[MP for DSLs]

Metaprogramming for Domain Specific Languages in Cyan

ANONYMOUS AUTHOR(S)

Compile-time metaprogramming (CTMP) has been used for a long time to implement embedded DSLs. However, CTMP in most languages offers a limited set of resources for DSL building. In this paper, we show how metaprogramming in language Cyan supports DSLs. This language has several mechanisms that help to build both embedded and external DSLs with the help of a Metaobject Protocol. There is a tight integration between the language, the compiler, and Domain Specific Languages. That makes it easy to use DSLs for metaprogramming and to integrate their code with Cyan.

Additional Key Words and Phrases: Object-oriented languages, DSL, metaprogramming, compiler, meta-object protocols

1 INTRODUCTION

Metaprogramming is an old concept used with many meanings [Damaševičius and Štuikys 2008]. Any handling of a program by another or by itself is metaprogramming. That encompasses changing the structure of a program at runtime or at compile-time, generating code by tools, observing the program code at runtime, etc. In this paper, we will use mainly metaprogramming made with the help of a compiler and directed by language mechanisms. This will be called compile-time metaprogramming (CTMP).

CTMP allows the introduction of new syntax in the language as done with Lisp-like macros [Seibel 2012]. Then DSL code can be embedded in the language code with no runtime loss in performance than that brought by the DSL code. The program itself or some external code can check the syntax and semantics of the DSL code during the compilation. Errors are discovered earlier. After that, some code generated from the DSL code has to be assimilated into the language code or to be called from it.

Metaprogramming can be made through a Metaobject Protocol (MOP) as in CLOS [Kiczales et al. 1993] (at runtime) or as in OpenC++ [Chiba 1995] (at compile-time). A MOP is a protocol for handling the structure and behavior of an object-oriented program. It may provide a means for changing the language semantics and the structure of classes and objects. For example, one may change how the instance variables of a class are allocated and how method dispatch is made. Classes may be created, methods may be added to classes. A MOP supplies access to and allows the change of features of the language.

This paper shows how the Metaobject Protocol of language Cyan [author as the paper 2018] supports the building of external and embedded DSLs. And how DSLs are used for directing how metaprogramming is made. Cyan is a prototype-based object-oriented language. It supports static typing with optional dynamic typing, an object-oriented exception system, anonymous functions, Java-like interfaces, generic prototypes, and non-nullable types. A prototype is a literal object that defines instance variables (fields) and methods. From a prototype, one can create new objects much like one can create objects from a class.

This paper presents several Cyan mechanisms for using metaprogramming for the building of DSLs and some interesting DSLs that can be built with the help of metaprogramming (MP). Although the language used is Cyan, the mechanisms are general enough to be adapted to other languages. These are cited below.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

(1) Code annotations are used for embedding DSL code inside Cyan code. A code annotation takes the form

```
@name(parms){* Text *}
```

in which `params` are the parameters and `Text` is a DSL code. Each annotation is associated univocally with a metaobject that directs how the DSL is compiled. The metaobject may use the compiler for parsing expressions, statements, and other Cyan elements. Or it could just use the lexical analysis. This greatly simplifies the constructions of small DSLs.
(2) A metaobject has access to all information available to the compiler. Depending on the compilation phase in which it acts, it has access to the object fields (names and types) and methods, visible local variables, and even to the package prototypes.
(3) A DSL can be used for code generation, which is assimilated to the source code, or to checks. Information brought on by the compiler may be used for both things.
(4) Metaobjects can communicate with each other. Data collected from the compiler or from the DSL code in one annotation can be transmitted to another and it can be used for both code generation and checks. Metaobject communication allows a better code modularization.
(5) A Cyan program is specified through a file that describes its component packages. Annotations can be attached to a package and even to the whole program. Then a DSL can be used for adding code or checking all prototypes of a package or program.

This article is organized as follows. Section 2 explains the mechanisms supporting DSLs in the Cyan language and how DSLs are used for metaprogramming. Section 3 compares the DSL support of other languages with that of Cyan. The last section concludes.

2 DSL IN CYAN

The Cyan language offers several mechanisms for metaprogramming, all of them are part of the Metaobject Protocol (MOP). The MOP acts at compile-time only and it allows the creation of new prototypes and the addition of code to prototypes and methods. Through it, one can intercept method calls, instance variable (field) access, method overridden in subprototypes, and inheritance. It is possible to embed DSL code inside Cyan code and do checks beyond those made by the type system. Lisp-like macros [Seibel 2012] are part of the MOP, being one of the ways of implementing DSLs in Cyan.

