM, N.; CLARKSON, M. R.; MYERS, A. C. Polyglot: An extensible compiler framework for java. In: *Proceedings of the 12th International Conference on Compiler Construction*. Berlin, Heidelberg: Springer-Verlag, 2003. (CC'03), p. 138–152. ISBN 3-540-00904-3. Available from Internet: <http://dl.acm.org/citation.cfm?id=1765931.1765947>. One citation in page 162.

ODERSKY LEX SPOON, B. V. M. *Programming in Scala: Updated for Scala 2.12 : a comprehensive step-by-step guide*. 3ed.. ed. [S.l.]: Artima, Incorporated., Artima Press [Imprint, 2016. ISBN 0981531687,9780981531687. One citation in page 98.

ODERSKY, M. et al. The Scala Language Specification. 2004. Available from Internet: <http://www.scala-lang.org>. One citation in page 205.

ODIFREDDI, P. *Classical Recursion Theory: The Theory of Functions and Sets of Natural Numbers*. Elsevier Science, 1992. (Studies in Logic and the Foundations of Mathematics). ISBN 9780444894830. Available from Internet: <https://books.google.com.br/books?id=zgE-lQEACAAJ>. One citation in page 154.

OIWA, Y.; MASUHARA, H.; YONEZAWA, A. Dynjava: Type safe dynamic code generation in java. In: *In JSSST Workshop on Programming and Programming Languages, PPL2001, March 2001*. [S.l.: s.n.], 2001. One citation in page 149.

OPDYKE, W. F. *Refactoring Object-oriented Frameworks*. Tese (Doutorado) — University of Illinois at Urbana-Champaign, Champaign, IL, USA, 1992. UMI Order No. GAX93-05645. One citation in page 165.

ORTIN, F.; REDONDO, J. M.; PEREZ-SCHOFIELD, J. B. G. Efficient virtual machine support of runtime structural reflection. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 74, n. 10, p. 836–860, ago. 2009. ISSN 0167-6423. Available from Internet: <http://dx.doi-org.ez31.periodicos.capes.gov.br/10.1016/j.scico.2009.04.001>. One citation in page 106.

PAEPCKE, A. In: PAEPCKE, A. (Ed.). *Object-oriented Programming*. Cambridge, MA, USA: MIT Press, 1993. cap. User-level Language Crafting: Introducing the CLOS Metaobject Protocol, p. 65–99. ISBN 0-262-16136-2. One citation in page 113.

PALMER, Z.; SMITH, S. F. Backstage java: Making a difference in metaprogramming. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 46, n. 10, p. 939–958, out. 2011. ISSN 0362-1340. 2 citations in pages 122 and 135.

PAPI, M. M. et al. Practical pluggable types for java. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2008. (ISSTA '08), p. 201–212. ISBN 978-1-60558-050-0. One citation in page 158.

PICKERING, R.; EASON, K. *Beginning F# 4.0*. 2nd. ed. Berkely, CA, USA: Apress, 2016. ISBN 1484213750, 9781484213759. One citation in page 98.

PLAUGER, P. et al. *C++ Standard Template Library*. 1st. ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000. ISBN 0134376331. One citation in page 150.

Pony, The Language. 2019. Available from Internet: <https://www.ponylang.io/>. One citation in page 166.

QI, X.; MYERS, A. C. Masked types for sound object initialization. In: *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 2009. (POPL '09), p. 53–65. ISBN 978-1-60558-379-2. One citation in page 191.

RAMALHO, L. *Fluent Python: Clear, Concise, and Effective Programming*. [S.l.]: O'Reilly Media, 2015. ISBN 9781491946251. One citation in page 137.

REDMOND, B.; CAHILL, V. Supporting unanticipated dynamic adaptation of application behaviour. In: *Proceedings of the 16th European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2002. (ECOOP '02), p. 205–230. ISBN 3-540-43759-2. One citation in page 138.

REPPY, J.; TURON, A. Metaprogramming with traits. In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2007. (ECOOP'07), p. 373–398. ISBN 3540735887. One citation in page 141.

RIVIèRES, J. d.; SMITH, B. C. The implementation of procedurally reflective languages. In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. New York, NY, USA: ACM, 1984. (LFP '84), p. 331–347. ISBN 0-89791-142-3. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/800055.802050>. One citation in page 108.

ROCHA, E. R. da. *Codegs para Desenvolvimento de Software*. 2017. Final Undergraduate Work. Available from Internet: <http://www.cyan-lang.org>. One citation in page 163.

Rust, The Language. 2018. Available from Internet: <http://www.rust-lang.org/>. 3 citations in pages 48, 75, and 153.

SCALA 3. 2022. Available from Internet: <https://docs.scala-lang.org/scala3>. One citation in page 98.

SEIBEL, P. *Practical Common Lisp*. 1st. ed. Berkely, CA, USA: Apress, 2012. ISBN 1430242906, 9781430242901. 2 citations in pages 102 and 113.

SERVETTO, M.; ZUCCA, E. A meta-circular language for active libraries. In: *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation*. New York, NY, USA: Association for Computing Machinery, 2013. (PEPM '13), p. 117–126. ISBN 9781450318426. 2 citations in pages 143 and 148.

SHEARD, T.; JONES, S. P. Template meta-programming for haskell. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*. New York, NY, USA: ACM, 2002. (Haskell '02), p. 1–16. ISBN 1-58113-605-6. Available from Internet: <http://doi.acm.org/10.1145/581690.581691>. 2 citations in pages 98 and 104.

SIEK, J. G.; LUMSDAINE, A. A language for generic programming in the large. *Sci. Comput. Program.*, Elsevier North-Holland, Inc., Amsterdam, The Netherlands, The Netherlands, v. 76, n. 5, p. 423–465, maio 2011. ISSN 0167-6423. 4 citations in pages 153, 154, 155, and 156.

SKALSKI, K. *Syntax-extending and type-reflecting macros in an object-oriented language*. Dissertação (Mestrado) — University of Wroclaw, Poland, 2005. Nemerle. 2 citations in pages 129 and 138.

SKALSKI, K.; MOSKAL, M.; OLSZTA, P. *Meta-programming in Nemerle*. 2005. Available from Internet: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.101.8265>. 2 citations in pages 104 and 139.

SMARAGDAKIS, Y.; BIBOUDIS, A.; FOURTOUNIS, G. Structured program generation techniques. In: CUNHA, J. et al. (Ed.). *Grand Timely Topics in Software Engineering - International Summer School GTTSE 2015, Braga, Portugal, August 23-29, 2015, Tutorial Lectures*. [S.l.]: Springer, 2015. (Lecture Notes in Computer Science, v. 10223), p. 154–178. One citation in page 97.

SMITH, B. C. Reflection and semantics in lisp. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA: ACM, 1984. (POPL '84), p. 23–35. ISBN 0-89791-125-3. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/800017.800513>. 2 citations in pages 106 and 108.

SMITH, R. *Working Draft, Standard for Programming Language C++*. 2018. 3 citations in pages 78, 152, and 153.

SNYDER, A. Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 21, n. 11, p. 38–45, jun. 1986. ISSN 0362-1340. One citation in page 41.

SOUZA, A. C. A. *Codegs: um Tipo Especial de Metaobjetos em Cyan, Codegs: a special kind of Cyan Metaobjects*. Dissertação (Mestrado) — Federal University of São Carlos, Brazil, 2017. 2 citations in pages 52 and 163.

STROUSTRUP, B. *Concept Checking - A More Abstract Complement to Type Checking*. [S.l.], 2003. Available from Internet: <http://www.stroustrup.com/n1510-concept-checking.pdf>. 2 citations in pages 78 and 152.

STROUSTRUP, B. *The C++ Programming Language*. 4th. ed. [S.l.]: Addison-Wesley Professional, 2013. ISBN 0321563840, 9780321563842. 5 citations in pages 105, 157, 189, 205, and 207.

SUMMERFIELD, M. *Programming in Python 3: A Complete Introduction to the Python Language*. 2nd. ed. [S.l.]: Addison-Wesley Professional, 2009. ISBN 0321680561, 9780321680563. One citation in page 106.

SUTTON, A.; STROUSTRUP, B. Design of Concept Libraries for C++. In: SLOANE, A. M.; ASSMANN, U. (Ed.). *Revised Selected Papers of the Fourth International Conference on Software Language Engineering*. [S.l.]: Springer International Publishing, 2011. (Lecture Notes in Computer Science, v. 6940), p. 97–118. ISBN 978-3-642-28829-6. One citation in page 153.

SWIFT. *The Swift Programming Language (Swift 5.6)*. 1st. ed. Apple Inc., 2022. Available from Internet: <https://docs.swift.org/swift-book/>. One citation in page 153.

TANIMOTO, S. L. A perspective on the evolution of live programming. In: *Proceedings of the 1st International Workshop on Live Programming*. Piscataway, NJ, USA: IEEE Press, 2013. (LIVE '13), p. 31–34. ISBN 978-1-4673-6265-8. Available from Internet: <http://dl-acm-org.ez31.periodicos.capes.gov.br/citation.cfm?id=2662726.2662735>. One citation in page 168.

TANTER, E. et al. Partial behavioral reflection: Spatial and temporal selection of reification. In: *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: Association for Computing Machinery, 2003. (OOPSLA '03), p. 27–46. ISBN 1581137125. One citation in page 138.

TATSUBORI, M. *An Extension Mechanism for the Java Language*. Dissertação (Mestrado) — University of Tsukuba, Japan, 1999. One citation in page 118.

TATSUBORI, M. et al. 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*. London, UK, UK: Springer-Verlag, 2000. p. 117–133. ISBN 3-540-67761-5. 4 citations in pages 15, 118, 119, and 135.

