COOLer: A Language Support Extension for COOL in VS Code

Linhan Li and ThanhVu Nguyen
George Mason University
USA

Abstract

COOL is an object-oriented programming language used to teach compiler design in many undergraduate and graduate courses. Because most students are unfamiliar with the language, and because code editors and IDEs often lack support for COOL, writing code and test programs in COOL is burdensome to students, causing them to not fully understand many important and advanced features of the language and its compiler.

In this paper, we describe COOLer, an extension providing support for COOL in the popular VS Code IDE. COOLer offers (i) syntax highlighting support for the COOL language through lexing and parsing, (ii) semantic-aware auto-completion features that help students type less and reduce the burden of remembering unfamiliar COOL grammar and syntax, and (iii) relevant feedback from the underlying COOL interpreter/compiler (e.g., error messages and typing information) integrated directly into the VS Code editor to aid debugging. We believe that COOLer will help students enjoy writing COOL programs and consequently learn and appreciate advanced compiler concepts more effectively.

CCS Concepts

• **Software and its engineering → Integrated and visual development environments**; *Compilers*; • **Social and professional topics → Computing education**; *Computational science and engineering education*; • **Human-centered computing → User interface design**.

Keywords

COOL, VS Code Extension, Compiler, LSP, IDE

ACM Reference Format:
Linhan Li and ThanhVu Nguyen. 2025. COOLer: A Language Support Extension for COOL in VS Code. In *34th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA Companion '25), June 25–28, 2025, Trondheim, Norway*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3713081.3731729

1 Introduction

COOL (Classroom Object-Oriented Language) is a language often used in undergraduate and graduate compiler classes. These include traditional courses at Stanford [15], UC Berkeley [1], University of Nebraska [11], University of Michigan [5, 17], and Vanderbilt [4], as well as online courses offered through Coursera [3], edX [16], and Stanford Online [14]. The language is sufficiently small for students to implement a complete interpreter or compiler within a semester but is still powerful enough to support major concepts in modern object-oriented programming languages, including polymorphism, inheritance, dynamic dispatch, type checking, and automatic garbage collection. As described by Alex Aiken, its author, the COOL language is *"object-oriented, statically typed, and has automatic memory management"* [1].

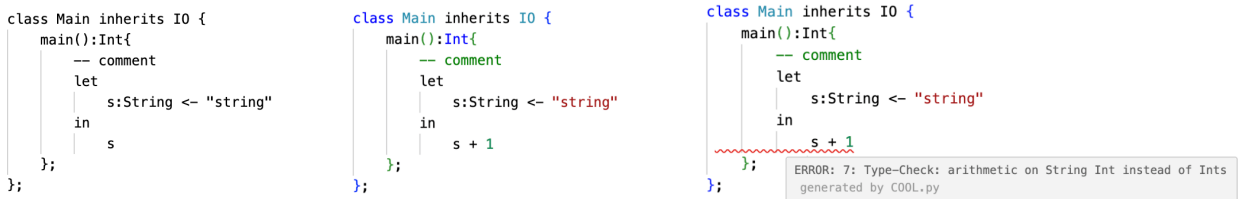
Although COOL is the chosen language for many compilers and programming languages courses, its syntax and semantics are often unfamiliar to students. As a result, students only write simple COOL programs to test their compiler implementations. Such limited tests cause students to overlook many important language features and thus miss crucial compiler designs required to handle these features (e.g., dynamic dispatch, inheritance, complex memory management). The reason for this is not that students are unwilling to create more complicated and thorough tests, but rather because they are not enthusiastic about writing long and complex COOL programs without support from their preferred code editors. More generally, because students have a difficult time writing COOL programs, they do not appreciate the language and lack the motivation to develop the compiler to support its features–ultimately missing key opportunities to deepen their understanding of compiler design.

Existing IDE extensions for COOL include `language-cool` [8] for VS Code, `cool-highlighter` [13] for Sublime Text, and `atom-language-cool` [12] for Atom. These provide only basic syntax highlighting and are largely repackaged versions of the same set of rules. While basic highlighting can aid readability, it is insufficient to engage students effectively or make COOL programming convenient. The absence of features like auto-completion and integrated error reporting forces students to either memorize COOL's syntax, frequently consult the language manual, or run the provided reference compiler separately to validate their programs. As a result, many students end up writing COOL programs in plain text mode with little to no meaningful editor assistance.

