Developing a Visual Studio Code Language Support Extension for the Snail Programming Language

Charles Reinhardt

June 27, 2023

Abstract

A high quality programming environment (often an integrated development environment, or IDE) can be vital to enhancing developer productivity. Visual Studio Code (VS Code) is a popular, open-source text editor maintained by Microsoft. VS Code delivers language-specific features through freely dowloadable, community-built extensions on an online marketplace. Many of these extensions allow developers to take advantage of editing features such as syntax highlighting, code-autocompletion, or debugging support. The snail language (Strings Numbers Arrays and Inheritance Language) is a simple, object-oriented programming langauge meant to be implemented in a one-semester undergraduate course. We present a new VS Code extension to provide language support for the snail language. The extension implements support for syntax highlighting, rudimentary auto-completion, and static error-checking diagnostics using VS Code's Langauge Server Protocol. This report summarizes the contents of a VS Code extension and gives an overview of how an extension runs, particularly highlighting the functions of VS Code's Langauge Server Protocol. I also discuss how this extension can be futher developed to make use of VS Code's Debug Adapter Protocol to implement a debugger with breakpoints, start/stop behavior, and variable inspection.

1 Introduction

Much of software development in the present day takes place in integrated development environments (IDEs) [22]. An IDE is a collection of software development tools, such as a code editor, debugger, and build system, often unified under a similar user interface, with the goal of simplifying the software development process and enhancing developer productivity [14, 53]. In addition to composing these tools together, many IDEs also offer advanced features within their code editors such as syntax highlighting, code auto-completion, and error-checking diagnostics. Today, there are any number of different IDEs available for use with any given programming language, many offered as freely-downloadable for public use. For example, a developer looking to write code in Java may choose to do so in Eclipse, IntelliJ IDEA, or BlueJ [11, 17, 23]. With so many high quality IDEs available today, it is no surprise that 75% of software developers today use an IDE in their everyday work [22]. Clearly, the IDE has become an integral part of the software development process today [57].

Visual Studio Code (VS Code), is a popular, open source text editor maintained by Microsoft [36, 47]. When first downloaded, VS Code is a lightweight text editor with minimal features. However, a number of community-built, freely downloadable extensions offer advanced langauge features. These extensions can be dowloaded on VS Code's online extension marketplace, and can turn VS Code into a very fast ,robust, and powerful development environment for any programming task or language [38].

The snail programming language (Strings Numbers Arrays and Inheritance Language) is a simple, object-oriented programming language meant to be implemented in a one-semester undergraduate course [3]. In order to be implemented in a short time frame, snail is defined by limited features and a relatively annoying syntax. The language lacks a for-loop structure, opting instead to provide only while-loops. Each if statment *requires* an else clause, even when the

developer does not need to use one. Every statement must end with a semi-colon, which is not an issue until you accidentally forget one, and the resulting parse error message is wildly uninformative. While this design makes snail easier to implement, it makes it hard for a developer to write programs in snail.

Currently, there are no tools to offer advanced langauge support for the snail language. This is no surprise, as snail has a small user base and was first released in February 2022 [2]. With no external support for the language, software developers are taken out of their comfort zone and are offered no guidance when navigating the snail language.

This report presents the Snail Language Support VS Code extension, which seeks to address the lack of programming support tools for the snail programming language. Snail Langauge Support provides several important features to make programming in the snail language easier. First, it features syntax highlighting to make reading snail code easier and help highlight key structures or keywords of the language. Further, it features rudimentary auto completion with auto-closing brackets, braces, and quotes, as well as if-else, while loop, and class definition snippets, reducing the burden of memorizing snail's strict and unintuitive syntax. The extension also has automatic, real-time error checking diagnostics that allow a user to see syntax or parse errors in a snail program before running it for themselves. Finally, the extension is structured to support a full debugger with breakpoints, step-in, step-over, and step-out functionality.

This paper will outline the process of building the Snail Language Support VS Code extension.[1] Specificaly, we will introduce the structure of a VS Code extension that is meant to add support for new programming langauges. We will also discuss how this extension uses VS Code's language server protocol (LSP) to provide realtime error diagnostics [35]. We will address how the Snail Language Support extension may be further developed to include debugging support with breakpoints, step in/out behavior, and variable inspection, particularly highlighting the role of VS Code's debug adapter protocol (DAP) [33]. Finally, we will review good software development practices such as version control and documentation.

2 Background

In this section, we will discuss the history of integrated development environments (IDEs) and debuggers, and how they both assist software developers today. We will also discuss Visual Studio Code (VS Code) in more detail, identifying the technologies that power VS Code.