TEITELMAN, W. *PILOT: A STEP TOWARDS MAN-COMPUTER SYMBIOSIS*. Cambridge, MA, USA, 1966. One citation in page 120.

THE AspectJ Language. 2020. Available from Internet: <https://www.eclipse.org/aspectj/doc/next/progguide/language.html>. 2 citations in pages 120 and 135.

THE Checker Framework Manual: Custom pluggable types for Java. 2018. Available from Internet: <https://checkerframework.org/manual/checker-framework-manual.pdf>. One citation in page 158.

TRATT, L. Compile-time meta-programming in a dynamically typed oo language. In: *Proceedings of the 2005 Symposium on Dynamic Languages*. New York, NY, USA: ACM, 2005. (DLS '05), p. 49–63. Available from Internet: <http://doi.acm.org/10.1145/1146841.1146846>. 2 citations in pages 98 and 99.

TRATT, L. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, ACM, New York, NY, USA, v. 30, n. 6, p. 31:1–31:40, out. 2008. ISSN 0164-0925. 3 citations in pages 101, 140, and 160.

UGLIARA, F. A.; VIEIRA, G. M.; GUIMARãES, J. de O. Transparent replication using metaprogramming in cyan. *Science of Computer Programming*, v. 200, p. 102531, 2020. ISSN 0167-6423. Available from Internet: <http://www.sciencedirect.com/science/article/pii/S0167642320301398>. One citation in page 169.

UGLIARA, F. A.; VIEIRA, G. M. D.; GUIMARÃES, J. d. O. Transparent replication using metaprogramming in cyan. In: *Proceedings of the 21st Brazilian Symposium on Programming Languages*. New York, NY, USA: ACM, 2017. (SBLP 2017), p. 9:1–9:8. ISBN 978-1-4503-5389-2. One citation in page 169.

UNGAR, D.; SMITH, R. B. Self: The power of simplicity. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 22, n. 12, p. 227–242, dez. 1987. ISSN 0362-1340. 2 citations in pages 106 and 112.

VELDHUIZEN, T. L. *C++ Templates are Turing Complete*. [S.l.], 2003. 2 citations in pages 105 and 150.

VISSER, E. Stratego: A language for program transformation based on rewriting strategies. In: *Proceedings of the 12th International Conference on Rewriting Techniques and Applications*. Berlin, Heidelberg: Springer-Verlag, 2001. (RTA '01), p. 357–362. ISBN

3-540-42117-3. Available from Internet: <http://dl-acm-org.ez31.periodicos.capes.gov.br/citation.cfm?id=647200.718711>. One citation in page 161.

VOELTER, M. et al. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. [S.l.]: dslbook.org, 2013. 1-558 p. ISBN 978-1-4812-1858-0. One citation in page 161.

VOUFO, L.; ZALEWSKI, M.; LUMSDAINE, A. ConceptClang: An implementation of c++ concepts in clang. In: *Proceedings of the Seventh ACM SIGPLAN Workshop on Generic Programming*. New York, NY, USA: ACM, 2011. (WGP '11), p. 71–82. ISBN 978-1-4503-0861-8. Available from Internet: <http://doi-acm-org.ez31.periodicos.capes.gov.br/10.1145/2036918.2036929>. One citation in page 153.

WARD, M. P. Language oriented programming. *Software — Concepts and Tools*, v. 15, p. 147–161, 1995. 2 citations in pages 45 and 160.

WEHR, S.; LäMMEL, R.; THIEMANN, P. Javagi: Generalized interfaces for java. In: *Proceedings of the 21st European Conference on Object-Oriented Programming*. Berlin, Heidelberg: Springer-Verlag, 2007. (ECOOP'07), p. 347–372. ISBN 3-540-73588-7, 978-3-540-73588-5. Available from Internet: <http://dl-acm-org.ez31.periodicos.capes.gov.br/citation.cfm?id=2394758.2394783>. One citation in page 153.

WINDOWBUILDER. 2018. Available from Internet: <http://www.eclipse.org/windowbuilder/>. One citation in page 163.

XTEND. *Xtend — Modernized Java*. 2020. Available from Internet: <https://www.eclipse.org/xtend/>. 3 citations in pages 123, 124, and 135.

ZHANG, K. *Visual Languages and Applications*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN 0387298134. One citation in page 164.

ZHANG, Y. et al. Lightweight, flexible object-oriented generics. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 50, n. 6, p. 436–445, jun. 2015. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/2813885.2738008>. One citation in page 153.

APPENDIX A – The Cyan Language

This chapter is an introduction to language Cyan, which is the language employed in this thesis. The text discloses just enough for the understanding of the thesis. Further information can be found in the language manual (GUIMARÃES, 2020) and at `www.cyan-lang.org`. It is assumed that the reader knows at least one object-oriented language. The knowledge of Java (GOSLING et al., 2014), C♯ (C#..., 2020), Smalltalk (GOLDBERG; ROBSON, 1983), or a language with related characteristics makes this chapter easier to understand.

A.1 Prototypes

Cyan is a statically typed object-oriented language based on prototypes. A prototype is a literal object that plays a role similar to that of a class in languages based on classes like C++ (STROUSTRUP, 2013), Java (GOSLING et al., 2014), or C♯ (C#..., 2020). The following code declares a prototype `Point` between lines 3 and 16. In line 1 is the mandatory declaration of which package the prototype belongs, similar to other languages like Java. All prototypes of the `main` package must be in files of a "`main`" directory. Each file must have a single prototype and the file name must be the name of the prototype plus the "`.cyan`" extension.

```
1  package main
2
3  object Point
4      func init: Int x, Int y {
5          self.x = x;
6          self.y = y
7      }
8      func getX -> Int = x;
9      func getY -> Int { return y }
10     func x: Int x
11          y: Int y {
12         self.x = x;
13         self.y = y
14     }
15     var Int x, y
16 end
```

Returning to the example, line 15 declares two *fields* or *instance variables* `x` and `y` of type `Int`. These fields are called by various names in the literature, such as "data members" and "attributes". The keyword `var` before the declaration indicates that fields can change their values through assignments. If `var` is not used or "`let`" is used, then the field is read-only (more on this will be explained in a moment). The type `Int` is one of the basic types. The others are `Boolean`, `Char`, `Short`, `Long`, `Float`, `Double`, and `String`. With the exception of `String`, the basic types are the basic Java types with the first letter capitalized. These types are prototypes in Cyan and are in the package "`cyan.lang`" which is automatically imported by every *source file* in Cyan. A package can have a name composed of identifiers separated by "`.`". The package directory must be the package name with "`.`" replaced by the operating system directory separator. So the package "`cyan.lang`" is in the "`cyan\lang`" directory in Windows.

A *source file* or *compilation unit* is a file containing code, which is composed of a package name, import statements, and a single prototype.

A *constructor* is declared between lines 4 and 7 with "`func`". A constructor is responsible for initializing the fields of a prototype. This constructor has two parameters, `x` and `y`. `self` in lines 5 and 6 is a reference to the object being created. It is the equivalent of `this` of Java, C♯ and C++. Smalltalk uses the same word for the same concept. Cyan requires that every field is initialized in the declaration or in each of the constructors — there are no default values for initializing variables like `null` for fields whose types are classes. The example below shows how `x` and `y` could be initialized in their declarations.

```
package main

object Point
    ...
    var Int x = 0;
    var Int y = 0;
end
```

A field declared with `let` or without `var` is read-only. It can receive an expression in its declaration and be initialized into a constructor. But it cannot be used on the left side of an assignment in a regular method. A prototype may have more than one constructor (see the rules for this in the language manual). In particular, a prototype may declare two constructors with different number of parameters.

Lines 8 and 9 show the declaration of the methods `getX` and `getY`, which start with the `func` keyword. The return type must be given after "`->`". `getX` returns the expression after "`=`". `getY` uses the "`return`" command to return an expression. Both forms are equivalent. A constructor is declared as any method although it must have

the name `init` (if it has no parameters) or `init:` (with parameters). The Cyan compiler removes the constructor from the generated code and adds a new method `new` (for `init`) or `new:` (for `init:`). In the latter case, the method has exactly the same parameters as the corresponding `init:` method. The `new` or `new:` method creates a prototype object and executes the same instructions as the corresponding `init` or `init:` method. There are numerous rules regarding constructors that will not be given here. The rules guarantee that an object created with a constructor always has all fields initialized and that a field is never used before it receives a value, as can happens with almost all other object-oriented languages as Java (QI; MYERS, 2009).

The `Point` prototype declares a method between lines 10 and 14. This syntax is based on that of Smalltalk. "`x:`" and "`y:`" are called "*keywords*" or "*method keywords*", which occasionally results in confusion with the usual meaning of this term, which is a keyword of **language** (such as `object` and `var`). Each keyword in a method can have zero or more parameters in Cyan (Smalltalk requires them to have exactly one). The return type of this `x:y:` method was not specified. As a result, it is considered to be `Nil`. Whenever the return type is `Nil` it is not necessary to have a `return` command. Methods without parameters and without "`:`" after the identifier, such as `getX`, are called "*unary methods*". A method with a single keyword and without parameters, as `getX:`, is illegal. The name of a method is composed of keyword names and number of parameters of each keyword. Therefore, the name of the method between lines 10 and 14 is "`x:1 y:1`".

A `Point` object can be created as in line 6 of the following example.

```
1  package main
2
3  object Program
4      func run {
5          var Point p;
6          p = Point new: 5, 7;
7          var n = p getX;
8          n println;
9      }
10 end
```

"`Point new: 5, 7`" is the sending of *message* "`new: 5, 7`" to the object `Point`. This is also called *message passing* or *message send*. The numbers are the arguments of the message and the *selector of the message* is "`new:`". Note that `Point` plays the role of *type* when declaring a variable or parameter (line 5 of this example) and *object* inside an expression (line 6). This is the fundamental difference between Cyan and a class-based language: a prototype is an object that already exists early in program execution. Therefore, it can be used in expressions and be the target of messages. Cyan is statically typed,

so the compiler checks whether `Point` has a `new:` method that accepts two integers as parameters.