A metaprogram is a program that can manipulate another program called the base program, which is the regular program. The metaprogram and the program can be made in the same language, as in CLOS [Kiczales et al. 1991]. Or they can employ different languages. Cyan uses Java for metaprogramming because the compiler is made in Java. It is very convenient to use the same language for the compiler and metaprogramming, the interface between the two becomes trivial. However, there is no technical problem that could prevent us from changing that, to make metaprogramming in Cyan. Since the Cyan compiler produces Java source code as output, both languages interact well. Some small changes would allow using the same language for the program and the metaprogram.

Metaprogramming in Cyan will be presented with an example followed by explanations on how it works. Listing 1 shows prototype `Program` of package `main` that declares a method `run` that does not take parameters. Keyword `object` precedes the prototype name. Method declaration is preceded by keyword `func`. Cyan uses a syntax for method declaration and message passing that is somehow similar to Smalltalk. Then a method `multBy5: Int n -> Int` takes an `Int` parameter and returns an `Int`. The parameter type is given before the name and the return value type after "`->`". Local variables and instance variables (fields) can be declared with keyword `var` or `let` (for read-only variables).

Listing 1. Prototype Student that uses metaobject annotations

```
1  package main
2
3  object Program
4      @insertCode{*
5          for num in [ 2, 3, 4, 5 ] {
6              insert: "    func multBy$num: Int n -> Int = n*$num;"
7          }
8      *}
9      func run {
10         assert multBy5: 2 == 10;
11         var sum = 0;
12         @insertCode{*
13             for elem in 2..4 {
14                 insert: "    sum = sum + (multBy$elem: 2);"
15             }
16         *}
17         assert sum == 18;
18     }
19 end
```

This example has two *annotations* `@insertCode`, one inside method `run` and other before this method. An annotation is a link between the metaprogram and the program. It triggers actions from the metaprogram relating to the program such as the addition of code or inspection. For each annotation in the source code, the compiler creates a *metaobject* of a Java *metaobject class* that depends on the annotation name. Method `String getName()` of the metaobject class returns the annotation name associated to the class. The compiled form of a metaobject class is kept in a special directory of a Cyan package. When this package is imported by a Cyan source code, the annotation name associated with it is made available. In this concrete example, the metaobject class whose method `getName()` returns `"insertCode"` is `CyanMetaobjectInsertCode`. File `CyanMetaobjectInsertCode.class`, the compiled form of the class, is kept in a special directory of package `cyan.lang`, a package that is automatically imported by every Cyan source code. That is the reason annotation `@insertCode` can be used in the example without importing explicitly any package.

A metaobject class should be created as a compiler class. After the compiler itself is compiled, the "`.class`" file of the metaobject class should be put in the special directory of a Cyan package (it will work in any package). When importing the package, the annotation will be in scope even if the compiler used was not itself compiled with the metaobject class.

The Cyan compiler has ten main compilation phases. It does three syntactic analysis, two partial semantic analysis, and two full semantic analysis. It is not obviously targeted for efficiency. The first phase is parsing when the Abstract Syntax Tree of the program is built. The second one is "type interfaces" when types are assigned to every entity outside methods. For example, before this phase, an AST object of an instance variable has the information that its type is `"Int"`, a string. After the phase, field "`type`" of the AST object is assigned to the AST object that represents the

prototype "Int". This process occurs with every entity outside a method body. That is, only the statements of methods are not typed.

The third compilation phase is called "ati actions". Metaobjects may ask the compiler to add fields and methods to the prototype in which their annotations are. Metaobjects may also ask to add statements to the beginning of specific methods of the prototype in which they are. The code to be added is given as Java strings. A copy of the Cyan source code, a char array internal to the compiler, is changed in memory. The original files are not touched. The use of strings, instead of AST objects, simplifies enormously code generation. There is no problem with this approach because the compiler does the regular checks after that. You may assume that a metaobject may act only in the prototype in which its annotation is.

Since the source code may have been changed in phase "ati actions" because of the addition of code, it should be compiled again, which is made in the fourth phase, another parsing step. The fifth phase types the interfaces and the sixth phase is semantic analysis made inside method bodies. All statements and expressions inside methods are checked. This sixth phase is called "calculate internal types". In this phase, metaobjects can add source code inside methods just after the annotation. Fields and methods cannot be added by metaobjects in this phase.

Since the source code of the program may have changed in phase 6, it must be parsed again in phase 7, which is followed by phase 8, "type interfaces", phase 9, "calculate internal types", and phase 10, "generate code". After phase 7 the Cyan code cannot be changed. Checks can be made by metaobjects in all phases but 2.