2.1 History of the Modern Integrated Development Environment

The first programming environment to remotely resemble a modern IDE was the Dartmouth BASIC programming language run on the Dartmouth Time Sharing System (DTSS), developed in the mid 1960s [25, 26]. Dartmouth BASIC was an example of a compile and go system, where program compilation was not separated from program execution [60]. Additionally, the DTSS also placed focus on making sharing time on a single university computer a simpler task, in order to help make programming more accessible to novices. This makes the DTSS an early example of combining multiple software development tasks into a single programming environment.

Borland's TurboPascal takes this idea one step further. Released in 1983, TurboPascal featured a Pascal compiler, code editor, file navigation user interface, and a rudimentary debugger [12, 21, 59]. This time period would also see language-specific IDEs in Microsoft's Visual BASIC 1.0, Microsoft's QuickC, and Borland's Turbo C/C++, all featuring similar characteristics of early IDEs [8, 10, 49, 50].

While early IDEs certainly had their merits, they usually only supported one programming language. Microsoft's Visual Studio, released in 1997, was one of the first IDEs to package support for a variety of programming languages in one piece of software [32]. Visual Studio also featured extensive tools to aid in development for software on the early internet. T oday, we see a trend

[1] This paper is formatted using LaTeX. Recognizing that using LaTeX is really hard, the LaTeX source code for this paper can be found hosted on GitHub: https://github.com/CharlesReinhardt/snail-language-support-paper/

towards open source IDEs such as Eclipse, NetBeans, or VS Code [4, 11, 36]. Even open source editors such as these include advanced development features such as an intelligent code editor, debugging tools, and version control integration. Many of these editors have advanced features within their code editors, providing functions such syntax highlighting, code autocompletion, and realtime error diagnostics.

There are many benefits to developing software using modern IDEs. By bundling code editing, build systems, and program execution into one tool, modern IDEs reduce the time and effort a developer needs to put forth in order to test a piece of code [14]. This also reduces the number of decisions a developer has to make while developing code, which can help increase productivity. Using an IDE can also standardize the software development process, by either helping a group of people use a consistent UI (and know how to help eachother), or allow a developer to avoid switching applications to complete a single task [58].

2.2 History of Modern Debugging Tools

Debugging is the process of searching for and fixing unexpected errors in a piece of code [7]. Early techniques of debugging software involved physically printing machine output and reading, step-by-step, through code and output, searching for a potential error [52]. Some computer systems, such as the IBM 704 Data Processing system released in 2968, allowed programmers to print information stored in specified memory locations in specified forms to assist with debugging activities [5, 6].

The **adb** and **dbx** debuggers for the Unix operating system saw a release in the 1970s and 1980s [27, 28]. These tols introduced the concept of breakpoints, which help a developer pinpoint the location of a fault in a piece of code. Both **adb** and **dbx** were run on the command line. Next, came the GNU project debugger (**gdb**), another command line debugger, released in 1986, which gave the user even greater control of tracing a program's execution throughout its runtime [16]. With the ability to debug low-level languages like C and Assembly, **gdb** is still in use today.

Modern debuggers with graphical user interfaces (GUIs) include the Visual Studio Debugger, which allows a developer to track variable values, function calls, and even change pieces of code or values of expressions while debugging [24]. Eclipse, a popular IDE for Java, has a similar debugger with a user friendly GUI [54]. Today, many developement workflows include the use of debugging tools [31].

Debugging tools can be of great assistance to developers. By allowing developers to more closely and quickly inspect a program's execution, debugging tools can reduce the amount of time a developer spends debugging, and thus enhance developer productivity [62].

2.3 What is Visual Studio Code?

Visual Studio Code is a popular, open source code editor. Visual Studio Code is designed to be fast and lightweight, with a focus on allowing a developer to write, test, adn debug source code quickly [45]. To achieve this, VS Code uses native, web, and language-specific technologoies. Tools like Electron allow VS Code to run on multiple platforms and operating systems using common web technologies like JavaScript, HTML, and CSS [9].

While VS Code is lightweight and fast, it still supports advanced features such as build or debugger tools available through the online VS Code extension marketplace [38]. VS Code is designed for extensibility, providing robust APIs to allow independent developers to create and publish extensions [37]. As a result, VS Code is able to grow and develop along with its community of users.

Existing extensions make VS Code a popular code editor [47]. The CodeSnap extension allows a user to capture and share improved screenshots of code [1]. GitLens adds additional visualization tools to help developers contribute to Git repositories while editing in VS Code [15]. Language Support for Java(TM) provides intellisense, formatting, refactoring, and build system support for the Java language [19]. The result of this extension support is an IDE that is used by over 75% of professional and hobby software developers [47].