At line 7, message `getX` is sent to `p`. The result is assigned to `n`. Purposely, object-oriented nomenclature confuses compilation and execution time in order to decrease the number of names for the description of facts. For example, "message passing" or "message send" is something that occurs only at runtime. But we say that, at line 7 of the example, a static *text*, there is a message passing. In fact, there is none, but there will be one when the code corresponding to this line is executed at runtime. We say that `getX` was sent to `p`. But `p` is a variable that refers to a dynamically created (at runtime) object. Accordingly, the correct phrase would be "the message `getX` will be sent, at runtime, to the object referenced by variable `p`". The name of a prototype such as `Point` is a reference to an object created prior to the execution of the program. It is easy to explain this by examining the translation of Cyan to Java, which is the target language of the Cyan compiler. A prototype `Point` causes the generation of a `_Point` class which has a static variable of `_Point` type:

```
...
class _Point {
    public static _Point prototype = new _Point();
    public _Point() { }
    ...
}
```

Any references to `Point` in an expression are translated, in the generated code, to

```
_Point.prototype
```

That means that fields `x` and `y` of `Point` can be accessed before they are initialized, a flaw that will be corrected in due time.

Any variable *refers* to an object, even when its type is a basic prototype such as `Int` or `Char`. In C++ language terminology, all variables are pointers. And a prototype like `Int` is an object like any other of the language, it can be used in expressions. `Int` has a value that is `0`. Therefore, you can write

```
var Int iAmNothing = Int*Int + Int - Int;
```

The first `Int` is a type and the following are `0`. The other prototypes or basic types have the expected values (e.g. verb|""| for `String`). The operators `*`, `+`, `<`, `=<`, etc. are all method names. So in this last example we have three message passings.

You can use `Point(5, 7)` instead of `Point: 5, 7` and `A()` instead of `A new`, assuming there is a prototype `A` with an `init` method. All prototypes have a method `clone` that returns a shallow copy (*shallow clone*) of the object. That is, only the fields of

the old object are copied to the new object created by `clone`, the objects referenced by the fields are not copied.

Cyan uses language-C like comments: `//` for a comment to the end of the line and everything that is between `/*` and `*/` is a comment. In the latter case, nested comments are allowed.

By default, methods are public, it is not necessary to put "`public`" before `func`. In addition to public, a method can be qualified with the kewords `private`, `protected`, or `package`. A `private` method can only be called from within the prototype where it is declared. A `protected` method can be called in its prototype and subprototypes of the prototype where it was declared. The concept of subprototype will be seen shortly. A `package` method is visible in the package of its prototype. Currently, fields can only be private, although this should change.

A field is shared by all objects of a prototype if it is declared with keyword `shared`:

```
object Planeta
    shared var Int counter = 0;
    ...
end
```

A `shared` field must be initialized in its declaration or in a special method called `initShared`. There are draconian limitations to these initializations for preventing the use of a field that has not yet been initialized. There are no `shared` methods, although some methods can only be called by sending messages to prototypes. These methods are preceded by

```
@prototypeCallOnly
```

The Cyan compiler, Saci, is made in Java and generates Java code. This facilitates interoperability between the two languages. A Cyan compilation unit can import Java packages and use Java classes as types of fields, parameters, method return value types, and local variables. Java values of the basic and wrapper types are automatically converted to values of the corresponding types of Cyan. And vice versa. There are some limitations in the interoperability between the languages: a Cyan prototype cannot inherit from a Java class or implement a Java interface, there is no conversion from a Cyan function to a Java lambda, and the `for` statement can only be used with Cyan collections. The following example shows some interactions between Java classes and Cyan code.

```
package main

import java.lang
import java.util
```

```
object Test
    func javaTest {

        // cast Java Integer to Cyan Int
        var Int k = Integer(5);
        var Integer ki = k; // and vice versa

        // prototype Int of Cyan used as
        // parameter to generic classes Set and
        // HashSet of Java
        var java.util.Set<Int> iset = java.util.HashSet<Int>();

        iset add: 0;
        iset add: 1;
        iset add: 2;
        var Boolean b;
        b = iset contains: 0;
        assert b;
        // 'iset contains: 1' returns Java 'boolean'
        // that is cast to Cyan 'Boolean'
        b = iset contains: 1;
        // macros do not cast Java to Cyan values
        // 'assert' is a macro
        assert b;

        b = iset contains: 2;
        assert b;
        b = iset contains: -1;
        assert !b;
        b = iset contains: 4;
        assert ! b;
        var java.lang.Boolean javaBooleanVar = true;
        if javaBooleanVar {
            "This will be printed" println;
        }

        var java.lang.Integer integer = Integer(5);
        // in this expression, argument to '==' of Cyan
        // is a Java value of type Integer
```

```
        assert 5 == integer;

    }
end
```

Saci takes Cyan code as input and produces Java code that can then be imported by Java code. A prototype `Proto` generates a class `_Proto` in Java. A Cyan method with keyword `with:` with two parameters and keyword `do:` with three parameters generates a Java method `at2do3____`.

A.2 Repetition, Decision, and Literals

Cyan supports an usual `if` command. The parentheses are not needed for the expression.

```
    // reads from the standard input
var age = In readInt;
if age < 0 {
    "illegal age" println;
    System exit: 1; // ends the program
}

if age < 3 {
    "baby" println
}
else if age >= 3 && age < 13 {
    "child" println
}
else if age >= 13 && age < 19 {
    Out println: "teenager"
}
else {
    Out println: "adult"
}
```

The `;` is optional before `}` and after all the commands that start with a keyword (`if`, `while`, etc.), except variable declaration (which is considered a command). The operators `&&` and `||` are **not** methods. The second expression of `&&` is evaluated only if the first one is `true` and the second expression of `||` is only evaluated if the first one is `false`. The curly brackets are required in statement `while` but not in `repeat-until`, exemplified below.

```
var n = 10;
while n > 0 {
    n print;
    " " print;
    --n
}
var sum = 0;
   // there is a repeat-until command
repeat
    sum = sum + n;
    ++n;
until n >= 10;
sum println;
```

Surprisingly, the language requires that `}` be on the same line as `{` or in the same column as the initial keyword of the command, be it `if`, `while`, or any other that begins with a keyword and uses curly brackets.

`Tuple` is the generic prototype for tuples of elements. A tuple is a set of ordered elements that can be of different types (very similar to C's struct). Literal tuples are given between `[.` and `.]` as in the example.

```
let Tuple<String, Int> person = [. "Newton", 1642 .];
pessoa f1 println; // prints "Newton"
pessoa f2: 1643; // oops, new calendar
```

Method `f`i is used to retrieve the i^{th} value stored in the tuple. Method `f`i`:` initialize the i^{th} tuple field with a new value. The tuple fields can have names other than `f1`, `f2`, etc.

```
let Tuple<name, String, birthyear, Int> person =
     [. "Newton", 1642 .];
person name println; // prints "Newton"
person birthyear: 1643; // oops, new calendar
```

`Array` is a generic prototype for vectors declared in package `cyan.lang`. The prototype `IMap<Key, Value>` is an interface to maps (also called dictionaries). Interfaces in Cyan are seen in a later section. At this point, it is sufficient to know that they are similar to interfaces of Java and C♯. Both `Array` and `IMap` can be used with the `for` command, which allows the iteration of a data structure:

```
    // sum all array elements
func sumArray: Array<Int> array -> Int {
    var sum = 0;
```

```
    for elem in array {
        sum = sum + elem
    }
    return sum
}
```

Cyan supports arrays and literal maps. The elements are given between `[` and `]`. In the maps, symbol `->` separates the key from the value.

```
   /* in a 'for' statement, the type of the
      element may be given or not
   */
for Int elem in [ 2, 3, 5, 7 ] {
    elem println
}
let IMap<String, Int> map = [ "I" -> 1, "V" -> 5,
     "X" -> 10, "L" -> 50, "C" -> 100, "D" -> 500, "M" -> 1000 ];
  // method asArray casts every pair (key, value)
  // in a tuple
for elem in map asArray {
    Out println: elem key ++ " worth " ++ elem value;
}
```

The `++` method converts the message receiver, left side, and also the argument, right side, into strings using the `asString` method that every object has. Hence, it returns the concatenation of the two strings.

Interval objects are returned by methods `..` of prototypes `Char`, `Byte`, `Int`, `Short` and `Long`.

```
var sum = 0;
for elem in 1..10 {
    sum = sum + elem
}
assert sum == 55;
```

The `assert` macro issues a message if its expression is `false` at runtime. It does not exit the program.

Of course, tuples, arrays, and maps can be nested:

```
var t = [.
      portuguese = [ [ 0 -> "zero", 1 -> "um" ], [ 10 -> "dez" ] ],
      english = [ [ 0 -> "zero", 1 -> "one" ], [ 10 -> "ten" ] ]
```

```
.];
```

Strings can use escape characters from the Java language: \n, \r, \t, etc. A literal string that does not consider these characters must be prefixed by n or N as in

```
let iHaveSize4 = n"\n\t";
```

There is a simplified way of expressing strings that use only letters, numbers, _ (underscore), . (Dot) and : (colon). The sequence of characters should be preceded by # as in the example below.

```
let sym = #cyan.lang;
let method = #at:put:;
#0zero println;
```

The type of this kind of literal is String although they are called **symbols**.