A metaobject may add code just after its annotation in phase 1, parsing. The inserted code is parsed just after the annotation that triggered the insertion and there is no need of parsing the whole source code again. Then code may be inserted in the Cyan program in

(1) phase 1, parsing, but code can be added only after the annotation;
(2) phase 3, ati actions. Fields and methods may be added to the prototype in which the annotations are and statements can be inserted at the beginning of methods;
(3) phase 6, calculate internal types. Code can be inserted only after the annotations.

Annotation `@insertCode` used in Listing 1 takes a DSL code between `{*` and `*}`. Other left and right sequence of symbols could be used in order to avoid conflicts with the DSL grammar, but the right sequence should be a mirror of the left one.

This DSL code is parsed by a method of `CyanMetaobjectInsertCode`, the class associated with the annotation, during the syntactic analysis of the Cyan compiler, phase 1. The AST of the DSL code is built and stored in the metaobject. The first `insertCode` annotation, in lines 4-8, is inside prototype `Program` but outside any methods. Its metaobject produces code in phase 3, ati actions, that is inserted in the prototype (it does not matter where). The code produced is given by the execution of the DSL code. The DSL has only two kinds of statements, `for` (that mimics the Cyan `for` statement) and `insert:`, that uses the syntax of message passing in Cyan, based on Smalltalk. Statement `insert:` gives the code to be inserted as strings. Any "`for`" variable, as `num`, is replaced by its value inside the string if it is preceded by $. The expression to be iterated over can be literal arrays of basic Cyan values (as in line 5), intervals (using `..`, see line 13) of integers and chars, and some selected pre-defined lists, as a list with the names of the parameters of a generic prototype. Four lines of code are produced by this DSL code, the first of which is

```
func multBy2: Int n -> Int = n*2;
```

This method returns the value of the expression after =.

The second annotation `insertCode`, in lines 12-16, is inside method `run` and produces code in phase 6. Its metaobject produces three lines of code, the first of which is

```
sum = sum + (multBy2: 2);
```

The code is inserted after the annotation.

A metaobject class should implement a Java interface of the compiler in order to act in a specific compiler phase. This is the way the compiler knows a metaobject should act in a specific phase and which methods should be called. For example, class `CyanMetaobjectInsertCode` implements interface `IAction_dsa` because it should produce code in phase 6, during semantic analysis (DSA). Method

```
StringBuffer dsa_codeToAdd(ICompiler_dsa compiler_dsa)
```

of the metaobject is called in order to generate code for the annotation of line 12. This method of the metaobject associated with the annotation of line 4 is also called in phase 6. But it must return `null`. It is called because the metaobject could do checks in the code. Code for the annotation of line 4 is generated by another metaobject method in phase 3 of the compilation.

2.1 Metaprogramming Mechanisms to Help Build DSL

The Cyan Metaobject Protocol offers several mechanisms for supporting DSLs at compile-time. One of them is support for DSL code inside Cyan code as in Listing 1. To make that possible, class `CyanMetaobjectInsertCode` implements interface `IParseWithCyanCompiler_dpa` that declares a method `dpa_parse`:

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

Interface `ICompiler_dpa` is a restricted view of the Cyan compiler. It offers methods for parsing such as `next()`, `getSymbol()` (get the current symbol), `expr()` (parse an expression), and `statement()` (parse a statement). This interface should be implemented by the metaobject class when the DSL has some similarities with Cyan. Method `next()` gets the next **Cyan** symbol of the DSL. Method `expr()` assumes that the next token starts a **Cyan** expression. Method `dpa_parse` of `CyanMetaobjectInsertCode` uses only methods `next()`, `getSymbol()`, and `expr()` of its parameter. It builds the AST of the DSL code using its own AST classes, which has references to Cyan AST objects but it is not related to the AST of Cyan. Each of these AST classes define a method to generate Cyan code that is to be inserted in the program. The AST object of the DSL code of an annotation `insertCode` is kept in the metaobject. It is used in the following compilation phases.

The type of the parameter `compiler_dpa` of method `dpa_parse` is `ICompiler_dpa`, which is a restricted view of the compiler. In fact, a copy of the compiler object is passed as a parameter because the compiler class implements `ICompiler_dpa`. Then any changes to fields of this compiler copy are not transmitted to the Cyan compiler.