3 Anatomy of a Visual Studio Code Extension

In this section, we will discuss the contents of a Visual Studio Code (VS Code) extension. First, we will highlight the directory structure of a typical language support VS Code extension. Next, we will discuss how a VS Code extension runs, from the moment it starts up to the moment it shuts down. We will also overview what can be provided to the user by a VS Code extension, specifically highlighting language configurations, autocompletion, and syntax highlighting.

3.1 Directory Structure

Most VS Code extensions share a similar directory structure. These structures may vary depending on what purpose the extension is meant to serve. In figure 1, we see a diagram of a typical language support VS Code extension. First, the **`.vscode`** directory configures how we run and test our exstension locally. Through the **`launch.json`** and **`tasks.json`** files, we define various tasks and configurations to streamline the build and testing process.[2] Note that these configuration files do not get packaged with a released VS Code extension. Next, a **`README.md`** file allows us to advertise the features of our extension. This file is rendered in the VS Code extension marketplace and is meant to be a user's first impression of an extension.

Snail Langauge Support is built with TypeScript. TypeScript is a syntactic superset of JavaScript that allows the developer the benefits of a type system without sacrificing JavaScript's flexibility [55]. TypeScript files are denoted with a **`.ts`** suffix. Using the TypeScript compiler (**`tsc`**), a program in TypeScript is transpiled to an equivalent JavaScript program. With a **`tsconfig.json`** file in an extension directory, we define how the TypeScript compiler handles an extension's TypeScript code. We define where the compiler looks for our **`.ts`** files, where to place resulting JavaScript files, and the strictness of type checks during transpilation.

The **`src`** directory contains the aforementioned TypeScript source code that runs our extension. In this directory, the **`extension.ts`** file defines special actions for our extension and launches the extension within an existing VS Code window. The **`server.ts`** file defines a language server that allows our extension to take advantage of realtime error diagnostics. We further discuss language servers later in this paper.

Lastly, the **`package.json`** file defines which features and configurations our extension supports. It also contains some biographical information about the extension, such as the author, publisher, or name of the exstension.

To build the structure for Snail Language Support, we used the **`Yeoman`** tool to generate an extension skeleton [61]. This skeleton was instrumental in providing basic configuration and source files to modify into the Snail Language Support extension.

3.2 Extension Lifecycle: From Startup to Shutdown

When a VS Code window launches, it also launches all extensions the user has downloaded and enabled by calling each extension's **`activate`** function, defined in the extension's source code. In Snail Language Support's case, the **`activate`** function is defined in the **`extension.ts`** file. This file has access to the VS Code extension API, allowing a developer to define certain VS Code client UI updates [37]. For example, Snail Language Support can display a VS Code UI style error message to the user if the user does not have a compatibale version of the snail language dowloaded.

The **`activate`** function is responsible for a few different behaviors. First, the **`activate`** function registers a few command handlers to watch for commands input from a user. A command handler is an example of the observer design pattern, which allows a developer to address a variety of potential input events that are experienced during runtime [13]. A developer can register

[2] From personal experience, we strongly recommend taking the time to understand these files before developing a VS Code extension. It will likely save painstaking debugging in the near future. See launch configurations: https://code.visualstudio.com/docs/editor/debugging#_launch-configurations and tasks: https://code.visualstudio.com/docs/editor/tasks

an observer that listens for a particular event and executes a block of code once that event is detected. Snail Language Support defines a few command handlers to detect when a user chooses to run or debug a snail file. We will touch on the topic of debugging within Snail Language Support more later.

The **activate** function also launches a snail language server, which communicates with VS Code via VS Code's Language Server Protocol (LSP). We further discuss the topic of language servers later in this paper.

Once the extension setup inside the **activate** function is complete, VS Code enables the extension within the workspace, allowing a developer to utilize the functions of an extension. Once the VS Code window is closed, or the extension is manually disabled, VS Code calls the **deactivate** function, which shuts down any separate processes left over from the extension run.

To inspect Snail Language Support's **extension.ts** file more closely, see appendix A

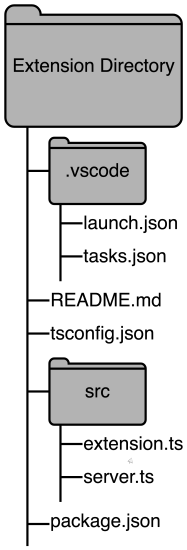


Figure 1: The typical directory structure of a VSCode language support extension.

3.3 Extension Contributions: The Extension Manifest

Recall that the **package.json** defines what our extension contributes to the user. We now discuss some of those contributions here.

First, we can define some metadata about our extension, such as a name, author, description, and version. We can also define which version of the VS Code engine this extension expects. This ensures that an extension knows which VS Code APIs it has proper access to. For example, an extension that expects a VS Code version of **1.7.0** won't be able to take advantage of an API released in **1.8.0** [42]. We can also define some categories that our extension falls under, helping our extension gain visibility on the online VS Code marketplace.