To help students effectively learn COOL, we have developed COOLer, a portable and feature-rich extension designed for major IDEs that support the increasingly popular Language Server Protocol (LSP) [10]. COOLer provides comprehensive syntax highlighting for the *entire* COOL language by leveraging regular expressions from the COOL lexer and parser to accurately recognize COOL syntax. Additionally, COOLer offers intuitive and practical auto-completion capabilities commonly found in modern IDEs, such as code suggestions and automatic structure completion. These features assist students in writing correct code and greatly reduce the need to memorize COOL's syntax and grammar details. Finally, COOLer takes advantage of the client-server design of LSP to deliver real-time error-checking and feedback directly to users within the VS Code editor. For example, COOLer can automatically underline problematic lines of code and provide immediate explanations of

ISSTA Companion '25, Trondheim, Norway

ACM ISBN 979-8-4007-1474-0/2025/06
https://doi.org/10.1145/3713081.3731729



(a) No extension **(b) `language-cool`** **(c) `COOLer`**

Fig. 1: Comparing `COOLer` with other extension settings.

```
1  class Main inherits IO {
2      main():Int{{
3          out_string("Hello, World!\n");
4          let
5              a:Int <- in_int()
6          in
7              if a<0
8                  then 0
9                  else a
10             fi;
11     }};
12 };
```

Fig. 2: Simple Cool program in VS Code with COOLer

the compilation errors, helping students quickly understand and debug their COOL programs.

Fig. 1 shows the same COOL program with different VS Code extensions. Notice that with `COOLer`, the program is color-coded with syntax highlighting *and* semantics error highlighting—showing that the program has a type error (Fig. 1c).

2 Tool Availability

`COOLer` is open source and available on GitHub (https://www.github.com/dynaroars/COOL-Language-Support), and can be installed directly to VSCode via its Marketplace [6]. An archive version is hosted on Zenodo (https://doi.org/10.5281/zenodo.15058779). A YouTube demo is also provided (https://youtu.be/X8tk2Ik4j8k), showing how the tool works in practice.

3 COOL

COOL is an object-oriented (OO) language and shares some design and syntax with Java. Fig. 2 shows a small COOL program that prints `HelloWorld` (lines 1-3) and a simple if statement. The COOL manual, called CoolAid [2], provides the formal syntax and semantics of the language, which we summarize below.

Syntax. The syntax of COOL is designed to be simple. This allows students to write regular expressions—regexes—for the lexer and parser to recognize syntactically correct COOL programs. The standard COOL specification does not have advanced features such as list/array structures, threading/multi-processing supports, or exception handling.

Tab. 1 displays the syntax and context-free grammar of COOL. A COOL program is a set of COOL `classes`, and each class consists of `features` which are attributes (variables) and methods. Each class defines a type and thus the programmer defines new types

Tab. 1: Cool Syntax and Grammar

program	::=	[*class*;]$^+$
class	::=	**class** TYPE [**inherits** TYPE] { [*feature*;]*}
feature	::=	ID([formal[,formal]*]):TYPE { *expr* }
	|	ID:TYPE[<- *expr*]
formal	::=	ID:TYPE
expr	::=	ID <- *expr*
	|	*expr*[@TYPE].ID ([*expr*[*expr**])
	|	ID([*expr* [,*expr*]*])
	|	**if** *expr* **then** *expr* **else** *expr* **fi**
	|	**while** *expr* **loop** *expr* **pool**
	|	{[*expr*;]$^+$}
	|	**let** ID:TYPE [<- *expr*] [,ID:TYPE [<-*expr*]]* **in** *expr*
	|	**case** *expr* **of** [ID:TYPE=> *expr*]$^+$ **esac**
	|	**new** TYPE | **isvoid** *expr* | ~*expr* | **not** *expr*
	|	(*expr*) | *expr* [+|-|*|/] *expr* | *expr* [<|<=] *expr*
	|	ID | constant(*integer*|*string*|*true*|*false*)

$$\frac{\vdash e_1 : Int \qquad \vdash e_2 : Int}{op \in \{*, +, -, /\}} \qquad \frac{\vdash e_1 : Int(i_1) \qquad \vdash e_2 : Int(i_2)}{op \in \{*, +, -, /\}}$$
$$\vdash e_1\ op\ e_2 : Int \qquad v_1 = Int(i_1\ op\ i_2) \vdash e_1\ op\ e_2 : v_1$$

Fig. 3: Type Checking and Operational Semantic Rules

and associated data and methods by creating new classes (similar to Java). COOL is an expression language, and thus most COOL constructs are expressions. Expressions take up a large portion of the COOL syntax, but in general are relatively straightforward and similar to expressions in traditional languages. Note that the `let` expression that declares a new variable is similar to the one used in a functional language such as OCaml.