Strings can be multi-line:

```
var quadrilha = """
    João loved Teresa that loved Raimundo
    that loved Maria that loved Joaquim
    that loved Lili that loved no one.
    ...
    """;
```

In this kind of string, escape characters are not considered. Therefore, \n are two characters, \ and n.

A.3 Inheritance, Nil, and Interfaces

Inheritance in Cyan is specified through keyword extends.

```
open object Shape
    func getColor -> Int = color;
    func setColor: Int color { self.color = color }
    var Int color = 0;
end

object Square extends Shape
    func init: Int side { self.side = side }
    func getSide -> Int = side;
    func setSide: Int side { self.side = side }
```

```
    var Int side = 0;
end
```

In this example, two prototypes were presented together for didactic reasons. In a program, they should be in separate files and preceded by the package declaration. `Square`, called a subprototype, inherits all methods and fields from `Shape`, called a superprototype. If a method of the superprototype is redefined in the subprototype, it must be preceded by the keyword `override`. The word `open`, which is not a keyword, precedes the declaration of `Shape` to indicate that this prototype can be inherited. Without this word the prototype would be final, it could not be inherited.

The language supports only single inheritance, each prototype has at most a sole superprototype. All prototypes in Cyan, except `Nil` and including the basic types `Int`, `Char`, etc., inherit directly or indirectly from `Any`.

Usually object-oriented languages use a value `null`, `NULL` or `nil` that is polymorphic, it has infinite types. The type of this value is a subtype of all other types or at least subtype of class types. In Java, for example, `null` can be assigned to a variable whose declared type is a class:

```
String s;
s = null; // ok
```

In Ceylon (KING, 2022), the value `null` is the only object of class `Null` and cannot be assigned to a variable whose type is a class. This value can only be assigned to a variable whose type is a union:

```
  // 'variable' is a keyword in Ceylon
variable Null|String s;
s = null; // ok
s = "ok"; // ok
```

The Cyan prototype `Nil` was partially based on the Ceylon class `Null`. `Nil` is not anyone's subprototype or superprototype. It is a singleton object, it is not possible to create objects from this prototype and it cannot be inherited. As a consequence, a variable of type `Nil` can only receive, at runtime, the value `Nil` (the prototype) in an assignment. Therefore, it does not make sense to declare a variable of this type.

`Nil` can be used with union types, declared using `|` as in

```
var String|Nil s;
s = "I am a string";
s println;
s = Nil;
  // s does not hold a string anymore
```

The type of s is an option between Nil and String and this variable can receive values of both types in an assignment. However, messages cannot be sent to expressions (including variables) whose types are unions, unless the messages are == and !=.

There are two ways to handle unions containing Nil: using the type or cast commands. The first is similar to a switch of the C language, but with types.

```
1   var String|Nil s;
2   s = Nil;
3   s = "I am a string";
4
5   type s
6       case String str {
7          (str substring: 1) println;
8       }
9       case Nil nil {
10         nil println
11      }
12
13  cast str = s {
14     (str substring: 1) println;
15  }
```

Line 7 is executed because s refers to a String object. The value of s is assigned to the variable str. If s referred to Nil, s would be assigned to the variable nil of type Nil and line 10 would be executed.

The cast command on line 13 checks whether s is Nil or not — currently this command can be used only with unions of type Nil|T or T|Nil in which T is a prototype. Since s refers to a String object, line 14 is executed. The compiler deduces that the type of str is String because of the type of s.

Nil is the return type of all methods that do not explicitly declare the return type, playing a role similar to void of C-based languages (KERNIGHAN, 1988). This is very convenient because all methods always return something. If the return type is Nil, they always return the prototype Nil. In this case, the command return is optional.

A union type can be used as the type of a variable, parameter, or method return value. It cannot be used as a superprototype name, after keyword extends.

```
1   var Int|String|Char sic;
2   sic = 0;
3   sic = "I am a string";
4   sic = 'A';
```

```
5
6     type sic
7         case Int n {
8            (2*n + 1) println;
9         }
10        case String s {
11           "I am the string $s" println
12        }
13        case Char ch {
14            "ch = $ch" println;
15        }
```

In lines 11 and 14, $ before a variable name inserts, in the literal string, the result of the transformation of the variable into a string. For that reason, the tests below do not issue warnings.

```
var ch = 'a';
assert "ch = $ch" == "ch = " ++ ch;
assert "ch = $ch" == "ch = " ++ (ch asString);
```

Method == compares two basic values, as integers and characters, by values, not addresses.

There is a generic Union prototype that works as a union of types but each one has a tag:

```
Union<tag1, T1, tag2, T2, ..., tagn, Tn>
```

tagi is any lowercase identifier and Ti is a prototype different from Nil. The same type may appear twice because the tags differentiate them. Method tagi: Ti initializes an union object, that should be created with new before used.

```
var Union<f1, Int, f2, String> unity =
    Union<f1, Int, f2, String> new;
unity f1: 0;
unity f2: "now a string";
```

Union elements can only be accessed using statement type-case. For each pair tag-type of the union there should be a case clause whose variable name should be the tag name. The case clause order should be the same as the tags in the union.

```
var Union<f1, Int, f2, String> unity =
    Union<f1, Int, f2, String> new;
unity f1: 0;
type unity
```

```
        case Int f1 {
            assert true;
        }
        case String f2 {
            assert false;
        }
    unity f2: "now a string";
    type unity
        case Int f1 {
            assert false;
        }
        case String f2 {
            assert true;
        }
```

An abstract prototype is declared with the `abstract` keyword before `object` and **may** have abstract methods, methods declared with `abstract` preceding `func`. Abstract methods should not have a body, they should only have the declaration of the keywords with the return type:

```
abstract object Animal
    abstract func eat: Food good
    abstract func walk: Int numMeters
end
```

When a non-abstract prototype inherits from an abstract prototype, it must define all inherited abstract methods, as usual.

A prototype can be declared with the `interface` keyword instead of `object`. In this case, the body of methods should not be given. A prototype that is not an interface can **implement** any number of superinterfaces:

```
interface Drawable
    func draw: Int backgroundColor
end
object Square extends Shape implements Drawable
    func init: Int side { self.side = side }
    func getSide -> Int = side;
    func setSide: Int side { self.side = side }
    func draw: Int backgroundColor {
       // code for 'draw'
    }
```

```
    var Int side = 0;
end
```

The prototype that implements the interface must define its methods. Or it should be declared with the `abstract` keyword. An interface can inherit from one or more interfaces using the word `extends`, but it cannot implement, with `implements`, any interface. A constructor cannot be defined in an interface and its `clone` method throws an exception, so no new objects can be created from one.

Throughout this text, the word "prototype" is used for both interfaces and non-interface prototypes. The reason is that an interface is a regular prototype for which some special rules apply. But an interface is an object that **defines** its methods, although they throw an exception at runtime. In the code below, message `size` is sent to an object that refers to an interface. That does not cause a compile-time error. At runtime, method `size` of the interface is called and it throws an exception.

```
var anInterface = IMap<String, Int>;
anInterface size println;
```

A.4 Dynamic Typing

Cyan is statically typed with optional dynamic typing. Therefore, the language has *gradual typing*. When using static typing, the compiler checks whether the type of an expression that receives a message has a method that *matches* the message: the name is the same, the method can accept the real message arguments. This check is not made if the receiver expression has the type `Dyn`. It instructs the compiler not to make type checking. The compiler does not check if messages sent to a variable of type `Dyn` has a method that matches the message passing.

```
var Dyn troublemaker = 0;
troublemaker setManagerName: "Bill Jobs";
```

This code compiles correctly but there will be a runtime error because `Int` does not have a `setManagerName:` method. There is another way to tell the compiler not to do type checking in a message passing: just prefix the keywords with "`?`" as in

```
var Int troublemaker = 0;
troublemaker ?setManagerName: "Bill Jobs";
```

Without the `?`, there would be a compilation error.

A local variable that is initialized in its declaration with an expression has the type of that expression. If neither the type nor the expression is given, the variable has the type Dyn.

```
   // variable has type Dyn
var troublemaker;
troublemaker = 0;
troublemaker setManagerName: "Bill Jobs";

   // age has type Int
var age = 8;
```

A parameter declared without the type has type Dyn as well. So programs in Cyan can be done as if the language were dynamically typed. Just do not put the type of parameters and variables that are not initialized in their declaration and use type Dyn in the following situations:

a) in local variables initialized in their declarations;

b) as the type of prototype fields.

A good programming technique is to make an initial version of a program with dynamic typing and then convert it to static typing.