Next, and very importantly, we define where the source code is located for our extension. Through the **main** attribute, we define the entry point for our extension. This is the file that contains the **activate** function. For Snail Language Support, this is the **/out/extension.js** file. Notice that this is the transpiled version of our **src/extension.ts** file. For a more detailed look at Snail Language Support's **package.json** file, see appendix B.

3.3.1 Language Configuration

Snail Language Support also provides language configuration settings supported by a VS Code extension. These configurations are found in **language-configuration.json** file. We can define what is considered a commend through the **comments.lineComment** or **comments.blockComment** attributes, which allows a user to comment lines using VS Code's comment shortcut. Similarly, we can also define auto-closing structures such as parentheses or brackets through the **autoClosingPairs** attribute, which allows VS Code to automatically place closing parentheses or brackets. This is particularly useful for snail, as more opening parentheses or brackets require a closing partner. By defining **autoClosingPairs**, a developer can expend less mental energy ensuring that each parenthesis and bracket has a closing pair, and more mental energy on developing code. The **language-configuration.json** file for Snail Language Support can be found in appendix C. For more attributes and configuration options, see the VS Code documentation on language configurations [39].

3.3.2 Rudimentary Autocompletion Through Snippets

Snail Language Support provides autocompletion suppport for common structures such as if-else clauses, while loops, and class definitions. This autocompletion is implemented through VS Code snippets. A snippet is a commonly used code structure that is defined by a **`prefix`** (or a few options for a **`prefix`**) and a **`body`**. When a user types a snippet **`prefix`**, they are given the option to replace the keyword with the **`body`** of the snippet to complete the structure. Snippet bodies can also contain placeholders, which a user can visit in sequence while pressing their **`tab`** button. For a full listing of the snippets that Snail Language Support provides, see appendix D. Similarly, for more information on snippets in VS Code, see the documentation [43].

3.3.3 Syntax Highlighting

Snail Language Support also supports syntax highlighting for the snail language as shown in figure 2. VS Code's syntax highlighting makes heavy use of TextMate language grammars. A generic language grammar is a set of rules that govern what constitutes a valid statement for a given programming language [48]. A TextMate grammar is a specific format for defining a language grammar [29]. TextMate grammars define patterns that match special elements of a text document and assign scope names to these elements. For example, keywords of a particular language might be assigned the **`keyword.control`** scope name. These patterns are defined with regular expressions.

A regular expression is a string of text that can be used to match a particular pattern in a piece of text. Regular expressions are used in a variety of programming activities, such as lexing a program or to find text in a code editor [20, 56].

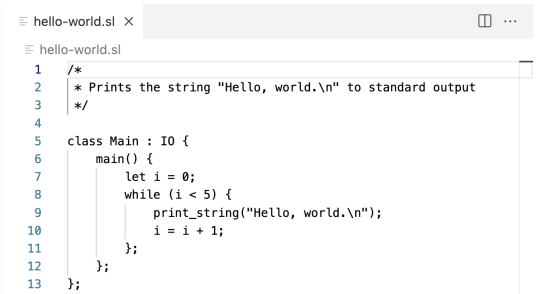
TextMate grammars make use of the advanced Oniguruma regular expression syntax, developed by K. Kosako [30, 44]. The Oniguruma regular expression syntax offers advanced features such as case insentivity options, character groupings, quantifiers, and anchors.

Snail Language Support defines a TextMate grammar in the **`snail.tmLanguage.json`** file, shown in appendix E. Taking a closer look at the structure of this document, we see three main relevant attributes: **`$schema`**, **`patterns`**, and **`repository`**.

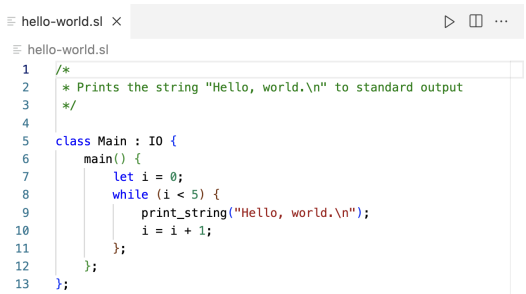
First, the **`$schema`** attribute links to a TextMate grammar schema document. This allows our editor to verify our **`snail.tmLanguage.json`** file and ensure we are defining valid attributes.

Next, we define a series of **`patterns`** using regular expressions. VS Code's tokenization engine reads our snail file one line at a time and tries to match the text to a pattern (or combination of patterns) defined in our **`patterns`** attribute. If the engine is successful in matching snail text to a pattern, it assigns that text a scope name. VS Code then uses these scope names to highlight the text appropriately, depending on the chosen color theme.