However, the statements and expressions parsed with this copy are put in a list and stored in the metaobject annotation. During phase 6 of the compilation, the compiler does, automatically, semantic analysis of this list of expressions and statements. This analysis is based on types available to the whole program. The same compiler environment object used to do checks in the program is used for checking the DSL code. This environment object has all type information of the program and it is not changed by any DSL code, which may only read information from it.

The expressions used in the DSL code are associated with types. In this example, this is not really necessary because literals are used and their types can be deduced easily. This semantic analysis of DSL code is appropriate only when the code is somehow related to language Cyan. If it is not, the `dpa_parse` method may ask that semantic analysis not be done. Method `next()` of the parameter `compiler_dpa` of `dpa_parse` does not allow the metaobject to get symbols past the DSL code, after "`*}`".

A metaobject class may implement interface `IParseWithoutCyanCompiler_dpa` for parsing the DSL code without any help from the Cyan compiler. It declares a method

```
void dpa_parse(ICompilerAction_dpa compiler, String code)
```

that takes a restricted view of the compiler, parameter `compiler`, and the code as a string, parameter `code`. The latter contains the text between the delimiters in the annotation (`{*` and `*}` in the examples). Interface `ICompilerAction_dpa` declares several methods such as `getCurrentPrototype()`, `getCompilationStep()`, and `getProgramVariable(String)` (a constant associated to the program, it may be a compiler option, for example). These same methods are available through the parameter to `dpa_parse`, `ICompiler_dpa` inherits from `ICompilerAction_dpa`. A metaobject class that implements `IParseWithoutCyanCompiler_dpa` can use a external tool to help compile the DSL code or it can implement all the compiler itself.

Metaobjects can communicate with each other before the compilation phases 3, 6, and 9. If a metaobject needs to pass or receive information from metaobjects associated to annotations of the prototype, its class should implement interface `ICommunicateInPrototype_ati_dsa`. This interface declares a method for sharing data with a parameter of kind `Object`, the top-level Java class. And a method for receiving data shared by other metaobjects of the same source code. If a metaobject class implements this interface, the Cyan compiler will first call the method to share information for all metaobjects associated to annotations of a prototype and collect the objects into a list. Then this list is passed as a parameter to a call to the method that receives information from every metaobject associated to an annotation of the prototype. In general, a metaobject will be interested only in elements of the list that are of a certain type. It can test the type of each element using the Java operator `instanceof`. The sharing of information is made before compilation phases 3, 6, and 9. Metaobject communication allows a better code modularization because metaobjects of annotations that are in different code locations can communicate with each other. More than that: different kinds of annotation can communicate with each other. As will be seen, there are seven different annotation kinds.

Metaobject classes that want to create new prototypes should implement interface `IActionNewPrototypes_dpa`, `IActionNewPrototypes_ati`, or `IAction_dsa`. Or all of them. Then new prototypes can be introduced during parsing, after typing interfaces, or during semantic analysis.

2.2 An Unconventional Kind of Annotation

A program may have, besides files with "`.cyan`" extensions, files with extensions that are DSL names. For example, "`math.fyan`" for a DSL "`fyan`". Each of such files should contain code of a single DSL and it should be in a special directory of a Cyan package. The DSL code is transformed into one or more Cyan prototypes. In order to explain that, it is first necessary to describe *project files*.

A Cyan program is specified by a DSL code that should be in a *project file* with extension "`pyan`". The DSL is called Pyan (for Project cYAN).

```
program
    package main
    package bank
```

Keyword `program` is followed by one or more `package` keywords with the package names. Optionally the directories of the packages can be given after "`at`". Otherwise, it is assumed that the directories are the same as the directory of the project file. Annotations can be used in Pyan files:

```
import cyanHelper at "C:\Dropbox\Cyan\lib\cyan\cyanHelper"
```

```
@checkStyle
program
    package main
    @immutable
    package bank
```

Metaobject `checkStyle` checks several style elements such as method and variable names. Since the annotation is attached to `program`, the whole program is checked. Annotations to a declaration (program, package, prototype, method) can visit all AST nodes of that declaration using the *Visitor* pattern. They can add code and do checks too. Annotation `immutable` is attached to package `bank`, which means all prototypes of the package should be annotated with `@immutable`. This metaobject, when attached to a prototype, assures its fields cannot be changed after initialization. Metaobjects of package `cyan.lang`, as `checkStyle` and `immutable`, are automatically imported.