The compiler makes the necessary checks when an object of type Dyn is assigned to a variable of another type. If the object type is not a subtype of the variable type, a runtime error occurs. Similarly, Dyn type expressions can be used in if, while, for, and repeat-until commands.

```
var Dyn s = "runtime error";
var Int n;
n = s; // runtime error!
```

A method whose name will only be known at runtime can be called using operator ` . A unary method whose name is in the String variable s is called using the syntax

```
obj `s
```

A method whose keywords are in String variables s1, ... sn is called using

```
obj `s1: p1 ... `sn: pn
```

The following code shows a real example.

```
var sz = "size";
var atStr = "at";
var putStr = "put";
var strArray = [ "red", "green", "blue" ];
   // call method 'size' of Array<String>
```

```
assert strArray `sz == 3;
   // call method 'at:' of Array<String>
assert strArray `atStr: 0 == "red";
   // call method 'at: put:' of Array<String>
strArray `atStr: 0 `putStr: "vermelho";
   // call method 'at:' of Array<String>
assert strArray `atStr: 0 == "vermelho";

let map = [ "+" -> 16, "-" -> 8, "/" -> 3, "*" -> 48 ];
for op in [ "+", "-", "/", "*" ] {
      // check the result of the dynamic call
      // with the one stored in 'map'
    cast result = map[op] {
       assert result == 12 `op: 4;
    }
}
```

For the curious reader, in the package `cyan.lang` there is a prototype `DTuple` that simulates the runtime creation of prototype fields.

```
var Dyn t = DTuple new;
t name: "Newton";
t age: 85;
Out println: t name; // "Newton"
Out println: t age; // 85
```

A.5 Generic Prototypes

A generic prototype in Cyan is the equivalent of a *template* class in C++ (GREGOR et al., 2006) (STROUSTRUP, 2013) and a generic class in Java, Scala (ODERSKY et al., 2004), and C♯ (C#..., 2020). The following example shows a generic prototype `Box<T>` of a file called "`Box(1).cyan`". The number of prototype parameters is within parentheses.

```
package main

object Box<T>
    func init: T elem { self.elem = elem }
    func get -> T = elem;
    func set: T elem { self.elem = elem }
    var T elem
end
```

This prototype is instantiated when a type is supplied as parameter, as in lines 6 and 8of the example.

```
1  package test
2  import main
3
4  object Test
5      func test {
6          var Box<Int> box = Box<Int> new: 5;
7          box get println;
8          Box<Int> prototypeName println;
9      }
10 end
```

During semantic analysis, the Cyan compiler checks the type of variable `box`, `Box<Int>`. Since there is no prototype with this name, Saci creates a new prototype replacing `T` with `Int` in the generic prototype. This is the scheme used by C++ and C♯, unlike Java and Scala, which use the same code for all instantiations of a parameterized class.

A generic prototype parameter `T` can be used in the body of a generic prototype in several ways:

a) as type of variables, parameters, or method return, after keyword `extends` or `implements`, or as a parameter to another generic prototype instantiation;

b) after `#` to compose a symbol: `#T`. If there are other characters, as in `#Tx`, the symbol is considered to be not related to the parameter `T`;

c) as a parameter to a metaobject annotation (yet to be seen);

d) as a method keyword.

In all of these uses, the compiler changes the formal parameter `T` by the actual argument used in the instantiation of the prototype. That is exemplified below.

```
package main

object EveryUse<T, R>
       // R used as method keyword
       // T used as a symbol
    func R: Int n -> String = #T;
      // T as a return type and a
      // prototype in an expression
    func newObject -> T = T new;
    func tPlusr -> String {
          // T and R as annotation parameters
```

```
        return @symbolToString(T) ++ @symbolToString(R);
    }
end
```

There is one case that was not covered by the above explanation: the use of the parameter inside the DSL code attached to an annotation. This case is explained in section 2.2.

There are numerous variations of generic prototypes that have not been described in this section:

a) there may be more than one set of < and >;

b) there is support for a variable number of arguments;

c) a non-generic prototype can use the same syntax as a generic prototype, but with real arguments in place of the generic parameters. Language C++ also supports this feature;

d) several generic prototypes that take different number of parameters can have the same name. For example, there may be `Box<T>` and `Box<T, R>`.

A.6 Anonymous Functions

An anonymous function is a unnamed literal function. It is called *blocks* in Smalltalk (GOLDBERG; ROBSON, 1983) and *lambdas* in C++ (STROUSTRUP, 2013), Java (GOSLING et al., 2014) and Kotlin (JETBRAINS, 2022). In Cyan, an anonymous function or simply "*function*" is declared with the following syntax. `[` and `]` is used for optional symbols, anything between `{` and `}` can be repeated zero or more times.

```
‘‘{’’ [ ‘‘(:’’ { ParameterDec } [ ‘‘->’’ ReturnType ] ‘‘:)’’ ]
  statementList ‘‘}’’
```

The function is given between `{` and `}`. If there are parameters, they appear between `(:` and `:)` After the last parameter, the return value type may be given using `-> ReturnType`. However, the return type can always be deduced from the values returned by the function and "`-> ReturnType`" is optional. The function body, a sequence of statements, is given after `:)` or after `{` if there is no pair `(: :)`. The function returns an expression `expr` using "`^expr`". The `return` statement for method return cannot appear inside an anonymous function.

```
{ (: Int n -> Int :) ^n*n }
```

The first line of the next example assigns a function to variable `sqr`.

```
var sqr = { (: Int n :) ^n*n };
assert sqr eval: 5 == 25;
```

The function is called by sending message `eval:` to `sqr`. Note that the operator precedence of `==` is smaller than that of message passing. Therefore, the expression of macro `assert` is equivalent to

```
(sqr eval: 5) == 25
```

An anonymous function without parameters is called with the `eval` method as in

```
let f = { "print this" println };
f eval;
{ /* do nothing */ } eval;
```

Anonymous functions are literal objects. For each literal function, the compiler creates a new prototype. This prototype inherits from an instantiation of the generic `Function` prototype. The parameters used with `Function` depend on the parameter types and return type of the function. As an example, the function

```
{ (: Int n -> Int :) ^n*n }
```

is defined as

```
object Fun_1__(Test self__) extends Function<Int, Int>
    override
    func eval: Int n -> Int { return n*n; }
end
```

The declaration "`Test self__`", after the prototype name `Fun_1__`, is a field that will refer to `self` at the moment the function is created.

A parameterless function that returns nothing or `Nil` has the type

```
Function<Nil>
```

Note that, although function `{ }` is the sole object of a prototype that inherits from `Function<Nil>`, its type is `Function<Nil>`.

A function that takes parameters whose types are `T1`, `T2`, ... `Tn` and whose return type is `R` has type `Function<T1, T2 , ..., Tn, R>`. This prototype inherits from `Any`, the top of the hierarchy. Hence, functions are first-class entities that can be assigned to variables, passed as parameters, and stored in prototype fields.

An anonymous function can use visible local variables. This does not cause problems since any local variable used within some function is allocated dynamically. `self` is visible inside anonymous functions. So a function can use the fields of the prototype where it is and send messages to `self`. The examples below show some additional features of functions.

```
package main

object Test
    func funcTest {
        var sum = 0;
        [ 2, 3, 5, 7 ] foreach: { (: Int n :)
            sum = sum + n;
            ++count
            };
        assert sum == 2 + 3 + 5 + 7;
          // if with anonymous functions,
          // as in Smalltalk
        (sum < 10) ifTrue: {
            self incCount;
            "sum < 10" println
            }
            ifFalse: {
               incCount;
               "sum >= 10" println
            };
          // while with anonymous functions
        { ^sum > 0 } whileTrue: {
            ++count;
            --sum
            };
        Out println: "count = $count";
    }
    func incCount { ++count }
       // count the number of times
       // functions were called
    var Int count = 0;
end
```

A.7 The Exception Handling System

The Cyan exception handling system was based on the exception system of language Green (GUIMARÃES, 2013). It supports only unchecked exceptions. That is, the compiler never points out that an exception may be thrown and not caught as in language Java, the only major language that supports this kind of exceptions.

The Cyan exception handling system is made exclusively using message passing. Exceptions are objects whose prototypes inherit direct or indirectly from prototype `CyException`.

```
package main

object IllegalTriangle(Double _sideA, Double _sideB, Double _sideC)
       extends CyException
    func sideA -> Double = _sideA;
    func sideB -> Double = _sideB;
    func sideC -> Double = _sideC;
end
```

Prototype `IllegalTriangle` declares three fields just after its name and inherits from `CyException`. An object of it can be thrown using method `throw:` inherited from `Any` by every prototype but `Nil`. Hence, exception `IllegalTriangle` can be thrown as in the code

```
    self throw: IllegalTriangle(13.0, 4.0, 2.0);
```

`self` is optional since a message passing without a receiver has `self` as receiver.

Exception treatment in Cyan is made through methods of prototype `Function<Nil>` of package "`cyan.lang`". This prototype is not an instantiation of prototype `Function<T>`, it is a non-generic prototype that uses the generic prototype syntax. Method "`catch: Any`" of `Function<Nil>` takes an object of `Any` and its subprototypes as parameter. A metaobject annotation (yet to be seen) assures that either the parameter type is `Dyn` or it has an `eval:` method that accepts an object of `CyException` as parameter. Let us see an example of exception handling.

```
1  package main
2
3  import cyan.math
4
5  object Test
6
7      func throwTest {
8          {
9              "Type in the sides of the triangle" println;
10             var aa = In readDouble;
11             var bb = In readDouble;
12             var cc = In readDouble;
13             if aa < bb + cc || bb < aa + cc || cc < aa + bb {
```

```
14                throw: IllegalTriangle(aa, bb, cc)
15            }
16            Out println: "The area of the triangle is " ++ (area: aa, bb,
                  cc);
17        } catch: { (: IllegalTriangle e :)
18            let a = e sideA;
19            let b = e sideB;
20            let c = e sideC;
21            Out println:
22               "The triangle is illegal because it has one side " ++
23               "bigger than the sum of the other two: $a, $b e $c";
24        };
25
26    }
27
28    func area: Double a, Double b, Double c -> Double {
29        let p = (a + b + c)/2.0;
30        return Math sqrt: p*(p - a)*(p - b)*(p - c);
31    }
32 end
```

Between line 8 and the } of line 17, there is an anonymous function that does not take parameters and does not return anything. As a result, it has type

```
Function<Nil>
```

Keyword `catch:` of line 17 is a message send that takes another function as parameter. This one takes an `IllegalTriangle` as parameter and ends in line 24. A ; is demanded in line 24 because lines 8 to 24 is a single message passing.

In line 14, exception `IllegalTriangle` is thrown by method "`throw:`". At runtime this triggers a search for a `catch:` method that takes a parameter that:

a) has an `eval:` method;

b) this method accepts an object of type `IllegalTriangle`.