The engine tries matching text in the order we define our patterns. This means that the first pattern we define in **`patterns`** is the first regular expression that the tokenization engine tries to match to a line in a snail file. It is important to pay attention to the order that patterns are defined.



(a) Developing snail code without syntax highlighting can be a mentally taxing experience.



(b) Syntax highlighting makes developing snail code much easier.

Figure 2: Demonatrating the benefit of Snail Language Support's syntax highlighting feature.

For example, Snail Language Support defines comments first, which ensures that all snail text, no matter how complicated or syntactically complex, is highlighted as a simple comment.[3]

TextMate grammars also allow you to nest patterns inside of one another. This is essential for snail, as it allows the tokenization engine to match text inside of parentheses or brackets. TextMate grammars' ability to recursively reference the grammar itself (through the **`$self`** keyword) is also essential for this functioanlity.

For Snail Language Support, we define our patterns in the **`repository`** attribute. This allows us to reference them in multiple places in our **`patterns`** attribute. We feel this makes the document easier to understand and modify when necessary.

It is important that Snail Language Support's syntax highlighting can handle a snail file with incorrect syntax. For example, if a snail file uses a class definition with the incorrect syntax, Snail Language Support should still be able to highlight the code contained in that class. To achieve this, we must make sure that all patterns are accessible from the top level of the grammar. While any pattern may be nested inside of another pattern, that pattern must be present elsewhere in the grammar so that it is not *only* accessible after matching the other pattern.

4 Language Servers and the Language Server Protocol (LSP)

Snail Language Support also provides realtime lexing and parsing error diagnostics when developing code in snail. To do this, Snail Language Support uses a language server that communicates via VS Code's langauge server protocol (LSP). We now discuss what a language server is, how it relates to the LSP, and how Snail Language Support uses langauge servers.

4.1 What is a Language Server?

A language server is a tool used to provide language-specific editing features such as advanced autocomplete, go-to definition, or automatic error checking [35]. Many IDEs today utilize language servers by launching a language server process that runs in the background while a developer writes code. The development tool then sends a *requests* to the language server for, and the language server *responds* with the information requested by the development tool. The development tool then displays the information to the developer. This process is displayed in figure 3.

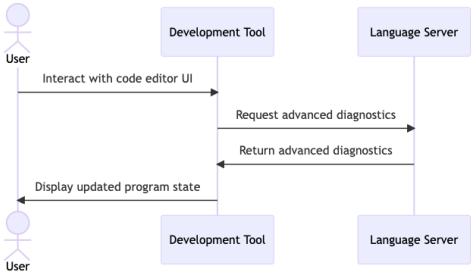
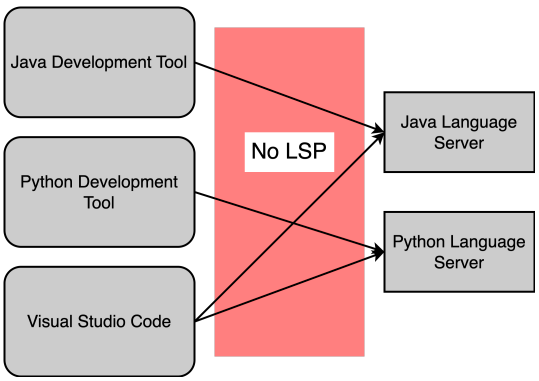


Figure 3: A development tool will communicate with a language server to display advanced diagnostics.

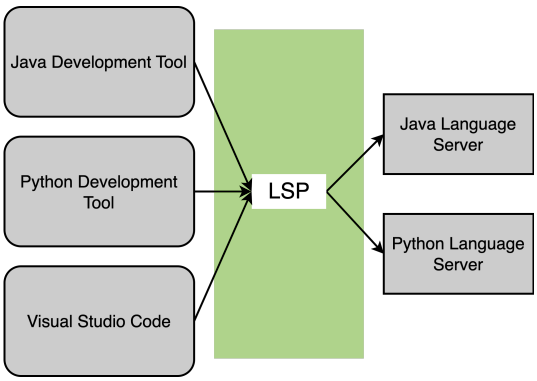
Many language servers are implemented in the language they assist with [40]. For example, a typical Python language server would be implemented in Python. This makes integrating language servers into development tools difficult, as many development tools are written in languages other than the language they are meant to develop. Further, a single language server might have to interact with a number of different development tools. For example, a Python language server might have to interact with VS Code, PyCharm, and Spyder. Without a standard framework for communication between development tools and language servers, development tools and language servers must implement specific functionality for each pairing. This problem is displayed in figure 4a.