One special kind of Cyan metaobject has an unusual annotation: it is a file that is in a directory `--dsl` of a package.[1] During parsing, the compiler looks for a metaobject called "`ext`" for every file extension with this name that appears in a file of a `--dsl` directory. The metaobject class should inherits the Java class `CyanMetaobjectFromDSL_toPrototype` and override method `dpa_NewPrototype`. This method should return a list of prototypes to be created by the compiler. Each element of the list is composed by the prototype name, its file name, and its source code. If the `--dsl` file (with the DSL) starts with an uppercase letter, only one prototype should be returned. If it starts with a lowercase letter, any number of prototypes can be on the list.

As method `dpa_parse` of interface `IParseWithCyanCompiler`, method `dpa_NewPrototype` takes a restricted view of a copy of the Cyan compiler as parameter. It can be used in the parser of the DSL. This method does not need to use this parameter. It has access to the text of the file and therefore the DSL compiler can be implemented from scratch or with the help of an external tool.

The last example of a *pyan* file uses keyword "`import`" to import metaobjects of package `cyanHelper` to the project file. Only metaobjects are imported and these can be used in the `pyan` file (the project) and in the `--dsl` directories of the packages of the project. Then a package may have a DSL file in its `--dsl` directory with an extension associated to `cyanHelper`.

2.3 DSLs for Metaprogramming

Listing 1 is an example of how DSLs can be used for metaprogramming. The DSL code attached to an `insertCode` annotation produces code that is inserted into the program, a typical role of metaprogramming. The code is an embedded DSL that is not regular Cyan code. Its syntax was made similar to Cyan on purpose but it could be anyone.

Metaobject `insertCode` is not attached to a declaration (prototype, method, etc.). It adds code in phase 3, ati actions, if the annotation is outside a method, and in phase 6 if the annotation is inside a method. Metaobjects can be used in other compilation phases and not only for adding code. They can do just checks, for example. We will show more examples of metaobjects used for metaprogramming.

Listing 2 shows annotation `grammarMethod` attached to method `action`. The DSL code of the annotation is a regular expression using method keywords (as `move:`), types, and operators `+` (one or more), `*` (zero or more), `|` (alternative), and `?` (optional). The metaobject associated to this annotation intercepts, at compile-time, message passings for which there is no adequate method. For example,

```
Robot move: 10 on: red: left: 20 move: 20 stop:;
```

[1] To please Unix users, the `--` will be changed to `++`.

Listing 2. Example of grammar method use

```
package main

object Robot
    @grammarMethod{*
        ( move: Int |  (on: (red: | green: | blue: )) |
          left: Int | right: Int | stop:  )+
    *}
    func action: Array<
            Union<f1, Int, f2,
                Tuple<Any,
                    Union<f1, Any, f2, Any, f3, Any>>,
                f3, Int, f4, Int, f5, Any>>
            params {
        // code that moves the Robot
    }
    // ... elided
end
```

should result in a compilation error. When the compiler fails in finding method

```
move: Int on: red: left: Int move: Int stop:
```

it asks the metaobject associated to `grammarMethod` if it can handle it. This metaobject tries to match the method description with the regular expression that is the DSL code of the annotation. If it does, the arguments of the message are packed in a combination of unions and literals of arrays and tuples. The packing will not be explained in details. Repetitions in the regular expression are mapped to arrays, sequences of keywords to tuples, and alternatives to unions. The final result is that the metaobject asks the compiler for replacing the above message passing by a call to method `action:` with a single parameter that encapsulates all the arguments of the original message.

Although `grammarMethod` employs a regular expression to define a grammar, another metaobject could use a more powerful grammar. The only limitation is that the tokens should be Cyan keywords and expressions.

The DSL of `grammarMethod` may take a name after the regular expression:

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

`checkPrintf` is a metaobject for checks only, it is not associated with an annotation. A method of it is called with data on the message passing. It then checks if the literal format string given as the first argument matches the remaining arguments. If not, an error is issued at compile-time.

The Java metaobject class of `grammarMethod` implements several interfaces. One is

```
IParseWithCyanCompiler_dpa
```

for parsing the regular expression and

```
ICompileTimeDoesNotUnderstand_dsa
```

Listing 3. Example of grammar method use

```
package main

@concept(test){*
    T has [ func * (T other) -> T
            func concat: T other -> String ],
        "T should have methods '* T -> T' and 'concat: String'",

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

        if a * b != b * a ||
           c * b != b * c  {
            ^"T is not commutative"
        }
        if a concat: b != a ++ b || b concat: a != b ++ a {
            ^"Method 'concat:' of T should " ++
              "concatenate the elements as Strings"
        }
        ^Nil
    %}
*}
object GP<T>
    // elided
end
```

for intercepting failures in finding methods. Metaobject `grammarMethod` uses DSL for selecting the message passings that should be intercepted, a typical use of Metaobject Protocols such as that of CLOS [Kiczales et al. 1991].

