Classroom Object-Oriented Language

Group COOL

Andreas Haglund, Kenneth K. Aastveit

https://github.com/uiano/COOL-MPS/tree/ikt445-2021

IKT445

19.12.2021

Abstract. COOL is a simple programming language which is mainly used for learning about compilers and language design. In this project, a COOL implementation in MPS is evaluated, documented and improved. A descriptive summary of the language is given, parts of the implementation is described and the improvements are documented along with suggestions for future improvements. Finally, our contribution to the project is evaluted, and some suggestions for making the project easier for future students are given.

1 Introduction

For the IKT445 Generative Programming course we were tasked with improving upon an existing project for Jetbrains' Meta Programming System (MPS), the Classroom Object-Oriented Language—COOL hereafter. Based upon existing work, our goal was to gain an understanding of the various modules that had already been implemented in MPS, such as the structure, editor, typesystem, and so on—and find areas of the language that were either missing or could be built upon further, such that we could implement said features.

Thus, the primary focus of this report is to reinforce and document the implementations and ideas of past students, as we ourselves had difficulties understanding the work that had gone into the project previously. As a result of this, we hope future students who may look at this report will gain a better understanding of both the COOL language and the MPS IDE, and therefore be able to improve upon the project more easily.

2 Language

COOL is a small language designed for compiler courses that consists of many features seen typically in modern programming languages, including objects, inheritance, static typing, and automatic memory management. Furthermore, COOL is an expression language, meaning most constructs are expressions—for example, constants such as Bool, Int, and String, conditionals, loops, blocks, and so on. In addition, COOL is type safe—procedures are guaranteed to be applied to the data of the correct type, and as such guarantees no runtime type errors. [1].

In this chapter we will give a quick overview of the COOL programming language. For a more detailed explanation we refer to The Cool Reference Manual by Alex Aiken [1].

2.1 Description

2.1.1 Classes

All code in COOL is organized into classes. Thus every COOL program must have a Main class serving as the entry point, or in other words, as the start point of execution. Furthermore, each class definition must have a type—and every class is also its own type—and can optionally inherit a class of another type. The feature list represents attributes or methods part of said class. With that said, COOL includes a few basic classes providing the language with common methods typically seen in most languages. For instance, String provides methods to check the length of strings or concatenate two strings, IO provides methods to perform simple input and output operations, and so on. Moreover, classes are constrained in that they cannot contain the keywords of the language such as new, true, false, and so on—nor can classes have the same name. In addition, classes must start with an uppercase letter followed only by the letters A to Z, numbers, or underscore. And just as every COOL program must have a Main class, so must the Main class include a method Main [1]. In the editor, a classic Hello World would look like either of the following examples:

```
1 class Main inherits IO {
2     main( ) : SELF_TYPE {
3         {
4             out_string( "Hello World!" )
5         }
6     }
7 }
```

```
1 class Main inherits IO {
2     main( ) : SELF_TYPE {
3         out_string( "Hello World!" )
4     }
5 }
```

In the above examples, we show two simple ways to write your typical Hello World—we

have two Main classes inheriting IO in order to access the out_string method which allows us to print values to the console. The main method is of type SELF_TYPE, which we will come back to. One detail is important to mention—by default, COOL methods may only have one expression child. Therefore, should you want to include several expressions in a method, a block scope expression is required containing an optional amount of expression children [1].

2.1.2 Feature List

As mentioned in 2.1.1, the body of a class definition consists of a list of feature definitions. There are two types of features: attributes and methods. For instance, an attribute of class A specifies a variable that is part of the state of an object of class A. Furthermore, a method of class A is a procedure that may manipulate the variables and objects of class A. In COOL, all attributes have private access modifier by default. The code example below shows a class containing one attribute, namely text, and one method, `out_a()`. The method manipulates the text attribute, and prints it to the console [1].

```
1  class A inherits IO {
2      text : String <- "This is a text.";
3
4      out_a( ) : Object {
5          {
6              text <- text.concat(" And this is more text.")
7              out_string(text)
8          }
9      };
10 }
```

Cool supports information hiding through a simple mechanism: all attributes have scope local to the class, and all methods have global scope. Thus, the only way to provide access to object state in COOL is through methods.

In terms of restrictions, they are largely the same as Classes. However, a method and an attribute may have the same name.

On the images to the right-hand side we see that both Method and Attribute implements IFeature as expressed earlier. A method has one expression child, but may optionally include any amount of parameters. An attribute consists of an optional expression, such as the assignment of a value to an attribute as shown in the previous slides example A. Due to type safety, both features must have a return type [1].