Each development tool has different standards for how they interact with language servers. To address this, a development tool may need to adjust how they request and recieve information

[3] While defining our regular expression patterns, we found the Rubular program essesntial. It allowed us to iterate and test regular expressions more rapidly. We strongly recommend investigating this site while developing a TextMate grammar of your own: https://rubular.com/



(a) Without the Language Server Protocol, development tool and language server communication can get messy.



(b) The Language Server Protocol streamlines the communication between development tools and language servers.

Figure 4: Demonatrating the benefit of VS Code's Language Server Protocol.

from each language server it interacts with. Alternatively, a language server might adjust how it recieves and responds to requests for each development tool it interacts with. Both options are not ideal.

4.2 The Language Server Protocol

The Langauge Server Protocol (LSP) defines a standard framework for development tool and language server communication [35]. A development tool is able to communicate with any language-specific language server by sending requests and recieving responses according to the LSP standard.

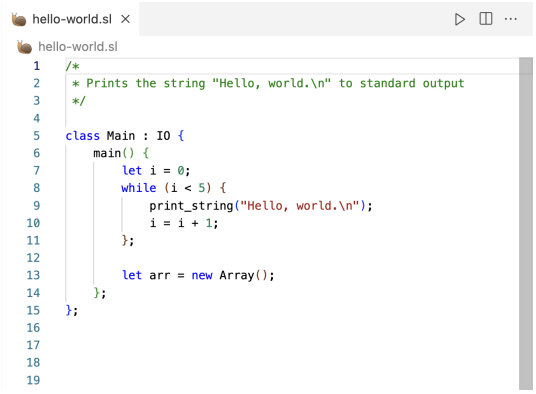
The LSP defines a number of common interactions between a development tool and a langauge server. For example, a development tool might send the **`DidchangeTextDocument`** notification when a development tool detects that a text document has been edited. A language server might validate this text document and send any notable diagnostics back to the development tool.

VS Code's LSP relies on requests and responses [41]. A request is a message that a development tool sends to a language server, asking the language server to perform some operation. A language server responds to the request by sending a corresponding response message with information about the executed operation. In order to follow the LSP, a language server **must** send a corresponding response upon recieving any requests. For example, a development tool might send the **`Initialize`** request to a language server, and a language server must respond with the **`InitializeResponse`** before the development tool is allowed to send any further requests. This contract goes both ways: when a language server sends a request to a developement tool, the development tool must respond.

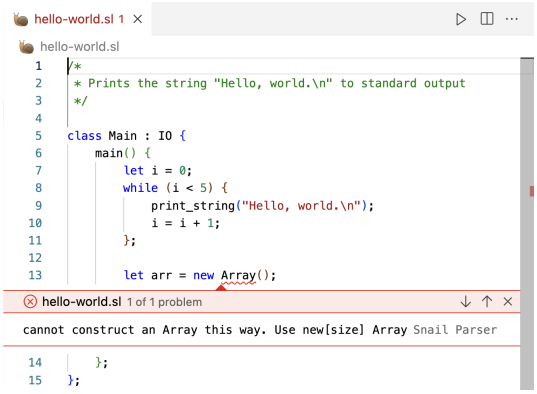
VS Code's LSP also uses notifications. A notification is a message that does not require a response. A development tool can send a notification to a language server and the language server can process the notification as it sees fit. Often, the language server will send information back to the development tool (in the form of another notification), but it is important to note that the language server is not required to respond.

With an established standard for communication as shown in 4b, both development tools and language servers can streamline their communications systems. Additionally, a developer producing one of these tools or features can rely on a consistent standard to make the development process easier.

In VS Code extensions, a language server (using LSP) can provide a number of different features for a programming language. By implementing support for the **`textDocument/hover`** request, an exstension could display additional documentation about code when a user hovers their cursor over some text. By implementing support for the **`widnow/showMessage`** request, an



(a) Developing snail code without realtime error diagnostics can make it difficult to spot syntax errors.



(b) With Snail Language Support, a developer can spot easily spot syntax errors and fix them before runtime.

Figure 5: Demonstrating the benefit of realtime error diagnostics provided in Snail Language Support.

extension can display an error message in VS Code's UI when code fails to build. With the **`textDocument/publishDiagnostics`** notification, an extension can validate that a program will run before a user actually tries to run it. For a full list of LSP specificatons, see the documentation [41].

4.3 Language Servers in Snail Language Support

In the Snail Language Support extension, we use a language server to validate snail files before runtime, checking for lexer and parser errors. With the extension enabled, VS Code is able to display lexer and parser errors, and highlight the relevant location of the error, before a developer runs the program themselves. This behavior is illustrated in figure 5.

To do this, Snail Language Support uses a snail language server. The majority of this language server is implemented in TypeScript in the **`server.ts`** file, which can be found in appendix F. This file responds to connections from a VS Code client and handles a few important LSP requests such as **`Initialize`** events and **`DidChangeTextDocument`** notifications.