Concepts [Gregor et al. 2006] are restrictions on parameters to generic classes (C++ templates). This feature originated in C++ to allow early error detection although unfortunately it was not incorporated in the language. The use of a template class, for example, with a real argument is called instantiation. It causes the creation of a new class with the formal class parameter replaced by the real argument, which we will consider to be another class. The instantiated class may have compilation errors because the real argument does not obey some restrictions demanded by the template class code. It may not have a certain method, for example. This would result in a compilation error usually difficult to understand because it is related to code inside a template class. Concepts are a mechanism to signal errors early, at the instantiation site. The template class specifies the restrictions that the real type argument should obey. It should support a given method, for example. Then the compiler checks that the real argument, a class, comply with the restrictions at the instantiation site. The error message is clearer.

Concepts are implemented in Cyan using the metaobject `concept` that also demands an attached DSL code, as shown in Listing 3. This DSL has statements for testing restrictions on the parameter of a generic prototype. In this example, `T` is the generic prototype parameter of prototype `GP`. When used with a real type, as in `GP<Proto>`, the metaobject demands that `Proto` obeys the restrictions given by the DSL code. Then `Proto` should have a method `*` and a method `concat:` with the given

Listing 4. Concept reuse

```
1  @concept{*
2      cyan.lang.arithmetic(R)
3  *}
4  object MatrixElement<R> ... end
```

parameters and return value. The DSL keyword `axiom` is used for generating a test case that is written to a file whose name is composed of the generic prototype name and real parameter name. This file is written to a subdirectory of the test directory of the package in which the instantiation occurred. Every package may have a `--test` directory to which metaobjects may add test cases. The DSL of `concept` has many more features: it may check that a parameter is an interface, that it implements an interface, that is subprototype or superprototype of another prototype, that it is in a list of prototypes, etc.

A bunch of type restrictions of a concept can be grouped in a file and reused. The file name should be used as a function that takes type parameters between parentheses, as shown in Listing 4. There is a file `arithmetic(T).concept` in a subdirectory `--data` of the directory of package "`cyan.lang`". The concepts of this file demand that the argument, a type, had all the arithmetic operators. There are also axioms in the file that will be used to create test cases for any instantiation of this generic prototype.

3 COMPARISON TO OTHER LANGUAGES AND SYSTEMS

In relation to the topic of this article, the closest language to Cyan is Converge [Tratt 2008], which permits the embedding of *DSL blocks* using the syntax

```
$<<dslCompiler>>
    code
```

`dslCompiler` is a *DSL implementation function* called at compile-time with the argument "`code`" (as a string). "`code`" is the DSL code and should be indented (Converge is Python-based). The Converge compiler supplies a function that can automatically generate a parse tree from an input and a grammar. The grammar can use a rule that represents a Converge expression. Then expressions of the language can be used inside DSLs. In Cyan this is allowed not only with expressions but with more rules such as statements, types, literals, etc. Methods for that are declared in interface `ICompiler_dpa` described in Subsection 2.1, which also declares methods for getting the current symbol and advancing to the next. In Cyan, semantic analysis is made with all expressions and statements parsed by objects of `ICompiler_dpa`, unless there is an explicit request to not do so. However, Cyan does not offer any tools to help generate a parse tree from a grammar. An external tool can be used, if necessary.

Metaobjects in Cyan are associated with annotations and they can access information available to the compiler such as the list of prototype fields and methods, the visible local variables, etc. This data is also available in languages that have a good support for metaprogramming such as Nemerle [Skalski 2005], Xtend [Xtend 2017], and Groovy [Koenig et al. 2007] [Groovy 2017]. In these languages, DSLs can be given inside strings as annotation parameters, a feature that is underutilized. Cyan annotations in the project file of a Cyan program, the ".pyan" file of Subsection 2.2, can analyze and even change the whole program. This can be achieved with Groovy *global AST transformations*. These transformations are not made with an annotation. The whole program AST is visited by a method that can do checks and code changes.

The Java-based language SugarJ [Erdweg et al. 2011] enables DSL embedding through the use of SDF [Heering et al. 1989] to specify syntax and Stratego [Visser 2001] to transform embedded DSL code and other syntax extensions into Java. Like Cyan, SugarJ allows syntactic extensions by the importing of libraries. However, the goals of both languages are different. SugarJ does not have a Metaobject Protocol as Cyan and it allows non-trivial new syntax to be added to Java. SugarJ is even classified as a language workbench by Erdweg et al. [Erdweg et al. 2015].