2.1.3 Types and Type Checking

In COOL, as mentioned earlier, every class name is also a type and every variable and attribute must have a type declaration at the point it is introduced. In addition, there is a type named SELF_TYPE that can be used in special circumstances. This type is special in that it returns the type of the "self" attribute in a class. For instance, given a class Dog, SELF_TYPE will refer to Dog. The basic type rule in COOL is that if a method or variable expects a value of type A, then any value of type B may be used instead, if A is an ancestor

of B in the class hierarchy. In other words, if B inherits from A, then B can be used wherever A would suffice.

Furthermore, the COOL type system guarantees at compile time that execution of a program cannot result in runtime type errors [1].

2.1.4 Expressions

Expressions are the largest syntactic category in COOL, and consist of various common expressions you would see in other modern languages. This includes, for instance, constants such as true/false, integers and strings. Part of expressions are also concepts such as conditionals, loops, arithmetic operations, and so on. Thus, almost all concepts in the COOL language inherit from the expression interface named IExpression, providing a common interface used to refer to all other expressions [1].

3 Solution

The COOL Reference Manual has been used to check the existing implementation of COOL. The language is fully implemented with all the expressions, features, basic classes and lexical structure. The missing parts are mainly related to the editor and type checking.

3.1 Classes in MPS

A class is the only concept which can be a root node. It has two children — features and inherits. While a class may have any number of features, it can only optionally inherit one class of concept ClassRef, specifically referencing only one class. It extends the INamedConcept, giving it a name property. As already mentioned in 2.1.1, a class name has certain constraints. The name cannot be any of the COOL keywords. The keywords can be found in The COOL Reference Manual. Additionally the name mus be unique, meaning no other classes can have the same name, and the name must start with an uppercase letter, and can contain only letters, numbers and underscore. These constraints are implemented in the Class_Constraints concept.

3.2 Features

Features in COOL are a commone interface for two concepts: attributes and methods. The IFeature interface in MPS is the feature interface which both Attribute and Method implements.

The Attribute concept consists of zero or one expression, and it has only one type, which is a Class concept. It implements IAttributeDeclaration, which again implements the new interface ICoolName. IAttributeDeclaration contains the specific constraints for the attribute names, while the new ICoolName interface contains the rules to avoid keyword names.

The Method concept is pretty similar to Attribute in many ways. It implements IMethodDeclaration, which contains the same naming constraints as IAttributeDeclaration. The reason they have to be in separate files is because an attriubte can have the same name as a method, but two attributes or two methods can not have the same name. IMethodDeclaration also implements ICoolName. The Method concept takes one expression and zero or more parameters. In order to add more than one expression to a method one have to wrap them inside of a block expression.

3.3 Expressions

As stated in 2, almost every construct in COOL is an expression. In MPS this is solved by letting all expression concepts implement IExpression. We will go into details about two of the expressions we think might be useful for future students, namely the Block and the Dispatch expressions.

3.3.1 Block

The block expression, which is already mentioned in 3.2, is an expression which basically just holds other expressions. The example used in 3.2 is when block is used in a method. As a method hold only one expression, a block expression can be used to add more than one expression to the method. One feature added regarding block expression was a Surround-With Intention added to IExpression. This lets the user of the language select an expression and surround it with a block expression. This makes it a lot easier to edit existing methods which only contains one expression. Instead of removing the expression, adding a block, and then inserting it again it can now be surrounded with a block using the shortcut Cmd+Opt+T/Ctrl+Alt+T. Note that this intention only applies if the expression is not already within a block.

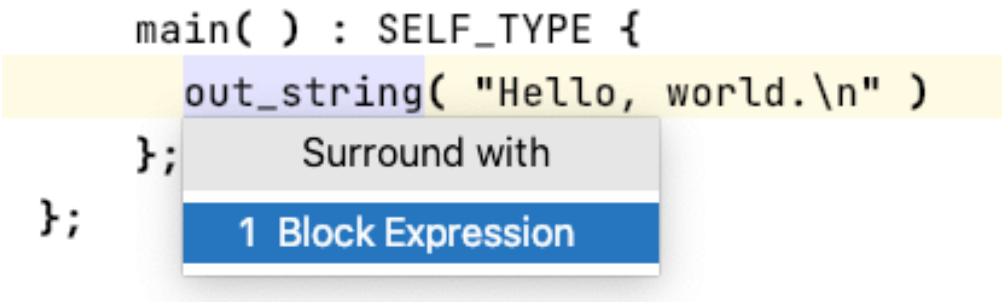

Figure 3.1: A Surround-With Intention on IExpression.