The snail-specific features are provided by modifying the snail interpreter to act as a language server. By passing a **`-s`** or **`--server`** flag while executing a snail file (i.e. **`snail -s hello-world.sl`**), the snail interpreter outputs relevant error diagnostics including program exit status, error source, and error location.[4] The **`server.ts`** file handles this output and provides it to the VS Code client in the LSP format.

A high-level sequence of interactions between a developer, VS Code client, and snail language server is documented in figure 6.

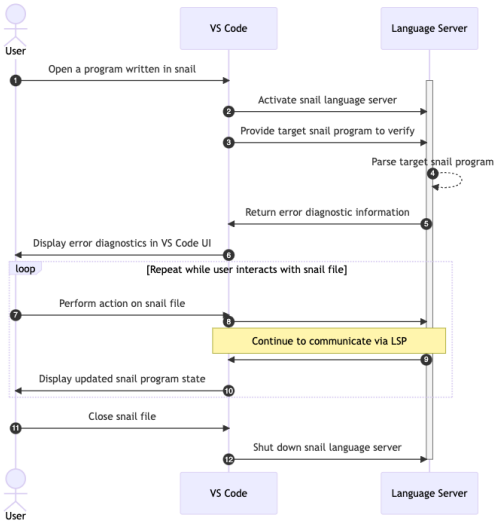


Figure 6: Modelling the interactions between a developer, a VS Code client, and snail language server while a developer opens and modifies a snail file.

[4] Further modifying the snail interpreter to handle a variety of potential language server requests (not just requests for error diagnostics) is a great area for future work.

First, a user opens a program written in snail
(1). When the VS Code client detects this activity, it launches a snail language server (2). Next, the VS Code client provides the snail program to verify (3) and the language server executes the snail program, using the aforementioned **`snail -s`** option (4). The snail language server returns the error diagnostics to the VS Code client (5) and the client displays these diagnostics to the developer (6). While the developer further modifies the snail file, the VS Code client and snail language server continue to communicate via LSP (7-10). When a developer is finished, they can close the snail file (11) and the VS Code client shuts down the snail language server (12).

5 Debug Adapter Protocol

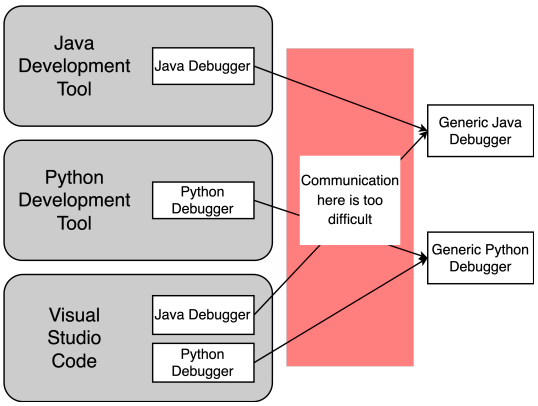
Snail Language Support is also structured to support the implementation of a debugger for snail. To do this, Snail Language Support defines a debug adapter that uses VS Code's Debug Adapter Protocol (DAP). In this section, we will discuss what the DAP is and how Snail Language Support uses it.

5.1 What is the Debug Adapter Protocol?

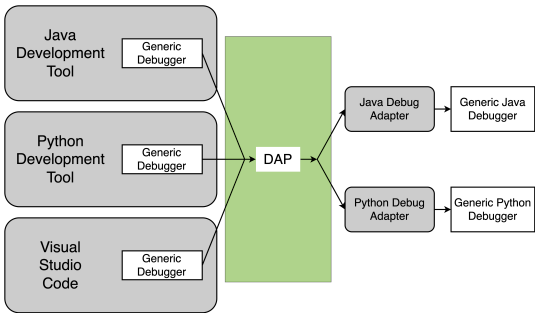
Debugging tools are an integral part of the software development process. Modern debuggers are often implemented with a user-friendly UI to help developers debug their programs [24, 31, 54]. Many of these debuggers are built intertwined with their corresponding code editor's UI, which can vary from code editor to code editor. As a result, valuable debugging logic is intermingled with code-editor-specific UI logic. This makes it nearly impossible to implement a debugger that is capable of communicating with different code editors [33]. This problem is illustrated in figure 7a.

Instead of asking a development tool to interface directly with a debugger, suppose a development tool interfaces with a debug adapter that translates a development tool's requests and sends them to a language-specific (but tool-agnostic) debugger. If the communication between the development tool and debug adapter is standardized, a language-specific debug adapter can be reused across multiple development tools.