Polyglot [Nystrom et al. 2003] and JastAddJ [Ekman and Hedin 2007] are extensible Java compiler frameworks that can be used to extend Java with new language constructs. Then both can be used for DSL building and are examples of *open compilers*. The Cyan compiler is not open, the MOP can *add* new checks and may be used for compiling embedded DSL code but no new grammar rules can be added to the language. Behavior can be added to the compiler, it cannot be replaced.

Many languages and tools that support metaprogramming employ DSLs for specifying code generation at compile or runtime. The quote and unquote mechanism of many languages as Lisp [Seibel 2012], Converge [Tratt 2005], Scala [Odersky and Others 2004], and Nemerle [Skalski 2005] can be considered a small DSLs. These mechanisms have a syntax and a semantics and they apply a transformation to the code producing an AST or another form of code representation. An example of that is macros in language Rust [Klabnik and Nichols 2018]. A macro is defined using a macro name and a set of patterns, each with the code that should be generated if that pattern matches the code that follows the name.

```
1 macro_rules! vec {
2     ( $( $x:expr ),* ) => {
3         { let mut temp_vec = Vec::new();
4             $( temp_vec.push($x); )*
5             temp_vec
6         }
7     };
8 }
```

In this example, taken from [Klabnik and Nichols 2018], there is only one pattern in line 2 with its associated code in lines 3-6. Then macros in Rust are defined using a DSL. And so are macros in Nemerle [Nemerle 2017], in which a single production can be added to the grammar. The macro feature of Rust and, to a less extent, that of Nemerle, is related to metaobject `grammarMethod` of Cyan. Both specify a pattern to be matched and both replace a code for another at compile-time.

Language workbenches (LW) are tools [Erdweg et al. 2015] to define DSLs and supporting tools for them such as compilers, IDEs, and even debuggers. As an example, Spoolfax [Kats and Visser 2010] is a LW that employs several meta-languages, also DSLs, such as SDF [Heering et al. 1989] for grammar rules and Stratego [Visser 2001] for transformations, including code generation. A DSL may be specified through a series of rules and, from them, a whole compiler can be built, including an IDE plugin. The tools for building DSLs in a language workbench are overwhelmingly better than in Cyan, which only offers basic support. However, an LW depends on tools, configuration files, and metadata outside the language and even the host language compiler. In Cyan the DSLs are integrated with the language Metaobject Protocol and can use all that the MOP offers.

4 CONCLUSIONS

DSL building may be supported by languages (Converge, Groovy, Nemerle, SugarJ, etc.), open compilers (Polyglot, JastAddJ), and language workbenches (any of them, by definition). Each solution has a different set of offsets. Cyan supports DSLs through metaprogramming with its

own advantages and drawbacks. On the cons side, no new grammar rule may be added to the language and there are no sophisticated tools to help build DSLs. On the pro side, the Cyan compiler offers parsing and semantic analysis of expressions and statements, access to compiler information, and metaobject communication. DSLs are imported as libraries and DSL files coexist in package directories with Cyan files. There is a tight integration between DSLs, the language, and compiler. This allows the use of DSLs for metaprogramming, as `insertCode` and `concept`. Although this paper describes the features of one language, the ideas presented here can be applied to other languages and Metaobject Protocols.

Cyan offers many syntax alternatives for DSLs, probably more than any other single language or system. A DSL may be expressed using

(a) "`@`" annotations, as in the article examples;
(b) a string preceded by a metaobject name, as in `xml" XML code "`. Multi-line strings, delimited by `"""`, can be used;
(c) a literal object delimited by user-defined characters, as `[# XML code #]`, in which `[#` and `#]` delimit the DSL code;
(d) a separate file that is kept in a special directory of a package.

There are annotations that are not covered in this paper: macros, a number ending with a metaobject name as `01010bin`, and "`@`" annotations associated with graphical user interfaces (GUI), called Codegs. This last annotation demands an IDE plugin, which has been implemented. During editing time in the IDE, when the user hovers the mouse on an annotation that is a Codeg, a GUI specified in the metaobject opens up. The data collected is passed on to the compiler as if it was DSL code. It can then be used for code generation and checks, as DSL code. It is important to notice that metaobjects of all annotation kinds have essentially the same power as "`@`" annotations.