Type Checking and Semantics. COOL is a *type-safe* language and thus its compiler type-checks the input program to ensure no typing errors at runtime. The typing rules for COOL, defined in the CoolAid manual, provide deduction rules for COOL expressions (e.g., if `x` is an integer and `y` is an integer then the expression `x + y` results in an integer).

The evaluation of a COOL program is provided using operational semantics rules. Similar to type checking rules that deduce the types of COOL expressions, these operational semantic rules deduce values for COOL expressions (e.g., if `x` is 3 and `y` is 7 then the expression `x + y` results in 10). These rules are also specified

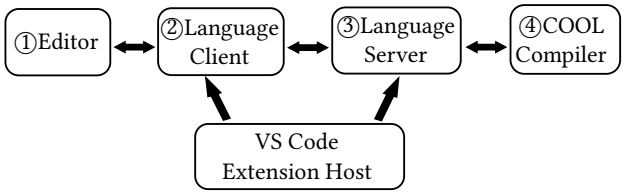


Fig. 4: Overview of COOLer

in the CoolAid manual. Fig. 3 shows the type checking and operational semantics rules for applying binary operators $*, +, -, /$ to expressions $e1$ and $e2$.

Reference Compiler. Students taking compiler courses using COOL often implement a full COOL compiler, which consists of several main phases, including lexing, parsing, type checking, semantic evaluation, and assembly code generation. To help students debug and make progress, the instructor provides a complete "reference" compiler, e.g., an executable binary compiled from a COOL compiler implementation written in C. Students then use the reference compiler to check each step of their implementation by comparing outputs and error messages. `COOLer` leverages the reference compiler to return error messages to the user.

4 COOLer

`COOLer` is composed of four components shown in Fig. 4.

① The **editor** provides a GUI interface to the user, and is the host of many editing features including syntax highlight (4.1) and auto-completion (4.2). It also interfaces with the language client (②) through its API to display information such as error messages from the COOL compiler.

② The **language client** interacts with the editor and the language server by sending and receiving information, e.g., getting the content of an opened file, cursor position, or display error messages. Whenever the client is activated, it creates a language server and sends data to the server.

③ The **language server** is a proxy to our COOL compiler. When a request from the client is received, the server retrieves the source code and sends it to the COOL compiler for analysis. The analysis results are composed into the LSP response format and returned to the client for display.

④ The COOL reference **compiler** performs standard compilation phases such as parsing, type checking, and communicating results (e.g., warning and error messages) to the language server).

Once installed, `COOLer` is automatically activated within VSCode when the user opens a COOL file with a `.cl` extension. It provides syntax highlighting and auto-completion at the editor level, and type checking and evaluation at the language server level.

4.1 Syntax Highlight

Syntax highlight in `COOLer` is handled by the VS Code editor. The editor tokenizes the source code using the regex rules specified in a `TextMate` configuration file. The editor then assigns colors to the tokens displayed in the IDE based on their defined roles as specified in IDE's current theme (e.g., comments are blue in some themes while red in others).

Tab. 2: Syntax Highlight Rules

Constructs	Regex Descriptions
integer	non-negative integer with no leading zero
ID(Class)	alphanumeric string starting with an uppercase letter
ID(other)	alphanumeric string starting with a lowercase letter
special ID	self, SELF_TYPE
string	alphanumeric string enclosed by double quotes
comment	line starting with "–" and block enclosed by "(*", "*)"
keywords	*class, else, false, fi, if, in, inherits, isvoid, let, loop, pool, then, while, case, esac, new, of, not, true*

```
"scopeName": "source.cool",
"name": "COOL",
"fileTypes": ["cl"]
"patterns":[
   "line_comment": {
     "begin": "--", "end": "$",
     "name": "comment.line.double-dash.cool"
   }
   "class": {
      "match": "\\b[A-Z][a-zA-Z0-9_]*\\b",
     "name": "entity.name.type.class.cool"
   }]
```

Fig. 5: Syntax Highlight Configuration in COOLer

For `COOLer`, we create a TextMate [7] JSON file consisting of regular expressions (regexes) for all of COOL's keywords, tokens, etc. Fig. 5 shows a snippet of a TextMate configuration having regexes for line comments and class identifiers. When the user opens a COOL (`.cl`) file, the `COOLer` extension is activated and uses these rules for syntax highlighting. Tab. 2 shows syntax highlighting regexes for the main constructs of COOL.