The function passed as parameter to `catch:` obeys these restrictions and is called. The type of this object is `Function<IllegalTriangle, Nil>` and it has a method

```
func eval: IllegalTriangle e {
    let a = e sideA;
    let b = e sideB;
    let c = e sideC;
    Out println:
```

```
            "The triangle is illegal because it has one side " ++
            "bigger than the sum of the other two: $a, $b e $c";
    }
```

Therefore, in this example, the message passing of line 14 transfer the flow of control to method `eval:` of the function passed as parameter to `catch:`. This method has the content of the anonymous function that is between lines 17 and 24. The search for a `catch:` method in Cyan is exactly equal to the search for a `catch` clause in Java, C++, and so on. In particular, there may be more than one `catch:` keyword in the message passing and they are searched for in the order they appear:

```
    {
       // some more complex code
    }
    catch: { (: IllegalTriangle e :) ... }
    catch: { (: ZeroSideTriangle e :) ... }
    catch: { (: NegSideTriangle e :) ... }
```

Returning to the prototype `Test`, the parameter to `catch:` may be any object that has an `eval:` method that accepts an object of type `CyException` or its subprototypes as parameter. Prototype `CatchIllegalTriangle` obeys these restrictions.

```
package main

object CatchIllegalTriangle
    func eval: IllegalTriangle e {
        let a = e sideA;
        let b = e sideB;
        let c = e sideC;
        Out println:
           "The triangle is illegal because it has one side " ++
           "bigger than the sum of the other two: $a, $b e $c";
    }
end
```

Therefore, we can write:

```
1           // prototype Test
2         {
3             ...
4                 throw: IllegalTriangle(aa, bb, cc)
5             ...
6         } catch: CatchIllegalTriangle;
```

Prototypes used to catch exceptions are called *catch prototypes*.

Prototype `CatchIllegalTriangle` can be extended to cope with several exceptions at the same time:

```
package main

object CatchIllegalTriangle
    func eval: IllegalTriangle e { ... }
    func eval: ZeroSideTriangle e { ... }
    func eval: NegSideTriangle e { ... }
end
```

The search for an adequate `eval:` method is made in the order of declaration in the prototype. And the search is made from subprototypes to superprototypes.

Using the Cyan exception handling system one can build a hierarchy of exception treatments. Hence, a library of prototypes can be supplied with *catch prototypes* to treat exceptions thrown by methods of the library. There may be options on what to do on error. For example, package `cyan.io` supplies two catch prototypes that treat IO related errors:

```
CatchExceptionIOMessage
    CatchExitExceptionIOMessage
```

The second inherits from the first one. `CatchExceptionIOMessage` treats exceptions `ExceptionFileNotFound` and `ExceptionIO`. It just issues an error message in the standard output. `CatchExitExceptionIOMessage` issues the message and ends the program.

Prototype `Function<Nil>` has other keywords and methods for exception treatment: `finally:`, `retry:`, and `hideException`. Keyword `finally:` may appear after the last `catch:` and works like keyword `finally` of Java and other languages. `retry:` calls the `eval:` method of the function whenever there is an exception. That is, the function is called again if there is an exception.

```
{
    var n = In readInt;
    if n == 0 {
        throw: ExceptionStr("zero is not allowed")
    }
    Out println: 5/n
}
catch: CatchAll // catch every exception
retry: {
```

```
        "Type in a number different from zero" println
    };
```

`hideException` simply hides any exception thrown in the function.

A.8 The Cyan Interpreter

Statements in Cyan can be generated and interpreted at runtime using prototype `CyanInterpreter` of package `cyan.reflect`. The language is not exactly Cyan because most type checks are not made when the code is "compiled". The interpretation consists only of two phases: parsing and execution. The parsing phase does not make any semantic check. The Cyan statements `cast` and `type` are the only ones that cannot be used in interpreted Cyan. Currently, only prototypes of `cyan.lang` and classes of `java.lang` can be used.

Listing A.1 shows an example Cyan code that is interpreted at runtime. Method `eval:` is called in line 8 with a multi-string with code. The result, `55`, is returned and extracted in lines 15-21. Method `eval:1 self: 1` evaluates a string assuming that `self` is the second parameter:

```
CyanInterpreter
    eval: "return self*2"
    self: 5
```

This message send returns `10`. The value to keyword `self:` can be anyone, including objects of basic values and Java objects.

The code interpreted in lines 25-28 of Listing A.1 is

```
5.0*pi*r*r
```

The method of `CyanInterpreter` called here is `eval:1 self:1 varList:1`, which takes a `self` object and an array of tuples consisting of a variable name and a value. The value must have type `Dyn`, this is the reason we declared variable `dynRadius` only to convert `radius` to `Dyn`. Since the parameter to keyword `self:` is `self`, methods of the current prototype, as `pi`, can be called in the call to `eval:1 self:1 varList:1`. The value of `radius` was passed to the interpreter with the name "`r`". It is read-only, there is no way of changing the value of `radius` inside the `eval:1 self:1 varList:1` method.

A.9 The Project File

A Cyan program is described by a **project file** with extension "`pyan`". The content of the file is itself a source code of a language called Pyan and it lists the packages that

Listing A.1 – Testing the Cyan interpreter

```
package main
import cyan.reflect

object InterpreterTest

    func run {
        var Any|Nil anyNil;
        anyNil = CyanInterpreter eval: """
            var sum = 0;
            for n in 1..10 {
                sum = sum + n;
            }
            return sum
            """;

        type anyNil
            case Int value {
                    assert value == 55;
                    "1 + 2 + ... 10 = $value" println
            }
        let radius = 5.0;
        var String code = "pi*r";
        var Dyn dynRadius = radius;
        anyNil = CyanInterpreter
                    eval: "return " ++ code ++ "*r"
                    self: self
                    varList: [ [. "r", dynRadius .] ];
        type anyNil
            case Double area {
                "Area = $area" println;
                    // equal to a precision
                assert area equal: 78.5398163397, 0.1;
            }
    }
    func pi -> Double = 3.14159265359;
end
```

compose the program. An example of a Pyan program follows.

```
1 program at "C:\Dropbox\Cyan\cyanTests\tese"
2     main main.Program // the main prototype
3     package main at "C:\Dropbox\Cyan\cyanTests\tese\main"
4     package cap.dynamic at "C:\Dropbox\Cyan\cyanTests\tese\cap\dynamic"
```

The Cyan compiler is called passing the address of the Pyan file as argument. After `at` of line 1 there should be the address of the directory of the program to be compiled. In line 2, Pyan keyword `main` gives the main prototype. The program execution starts at method "`run`" or "`run: Array<String>`" of this prototype. Lines 3 and 4 lists the two packages of the program, `main` and "`cap.dynamic`". After `at` there should appear the directory of each package as a literal string in which escape characters are not taken into account. If a package does not specify its directory with "`at`", it should be in a subdirectory of the program.

The declaration of the main prototype may be omitted if the main prototype is `Program` of package `main`. Therefore, line 2 can be removed. If all subdirectories of the program directory contains a package of the program, statement `package` may be omitted. Assuming directory

`C:\Dropbox\Cyan\cyanTests\tese`

has only directories `main` and `cap\dynamic` with Cyan source code, lines 3 and 4 of the example can be removed. The project file could be just

```
program at "C:\Dropbox\Cyan\cyanTests\tese"
```

If this is in a "`.pyan`" file of directory

`C:\Dropbox\Cyan\cyanTests\tese`

the project file could be just "`program`". In this case, the compiler can be called passing as argument this directory. It creates a project file "`project.pyan`" with one package for each subdirectory containing Cyan source files.

APPENDIX B – Metaobject property2 Implemented in Java

```
package meta.cyanLang;

import java.util.List;
import meta.AnnotationArgumentsKind;
import meta.AttachedDeclarationKind;
import meta.CyanMetaobjectAtAnnot;
import meta.IAction_afterResTypes;
import meta.ICompiler_afterResTypes;
import meta.ISlotSignature;
import meta.Tuple2;
import meta.WrAnnotation;
import meta.WrAnnotationAt;
import meta.WrFieldDec;


public class CyanMetaobjectProperty2 extends CyanMetaobjectAtAnnot
                                     implements IAction_afterResTypes

    public CyanMetaobjectProperty2() {
        super("property2", AnnotationArgumentsKind.ZeroParameters,
                new AttachedDeclarationKind[] { AttachedDeclarationKi
    }

    @Override public
    Tuple2<StringBuffer, String> afterResTypes_codeToAdd(
            ICompiler_afterResTypes compiler, List<Tuple2<WrAnnotatio

        final StringBuffer s = new StringBuffer();
        final WrAnnotationAt atAnnot = this.getAnnotation();
        final WrFieldDec iv = (WrFieldDec ) atAnnot.getDeclaration();
        final String name = iv.getName();

        String methodGet;
```

```
        String methodSet;
        final String nameUpper = Character.toUpperCase(name.charAt(0)) +
        String ivTypeName = iv.getType().getFullName();
        methodGet = "func get" + nameUpper + " -> " + ivTypeName;
        methodSet = "func set" + nameUpper + ": " + ivTypeName + " other"
        String code = methodGet + " = " + name + ";\n" +
                    methodSet + "\n    { " + "self." + name + " = other }\
        return new Tuple2<StringBuffer, String>(
                new StringBuffer(code),
                methodGet + ";\n" + methodSet + ";\n"
                );
    }
}
```

APPENDIX C – Example of a Metaobject Coded in Cyan

The Cyan code that follows is the prototype of metaobject `createMissingField`.