DSL code in an annotation may be used:

(a) as an expression. For example, `@rpn{* 2 3 + *}` may be used anywhere an `Int` expression is expected;
(b) in the project file (`.pyan`). Currently no metaobject uses this feature. Metaobject `checkStyle`, cited in Subsection 2.2, will be changed to take the style from a DSL code;
(c) in an annotation attached to a type, as in

```
Int@restrictTo{* self >= 0 && self <= 10 *}
```

Variables of this annotated type must hold values that satisfies the expression. We are unaware of any pluggable type system [Papi et al. 2008] that accepts DSL code in annotations.

The Cyan compiler is fully functional and supports all of the features cited in this text. The compiler and documentation can be found in `www.cyan-lang.org`. This site has an "articles" page with all the code of this paper.

REFERENCES

Same author as the paper. 2018. The Cyan Language. http://www.cyan-lang.org/

Shigeru Chiba. 1995. A Metaobject Protocol for C++. In *Proceedings of the Tenth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '95)*. ACM, New York, NY, USA, 285–299. https://doi.org/10.1145/217838.217868

Robertas Damaševičius and Vytautas Štuikys. 2008. Taxonomy of the Fundamental Concepts of Metaprogramming. In *Information Technology and Control*. Vol. 37. Kaunas University of Technology.

Torbjörn Ekman and Görel Hedin. 2007. The Jastadd Extensible Java Compiler. *SIGPLAN Not.* 42, 10 (Oct. 2007), 1–18. https://doi.org/10.1145/1297105.1297029

Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus Ostermann. 2011. SugarJ: Library-based Syntactic Language Extensibility. *SIGPLAN Not.* 46, 10 (Oct. 2011), 391–406. https://doi.org/10.1145/2076021.2048099

Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen

Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and Comparing Language Workbenches. *Comput. Lang. Syst. Struct.* 44, PA (Dec. 2015), 24–47. https://doi.org/10.1016/j.cl.2015.08.007

Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. 2006. Concepts: Linguistic Support for Generic Programming in C++. *SIGPLAN Not.* 41, 10 (Oct. 2006), 291–310. https://doi.org/10.1145/1167515.1167499

Groovy 2017. Runtime and compile-time metaprogramming. http://groovy-lang.org/metaprogramming.html http://groovy-lang.org/metaprogramming.html.

J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. 1989. The Syntax Definition Formalism SDF&Mdash;Reference Manual&Mdash;. *SIGPLAN Not.* 24, 11 (Nov. 1989), 43–75. https://doi.org/10.1145/71605.71607

Lennart C.L. Kats and Eelco Visser. 2010. The Spoofax Language Workbench. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (OOPSLA '10)*. ACM, New York, NY, USA, 237–238. https://doi.org/10.1145/1869542.1869592

Gregor Kiczales, J.Michael Ashley, Luis Rodriguez, Amin Vahdat, and Daniel G. Bobrow. 1993. *Metaobject protocols: Why we want them and what else they can do.* MIT Press, Cambridge, MA, USA. 101–118 pages.

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

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language* (second ed.). No Starch Press. https://doc.rust-lang.org/book/second-edition

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

Nemerle 2017. The Nemerle Programming Language. http://nemerle.org/ http://nemerle.org/.

Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. 2003. Polyglot: An Extensible Compiler Framework for Java. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*. Springer-Verlag, Berlin, Heidelberg, 138–152. http://dl.acm.org/citation.cfm?id=1765931.1765947

Martin Odersky and Others. 2004. The Scala Language Specification. (2004). Available at http://www.scala-lang.org.

Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. 2008. Practical Pluggable Types for Java. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*. ACM, New York, NY, USA, 201–212. https://doi.org/10.1145/1390630.1390656

Peter Seibel. 2012. *Practical Common Lisp* (1st ed.). Apress, Berkely, CA, USA.

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

Laurence Tratt. 2005. Compile-time Meta-programming in a Dynamically Typed OO Language. In *Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05)*. ACM, New York, NY, USA, 49–63. https://doi.org/10.1145/1146841.1146846

Laurence Tratt. 2008. Domain Specific Language Implementation via Compile-time Meta-programming. *ACM Trans. Program. Lang. Syst.* 30, 6, Article 31 (Oct. 2008), 40 pages. https://doi.org/10.1145/1391956.1391958

Eelco Visser. 2001. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *Proceedings of the 12th International Conference on Rewriting Techniques and Applications (RTA '01)*. Springer-Verlag, Berlin, Heidelberg, 357–362. http://dl-acm-org.ez31.periodicos.capes.gov.br/citation.cfm?id=647200.718711

Xtend 2017. Xtend — Modernized Java. https://www.eclipse.org/xtend/ https://www.eclipse.org/xtend/.