4.2 Auto Completion

`COOLer` supports standard auto-completion (also called "Intellisense" in VS Code [9]). This allows the user to type a few keywords and the IDE can suggest and insert common code structures. For example, when `COOLer` detects that the user is typing the keyword `if`, it will ask if the user wants to replace that with the COOL snippet `if condition then expression else expression fi`. This auto-completion is straightforward yet useful in code development in an IDE as they help accomplish the goal of reducing typing effort and syntax errors from the developer.

To offer syntactically and structurally correct snippets, `COOLer`'s auto-completion has knowledge about the syntax and grammar of COOL. Similar to syntax highlighting (§4.1), this is achieved at the editor level using a TextMate configuration file. This TextMate file consists of rules mapping prefix strings (e.g., `let`) with code snippets (e.g., `let var ... in ..`). The code snippets can have multiple placeholders for any sub-expressions, and the user can switch between them using the "Tab" key. Once the configuration file is loaded with the extension, the editor will automatically try to match the prefix with the string the user entered. If the input

matched (can be fully or partially) the prefix of any snippets, the name and description of the snippet will be listed in a drop-down menu for the user to select.

```
"COOL_class_inherits": {
   "prefix": "class",
   "body": [
      "class ${1:Name} inherits ${2:Object}{",
      "\t${0:body}",
      "};"],
   "description": "COOL: class inherits"}
```

Fig. 6: TextMate auto-completion Configuration

Fig. 6 shows an example of COOLer's TextMate configuration. Here, "body" defines the code snippet line by line, and the "${...}" marks the placeholders. This snippet defines the snippet for a `class` definition in COOL (i.e., when the user types `class`, the COOLer can fill in the skeleton for defining a class in COOL).

Example. Fig. 7 shows an example when COOLer senses that the user wants to declare a new variable through the `let` keyword and generates the snippet for new variable declaration with appropriate placeholders for specifying the type and initial value.

4.3 LSP-based analysis and interaction

COOLer uses LSP for feedback communication, i.e., displaying information from the backend COOL compiler such as errors and warnings to the frontend editor. The design of the LSP allows the analysis to be done in separate processes, and communicate the results with the editor through inter-process communication (thus analysis done in the server does not affect the user's interaction with the editor).

COOLer includes a pair of language client and server. The language server is used as a proxy to communicate with the COOL reference compiler. When the user saves their code changes, the server invokes the COOL compiler on the code and sends outputs to the client. The language client acts as a middle layer between the editor and the language server. It receives data such as error messages from the server and invokes the editor's API to display them to the user.

For COOLer, reporting data such as error messages from the COOL compiler to the client is achieved by formatting and translating the error message into a representation that the editor can understand and display. More specifically, when the compiler reports an error or warning is found in the program, the editor displays the line number, the stage of compilation, and the detailed error message. These errors and warnings can come from any compilation

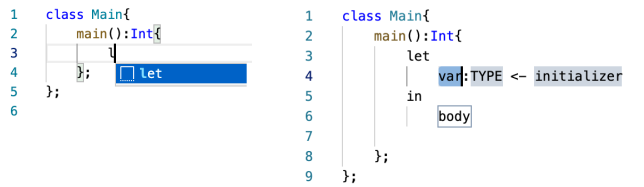


Fig. 7: Auto-completion in COOLer

phase, e.g., failure to parse certain expressions or having incorrect type associations.

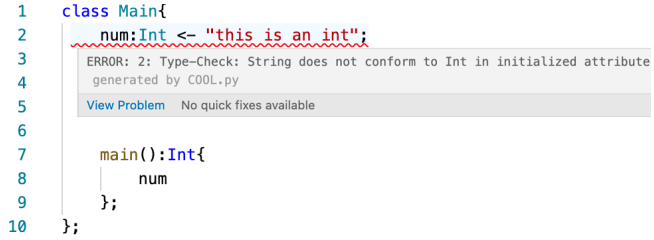


Fig. 8: Displaying Error Messages

Example. Fig. 8 shows how COOLer communicates error messages to the user via the editor. In line 2, the user assigns a string constant to variable `num` of type `Int`, which is a type error in COOL. This error is caught by the backend compiler and reported back to the user. The red wavy line indicates the location of the error, and hovering over the line will display a detailed error message as shown in the figure.