Annotations of metaobject `createMissingField` should be attached to prototypes. They take an even number of parameters, at least two, consisting of pairs of "constant name" and "constant value". The constants are accessed using `self`.

```
@createMissingField("zero", 0, "pi", 3.14, country, Brasil)
object Program
    func run {
        assert self.zero == 0;
        let radius = 5;
        Out println: "The area of a circle of radius $radius is "
           radius*radius*self.pi;
        ("I live in " ++ self.country) println;
    }
end
```

Method `semAn_replaceGetMissingField` of the metaobject is called whenever a field that does not exist is accessed. Therefore, this method is called when `zero`, `pi`, and `country` are accessed. The method replace the field access by the corresponding constant. Note that the constant name may be given, in the annotation, between quotes or not. The metaobject prototype follows. Method `semAn_replaceSetMissingField` should be specified because it is declared in interface `IActionFieldMissing_semAn`. In Java, the corresponding method in the interface has a default implementation. That means that a Java class does not need to implement it. In Cyan, there is no default method implementation in interfaces yet. Method

`semAn_replaceSetMissingField`

returns an empty string. The compiler will not consider this value as code because it is an empty string.

```
package cyan.reflect

import meta
import java.lang
```

```
open
object CyanMetaobjectCreateMissingField extends CyanMetaobjectAtAnnot
         implements IActionFieldMissing_semAn


    func init {
        super init: "createMissingField", "OneOrMoreParameters",
             [ "prototype" ];
    }


    override
    func semAn_replaceGetMissingField:
            WrExprSelfPeriodIdent fieldToGet,
            WrEnv env
        -> Tuple<String, String, String> {

        let String fieldName = fieldToGet asString substring: 5;
        var annot = self getAnnotation;
        let Int size = annot getJavaParameterList size;
        if size odd {
            addError: "This annotation should be used with a even number
            return [. "", "", "" .]
        }
        var i = 0;
        while i < size {
            let java.lang.String javaParam = ((annot getJavaParameterList
            var String strparam =
                CyanMetaobject removeQuotes: javaParam;

            if strparam == fieldName {
                var value = (annot getRealParameterList) get: i + 1;
                var java.lang.String js = value asString;
                var String s = js;
                js = value getJavaType;
                var String javaType = js;
                var Nil|String cyanTypeNil = [
                    "String" -> "String",
                    "boolean" -> "Boolean",
```

```
                    "Boolean" -> "Boolean",
                    "Character" -> "Char",
                    "char" -> "Char",
                    "Byte" -> "Byte",
                    "byte" -> "Byte",
                    "Integer" -> "Int",
                    "int" -> "Int",
                    "Short" -> "Short",
                    "short" -> "Short",
                    "Long" -> "Long",
                    "long" -> "Long",
                    "Double" -> "Double",
                    "double" -> "Double",
                    "Float" -> "Float",
                    "float" -> "Float"
                     ] get: javaType;
                type cyanTypeNil
                    case String typeName {
                        if typeName == "String" {
                            s = "\"" ++ (CyanMetaobject removeQuotes:
                        }
                        return [. "cyan.lang", typeName, s .]
                    }
                    case Nil nil {
                        addError: "Unknown type: " ++ javaType;
                        return [. "", "", "" .]
                    }

            }
            i = i + 2
        }
        return [. "", "", "" .]
    }

    override
    func semAn_replaceSetMissingField:
            WrExprSelfPeriodIdent fieldToSet,
            WrExpr rightHandSideAssignment,
            WrEnv env
```

```
        -> String

        return ""
    }

end
```

APPENDIX D – Definitions

This chapter gives some definitions of terms used in the article.

Source code is any text with code in any programming language. It may refer to the whole program in text format, a set of text files with code. The common meaning will be "any text with code".

A *Compilation unit* or *source file* is a single file with source code. In Cyan, a *compilation unit* is composed by a package declaration, import statements, and a single prototype.

In a message passing, like `x.m(0)`, the runtime system has to look for a method to be called. The algorithm used for finding an adequate method is called *method dispatch*.

The attached text of an annotation or *the attached DSL code* is the text that is attached to an annotation. In the example that follows, it is the text between `{*` and `*}`.

```
@insertCode{*
    insertCode: "a = 0;";
*}
```

The name of a Cyan method is composed by each keyword method followed by the number of parameters of each keyword. Hence, the name of

```
func with: Int n, String s put: Float f
```

is

```
with:2 put:1
```

An Abstract Syntax Tree (AST) is a data structure for storing data on language declarations and statements. For example, the AST class for a local variable follows.

```
class LocalVariable {
    String name;
    Type type;
    // methods elided
}
```

Each local variable in the program is represented by an object of `LocalVariable`.

An *embedded DSL* is a Domain Specific Language whose code is inside the code of a host language, a general purpose language. A prime example are Groovy (KöNIG, 2007) builders for creating and initializing objects:

```
UndergradCourseBuilder builder = new UndergradCourseBuilder()
```

```
UndergradCourse dcomp = builder.create {
    course('Compiler Construction') {
        student('Wirth'),
        student('John')
    }
    course('Programming Languages') {
        student('Newton'),
        student('Einstein')
    }
}
```

APPENDIX E – Metaobject Classes

All the sections of this chapter, but the last one, contains the complete code of some classes and interfaces to make it easy to understand the text. The last section explains the convertion of a message passing to a grammar method to regular Cyan code.

E.1 Class CyanMetaobjectLineNumber

```
package meta.cyanLang;

import ast.AnnotationArgumentsKind;
import ast.AnnotationAt;
import meta.CyanMetaobjectLiteralObject;
import meta.CyanMetaobjectAtAnnot;
import meta.IAction_parsing;
import meta.ICompilerAction_parsing;

public class CyanMetaobjectLineNumber
        extends CyanMetaobjectAtAnnot
        implements IAction_parsing {

    public CyanMetaobjectLineNumber() {
        super("lineNumber",
              AnnotationArgumentsKind.ZeroParameter);
    }


    @Override
    public StringBuffer
    parsing_codeToAdd( ICompilerAction_parsing compiler ) {
        AnnotationAt annot =
            this.getAnnotation();
        return new StringBuffer(""
            + annot.getSymbolMetaobjectAnnotation()
              .getLineNumber() );
    }
```

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

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

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

E.2 Interface IAbstractCyanCompiler

```
package meta;

import java.util.List;
import java.util.Set;
import ast.Annotation;
import ast.CyanPackage;
import error.FileError;
import lexer.Lexer;
import saci.CompilerManager;
import saci.DirectoryKindPPP;
import saci.NameServer;
import saci.Tuple2;
import saci.Tuple3;
import saci.Tuple4;
import saci.Tuple5;

public interface IAbstractCyanCompiler {

    default String escapeString(String s) {
        return Lexer.escapeJavaString(s);
    }
    default String unescapeString(String s) {
```

```
        return Lexer.unescapeJavaString(s);
    }
    Object getProgramVariable(String variableName);
    Set<String> getProgramVariableSet(String variableName);
    Tuple2<FileError, byte []>
    readBinaryDataFileFromPackage(String fileName,
                                  String packageName);

    FileError writeTestFileTo(StringBuffer data,
        String fileName, String dirName);
    FileError writeTestFileTo(StringBuffer data,
        String fileName, String dirName, String packageName);

    Tuple5<FileError, char[], String, String, CyanPackage>
    readTextFileFromPackage(
            String fileName,
            String extension,
            String packageName,
            DirectoryKindPPP hiddenDirectory,
            int numParameters,
            List<String> realParamList);

    default Tuple5<FileError, char[], String,
                   String, CyanPackage>
    readTextFileFromPackage(
            String fileName,
            String packageName,
            DirectoryKindPPP hiddenDirectory,
            int numParameters,
            List<String> realParamList) { ...
    }

    Tuple4<FileError, char[], String, String>
    readTextFileFromProgram(
            String fileName,
            String extension,
            DirectoryKindPPP hiddenDirectory,
            int numParameters,
            List<String> realParamList);
```

```
    Tuple3<String, String, CyanPackage>
    getAbsolutePathHiddenDirectoryFile(
            String fileName, String packageName,
            DirectoryKindPPP hiddenDirectory);

    String getPackageNameTest();

    FileError writeTextFile(
            char[] charArray,
            String fileName,
            String prototypeFileName,
            String packageName,
            DirectoryKindPPP hiddenDirectory);

    FileError writeTextFile(
            String str,
            String fileName,
            String prototypeFileName,
            String packageName,
            DirectoryKindPPP hiddenDirectory);

    String getPathFileHiddenDirectory(
            String prototypeFileName, String packageName,
            DirectoryKindPPP hiddenDirectory);

    boolean deleteDirOfTestDir(String dirName);

    default String
    nextLocalIdentifier() {
        return NameServer.nextLocalVariableName();
    }
}
```

E.3 Interface ICompilerAction_parsing

```
package meta;

import java.util.List;
```

```
import ast.CompilationUnit;
import ast.ExprAnyLiteral;
import ast.MethodDec;
import ast.ProgramUnit;
import lexer.Symbol;
import saci.CompilationStep;
import saci.Tuple2;

public interface ICompilerAction_parsing
       extends IAbstractCyanCompiler {

    List<List<String>>
    getGenericPrototypeArgListList();

    String getCurrentPrototypeName();
    String getCurrentPrototypeId();
    MethodDec getCurrentMethod();

    List<Tuple2<String, ExprAnyLiteral>>
    getFeatureList();

    CompilationUnit getCompilationUnit();

    char[] getText(int offsetLeftCharSeq,
                   int offsetRightCharSeq);

    ProgramUnit searchPackagePrototype(
        String packageNameInstantiation,
        String prototypeNameInstantiation);

    void errorAtGenericPrototypeInstantiation(
             String errorMessage);
    String getPackageNameInstantiation();

    void setPackageNameInstantiation(
             String packageNameInstantiation);

    String getPrototypeNameInstantiation();
```

```
    void setPrototypeNameInstantiation(
            String prototypeNameInstantiation);

    int getLineNumberInstantiation();
    void setLineNumberInstantiation(
            int lineNumberInstantiation);
    int getColumnNumberInstantiation();
    void setColumnNumberInstantiation(
            int columnNumberInstantiation);
    void error(int lineNumber, String message);
    void error(Symbol sym, String message);
    void error(int lineNumber, int columnNumber,
              String message);
}
```