3.3.2 Dispatch

The Dispatch expression represents a method call. In the editor this can prove an annoyance if this is not known to the user. In order to call a method, one has to choose the Dispatch option from the code completion menu. Furthermore, the Dispatch concept has two children: parameters and an expression, both which are of type IExpression. It can take 0 to n parameters, but only zero or one expression. The expression can be put in front of the Dispatch with a dot between, to access methods of the type of that expression. In the editor, in order to add parameters to a Dispatch, hit the comma key.

3.4 Contributions and Future Work

Our most noteworthy contribution to the COOL project is extended documentation through the use of several sandbox models, and this report itself. Our aim was to create code samples of each COOL expression—a goal we fulfilled. These code samples include,

ranging from limited to extensive comments, detailing how to work with the MPS editor in the context of both COOL and the program itself. By doing this, we hope future students who may look at the project will gain a far better understanding of the project than we did. Furthermore, among our contributions, are:

- The ICoolName concept was added, such that keyword constraints can be added to this concept instead of individual concepts.
- Equality of strings in Java use the `equals()` method instead of `==`. This was previously not captured by the generator.
- Syntax highlighting based on the IDE color scheme settings.
- Surround-With Intention to surround an expression with a block expression is added.

Due to the fact that a lot of our time was spent learning the IDE and the editor for COOL, we gained significant insight into the problems that currently exists, and possible solutions that would provide quality of life improvements. As a result, here is a list of areas we deem important for future work:

- The name of the Main class should use a checking rule instead of a constraint on the name itself.
- Methods currently use the comma key as an interaction method in order to write parameters. This is a side transformation that causes notable confusion.
- The reduction rule for the Let concept does not work properly—although the code is able to generate, expressions such as a while loop is unable to use the name of the attribute in the generated code, thus leaving an empty statement that cannot be run.
- The code completion menu could be improved based on what is currently needed. Currently, it displays everything possible within the scope of a class or a method, and thus lacks further context. For example, always showing every arithmetic and comparison operation is a little overkill.
- Some side transformations, such as the one for assignment (`<-`), cause the identifier to vanish, requiring you to type it over again. This, including the fact that the assignment operator sometimes overlaps other nodes, can be annoying.
- Comments could be improved upon such that you could have intentions to comment out one line in particular or several lines at once.

4 Discussion

Working on COOL in MPS has been challenging. We were not able to fully understand MPS, and several failing attempts and frustrating trials stole both time and motivation. We will explain three main reasons why we think this this task was challenging, and come up with ideas to improve on these factors.

First of all, most people introduced to MPS are already familiar with other products from Jetbrains, like PyCharm and IntelliJ, just to mention some. MPS has many similarities, but at the same time there are so many dissimilarities, making it really confusing and frustrating using the MPS editor. The structure of the models, and the fact that models are not files makes it hard to navigate and search through the project. This also applies to version control. The fact that the actual saved files look so different from the implemented code makes it really hard to track changes in GitHub. A better introduction to MPS, and a comparison between MPS and other Jetbrains editors might be helpful for future students.

Secondly, there are limited sources on MPS available, and almost everything available is content made by the same person. Normally when learning a new programming language or a new tool there are plenty or sources from different people, explaining it differently. Some kind of documentation, like javadoc, would have been of big help. However, for future students we would recommend the tutorials at stepik.org. Tying the course content more together with MPS during the course could also have been really helpful. The introduction to MPS at the beginning of the course did not help much, as we had no knowledge of what pieces a programming language actually exists of at that time.

The third factor is the COOL implementation itself. The COOL implementation is large and confusing. The project already contained several errors and the error messages was very little descriptive. It felt like being new to running, starting the career with a marathon. Starting a project from scratch, or having a simpler language to work on could have made the experience better.

All in all it comes down to one big ingredient in learning, which was more or less absent for us in this project: motivation. Even though other courses took much time away from the project, there were plenty of time left to work on COOL, but hours of frustration, no results and not enough ressources made the task too complicated. However, we have done our best effort. We have learnt many things about MPS and language construction which will come handy in the future.

5 Plan and Reflection

We did not work on this project as much as we would have liked to, mainly due to the fact that we only got a one day introduction to MPS. As a consequence of this, we had significant trouble understanding the already existing implementation of COOL that we inherited. Furthermore, due to both the project itself and MPS's limited documentation, figuring out where to start was a heavy undertaking. Thus, other courses saw higher precedence, but we nonetheless did our best toward learning the COOL language itself and divided what work we could based on an existing to-do list of the MPS implementation.

Bibliography

[1] Alex Aiken. *The COOL Reference Manual*. Accessed: 10.12.2021. URL: https://theory.stanford.edu/~aiken/software/cool/cool-manual.pdf.