5 Conclusion

We presented COOLer, an LSP-based extension that allows existing IDEs and editors to support the COOL programming language.

We have shared COOLer to colleagues teaching courses involving COOL so that they can refer their students to try the tool. We have also started using COOLer in our compiler course. We will continue to maintain COOLer and add more features in based on feedback from our students and others. It is also worth mentioning that COOLer has been *downloaded and installed in VS Code over 1,000 times* as reported from VS Code Marketplace, suggesting that COOLer is being used by students and instructors in courses involving COOL. A user succinctly captured difficulty of COOL and the effectiveness of COOLer in a 5-star review: *"Static CA[1] saved my life when it comes to a language like this, so much that I'm willing to spend time to write a review".*

COOLer also demonstrates that researchers can easily make their work more accessible to users who are familiar with IDEs but not with traditional research tools. Researchers can integrate their static or dynamic analysis tools into the LSP and interact with users in the same way COOLer interacts with the COOL compiler. For instance, besides creating a command-line fault localization or program repair tool, developers can also build an LSP connection to the IDE, allowing users to directly interact with the tool through any editor supporting LSP. Given the popularity of IDEs like VS Code and their powerful extension ecosystems, we believe this approach will make research tools more attractive to users, who often enjoy exploring new extensions that are easy to install and use.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This material is based in part upon work supported by the National Science Foundation under grant numbers 2422036, 2319131, 2238133, and 2200621, and by an Amazon Research Award.

[1]CA stands for code analyzer—this user refers to COOLer as a static code analyzer.

References

[1] Alexander Aiken. 1996. Cool: A Portable Project for Teaching Compiler Construction. *SIGPLAN Not.* 31, 7 (jul 1996), 19–24.

[2] Alexander Aiken. 2025. COOL. http://theory.stanford.edu/~aiken/software/cool/cool.html. Accessed: July 28, 2025.

[3] edombowsky. 2025. edombowsky/coursera-compiler. https://github.com/edombowsky/coursera-compiler. Accessed: July 28, 2025.

[4] Kevin Leach. 2025. CS3276: Compilers. https://cumberland.isis.vanderbilt.edu/cs3276/. Accessed: July 28, 2025.

[5] Kevin Leach. 2025. EECS 483: Compiler Construction. https://dijkstra.eecs.umich.edu/eecs483/. Accessed: July 28, 2025.

[6] Linhan Li. 2025. COOL Language Support - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=Linhan.cool-language-support. Access: July 28, 2025.

[7] MacroMates. 2021. TextMate: Text editor for macOS. https://macromates.com/. Accessed: July 28, 2025.

[8] Guangming Mao. 2016. language-cool - Visual Studio Marketplace. https://marketplace.visualstudio.com/items?itemName=maoguangming.cool

[9] Microsoft. 2025. IntelliSense in Visual Studio Code. https://code.visualstudio.com/docs/editor/intellisense, accessed on July 28, 2025.

[10] Microsoft. 2025. The Language Server Protocol. https://microsoft.github.io/language-server-protocol. Accessed: July 28, 2025.

[11] ThanhVu Nguyen. 2025. Compiler Construction. https://nguyenthanhvuh.github.io/class-compilers/index.html. Accessed: July 28, 2025.

[12] Jeff Principe. 2025. atom-language-cool. https://github.com/princjef/atom-language-cool. Accessed: July 28, 2025.

[13] Jeff Principe. 2025. Sublime Cool Highlighter. https://github.com/princjef/sublime-cool-highlighter. Accessed: July 28, 2025.

[14] Stanford University. 2025. Compilers | Stanford Online. https://online.stanford.edu/courses/soe-ycscs1-compilers. Accessed: July 28, 2025.

[15] Stanford University. 2025. CS143: Compilers. https://web.stanford.edu/class/cs143/. Accessed: July 28, 2025.

[16] Stanford University. 2025. Stanford Online: Compilers | edX. https://www.edx.org/learn/computer-science/stanford-university-compilers. Accessed: July 28, 2025.

[17] Westley Weimer. 2025. CSCI 2320: Compilers. https://weimer.github.io/csci2320/index.html. Accessed: July 28, 2025.