E.4 Metaobject Class CyanMetaobjectShout

```
package meta.cyanLang;

import meta.AnnotationArgumentsKind;
import meta.AttachedDeclarationKind;
import meta.CyanMetaobjectAtAnnot;
import meta.IAction_semAn;
import meta.ICompiler_semAn;
import meta.WrASTVisitor;
import meta.WrAnnotationAt;
import meta.WrEnv;
import meta.WrExprLiteralString;
import meta.WrMethodDec;

public class CyanMetaobjectShout extends CyanMetaobjectAtAnnot implements

    public CyanMetaobjectShout() {
        super("shout", AnnotationArgumentsKind.ZeroParameters,
                new AttachedDeclarationKind[] {
                        AttachedDeclarationKind.METHOD_DEC });
    }
```

```
    @Override
    public StringBuffer semAn_codeToAdd(ICompiler_semAn compiler_semA
        final WrAnnotationAt annot = this.getAnnotation();
        final WrMethodDec dec = (WrMethodDec ) annot.getDeclaration()

        dec.accept( new WrASTVisitor() {
            @Override
            public void visit(WrExprLiteralString node, WrEnv env) {
                final StringBuffer strUpper = new StringBuffer();
                final StringBuffer str = node.getStringJavaValue();
                for (int i = 0; i < str.length(); ++i) {
                    strUpper.append(Character.toUpperCase(str.charAt(
                }
                replaceStatementByCode(node,
                        strUpper, node.getType(), env);
            }
        }, compiler_semAn.getEnv());

        return null;
    }
}
```

E.5 The Class of Macro assert

```
package meta.cyanLang;

import ast.AnnotationMacroCall;
import ast.Expr;
import ast.ExprIdentStar;
import ast.ExprMessageSendWithKeywordsToExpr;
import ast.MessageBinaryOperator;
import ast.Type;
import error.ErrorKind;
import lexer.Lexer;
import lexer.Token;
import meta.CyanMetaobjectMacro;
import meta.ICompilerMacro_parsing;
import meta.ICompiler_semAn;
import saci.Env;
```

```
/**
 * This class represents macro 'assert' which is used as<br>
 * <code>    assert boolExpr;<code><br>
 * At runtime, if <code>boolExpr</code> is false, an error
    message is issued. The program is NOT terminated.
 *
   @author José
 */
public class CyanMetaobjectMacroAssert
       extends CyanMetaobjectMacro {
s
    public CyanMetaobjectMacroAssert() {
        /*
         * there is only one macro keyword
         */
        super( new String[] { "assert" },
               new String[] { "assert" });
    }

    /**
     * parse the macro call
     */
    @Override
    public void
    parsing_parseMacro(ICompilerMacro_parsing compiler_parsing) {

          /*
           *   compiler_parsing.getSymbol() is the lexical
               symbol for 'assert' from this symbol we
               get its line number and column
           */
        lineNumberStartMacro = compiler_parsing.getSymbol()
            .getLineNumber();
        offsetStartLine = compiler_parsing.getSymbol()
            .getColumnNumber();
          // get past symbol 'assert'
        compiler_parsing.next();
            /*
```

```
            calls the compiler to parse the expression
            that should come after 'assert'. The
            expression is kept in field 'assertExpr'
         */
      assertExpr = compiler_parsing.expr();
         /*
            if there was any errors when parsing the
            expression, returns. Any errors will be
            reported back to the Cyan compiler
         */
      if ( compiler_parsing.getThereWasErrors() )
          return ;
        // does the macro ends with ';' ?
      if ( compiler_parsing.getSymbol().token !=
           Token.SEMICOLON ) {
          compiler_parsing.error(compiler_parsing.getSymbol(),
              "';' expected", null,
              ErrorKind.metaobject_error);
          return ;
      }
      else  // eats the ';'
          compiler_parsing.next();
      return ;
  }

  /**
     generate code for the macro. The string returned
     will replace the macro call
   */
  @Override
  public StringBuffer
  semAn_codeToAdd(ICompiler_semAn compiler_semAn) {

        // the annotation is the macro call, an object
        // of AnnotationMacroCall
      AnnotationMacroCall annotation =
          (AnnotationMacroCall )
          this.getAnnotation();
         // env keeps all the environment of the call:
```

```
    // the current method, prototype, etc
Env env = compiler_semAn.getEnv();
  // if there was any errors before the
  // macro call, return
if ( env.isThereWasError() )
    return null;
  // the  type of the assert expression should
  // be Boolean or Dyn
if ( assertExpr.getType(env) != Type.Boolean &&
     assertExpr.getType(env) != Type.Dyn ) {
    compiler_semAn.error(
        assertExpr.getFirstSymbol(),
        "Expression of type Boolean or Dyn expected");
    return null;
}

// the if statement below is not really necessary.
// It just adds a gentle message send if the
// expression is false
Expr firstExpr = null;
if ( assertExpr instanceof
     ExprMessageSendWithKeywordsToExpr ) {
    if (  ((ExprMessageSendWithKeywordsToExpr)
          assertExpr).getMessage() instanceof
          MessageBinaryOperator ) {
        MessageBinaryOperator mso =
            (MessageBinaryOperator )
            ((ExprMessageSendWithKeywordsToExpr)
              assertExpr)
            .getMessage();
        if ( mso.getkeywordparameterList().get(0)
             .getkeyword().token == Token.EQ ) {
            /*
             * something as
             *     assert s == "a";
             */
            firstExpr =
                ((ExprMessageSendWithKeywordsToExpr)
                      assertExpr).getReceiverExpr();
```

```
                if ( !(firstExpr instanceof
                        ExprIdentStar) ) {
                    firstExpr = null;
                }
            }
        }
    }
      // the generated code is put in string 's'.
      // the line number of the assert statement
      // is recovered from field 'lineNumberStartMacro'
      // Note that method 'asString' of 'assertExpr'
      // is used for getting the expression code.
    StringBuffer s = new StringBuffer();
    if ( offsetStartLine > CyanMetaobjectMacro
            .sizeWhiteSpace )
        offsetStartLine = CyanMetaobjectMacro
            .sizeWhiteSpace;
    String identSpace = CyanMetaobjectMacro.whiteSpace
        .substring(0, offsetStartLine);
    s.append("\n");
    s.append(identSpace + "if !(");
    s.append(assertExpr.asString() + ") {\n");
    s.append(identSpace + identSpace +
      "\"Assert failed in line " +
      lineNumberStartMacro +
      " of prototype '" + annotation
          .getPackageOfAnnotation() +
      "." + annotation.getPrototypeOfAnnotation() +
      "'\" println;\n");
    String str = Lexer.escapeJavaString(assertExpr
        .asString());
    s.append(identSpace + identSpace +
        "\"Assert expression: '" + str +
        "'\" println;\n");
    if ( firstExpr != null ) {
        s.append(identSpace + identSpace + "\"'" +
        firstExpr.asString() + "' = \" print;\n");
        s.append(identSpace + identSpace +
            firstExpr.asString() + " println;\n");
```

```
        }
        s.append(identSpace + identSpace + "};\n");
        return s;
    }

    private Expr assertExpr;
    private int offsetStartLine;
    private int lineNumberStartMacro;
}
```

E.6 Objects Passed as Argument to Grammar Methods

Using the `Player` prototype of Subsection 2.1.3, a DSL code to play video and music can be given as a message passing, as shown below.

```
Player()
    playVideo: "Color demo" duration: 30
    playVideo: "Reef fish" duration: 120
    pause: 10
    playMusic: "Bach BC Allegro"
    stop:;
```

The `grammarMethod` metaobject groups all the arguments of this message into the single object that follows and passes it to method `action:`.

```
[ (
  Union<f1,
    Tuple<String,
         Union<some, Int, none, Any>>,
    f2, String,
    f3, Int,
    f4, Any>() f1: [. "Color demo",
        // the duration is a union with tag 'some:'
       ( Union<some, Int, none, Any>() some: 30 )
       .] ),
  ( Union<f1,
     Tuple<String,
          Union<some, Int, none, Any>>,
          f2, String,
          f3, Int,
          f4, Any>() f1: [. "Reef fish",
```

```
            ( Union<some, Int, none, Any>() none: Any )
            .] ) ,
   ( Union<f1,
      Tuple<String,
         Union<some, Int, none, Any>>,
         f2, String,
         f3, Int,
         f4, Any>() f3: 10 ) ,
   ( Union<f1,
      Tuple<String,
         Union<some, Int, none, Any>>,
         f2, String,
         f3, Int,
         f4, Any>() f2: "Bach BC Allegro" ) ,
   ( Union<f1,
      Tuple<String,
         Union<some, Int, none, Any>>,
      f2, String,
      f3, Int,
      f4, Any>() f4: Any )
]
```

Method `fi:`, `i` a number, is used for creating union objects. The message passing

```
Union<f1, Int, f2, String> f1: 0
```

creates a union object of prototype

```
Union<f1, Int, f2, String>
```

whose contents if an `Int`, `0`, associated with tag `f1`. In the above example, tuples and arrays are given literally, with the syntax `[ ... ]` for arrays and `[. ... .]` for tuples. In some cases, as in the last but one line, prototype `Any` is passed as parameter.

Index