VS Code's Debug Adapter Protocol (DAP) standardizes the communication between a development tool and debug adapter. As shown in figure 7b, this allows language-specific debuggers and debug adapters to be reused for multiple development tools.



(a) Communication between a generic debugger and a development tool is incredibly complicated.



(b) With the Debug Adapter Protocol, a development tool can easily communicate with a generic langauge-specific debugger.

Figure 7: Illustrating the benefit of the Debug Adapter Protocol in aiding development tool and debugger communication.

VS Code's DAP relies on requests and responses very similarly to the LSP [34]. A request is a message that a development tool sends to a debug adapter, asking the debugger to perform some debugging task. The debug adapter communicates with the language-specific debugger to perform this operation and sends a corresponding response message to the development tool. The debug adapter **must** send a corresponding response after recieving a request, even if the response is an empty message. The DAP also defines reverse requests, which function the same as regular requests except that they are sent from the debug adapter to the development tool.

VS Code's DAP also allows a debug adapter to send events to the development tool. This allows the debug adapter to update the development tool on a debugging session's state. For example, a debug adapter can send the development tool a **`Stopped`** event when program execution has stopped due to a breakpoint, step request completion, or expression evaluation completion.

5.2 Debug Adapter Protocol in Snail Language Support

Snail Language Support does not currently provide a debugger for the snail language. However, the extension is structured to support a snail debugger in the future.

Snail Language Support provides a debug adapter for the snail language in two parts, as shown in figure 8. First, an extension-level debug adapter is defined within Snail Language Support in the **`debugAdapter.ts`** file, as found in appendix G. This debug adapter is launched using the **`node.js`** runtime within the VS Code client. The VS Code client sends DAP requests to the extension-level debug adapter via standard input (**`stdin`**), and the debug adapter uses the transmission control protocol (TCP) to forward these DAP requests along to snail. Think of the extension-level debug adapter as a tool to echo DAP requests from VS Code along to snail.

Next, the DAP requests are processed in a snail-level debug adapter that is defined as an additional module alongside the snail interpreter. This snail-level debug adapter is defined in **`snaildap.re`** as shown in appendix H and is responsible for responding to DAP requests with the necessary snail-specific information.

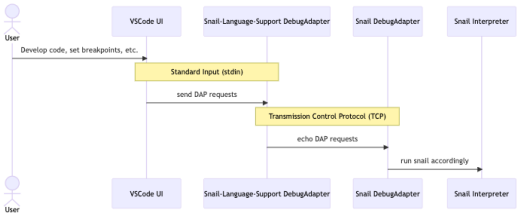


Figure 8: A high-level overview of VS Code, Snail Language Support, and snail debugging communication

5.2.1 The `snaildap` Snail Debug Adapter

The **`snaildap`** module is written using the Reason programming language. Reason is built on top of the strictly-typed functional OCaml programming language [51]. Reason introduces a JavaScript-like syntax on top of OCaml's type system, giving developers a familiar syntax while still providing improved code safety and maintanability. Anything written in Reason can be translated to OCaml (and vice versa), allowing developers to take advantage of OCaml tools when developing in Reason.

The **`snaildap.re`** debug adapter makes use of the **`ocaml-dap`** library developed by Hackwaly [18]. The **`ocaml-dap`** library is an implementation for VS Code's DAP in the OCaml langauge, allowing a developer to reference relevant types, requests, and commands when handling DAP requests in OCaml. While using the **`ocaml-dap`** library in Reason proves to be convenient, it does introduce one problem. After performing debug initialization, **`ocaml-dap`** requires a program to, essentially, sit and wait for DAP requests to be recieved. This prevents **`snaildap`** from actually running snail and aqcuiring program details to respond to DAP requests once they are recieved.

The **`snaildap`** debug adapter can solve this problem by making use of the Lightweight Thread (**`Lwt`**) library for OCaml and Reason. The **`Lwt`** library introduces the concept of a promise, a placeholder return object for time-consuming operations [46]. When a developer wants a program to continue running while waiting for operation to finish, they can introduce a promise that contains an empty value. Once the time-consuming operation is finished, the promise becomes fulfilled and

gains a meaningful value. During that time spent waiting for a promise to be fulfilled, a program is able to perform other operations. In the context of **`snaildap`**, the **`Lwt`** module allows **`snaildap`** to perform snail interpreter operations *and* wait for DAP requests *simultaneously*.

Taking a look at the **`snaildap.re`** file as shown in appendix H, the important initialization of our debug adapter can be found in three functions: the **`uninitialized`**, **`initialized`**, and **`debug`** functions.

In the **`unitialized`** and **`initialized`** functions, **`snaildap`** handles **`Initialize`** and **`Launch`** requests. If **`snaildap`** were to also support the **`Attach`** request, that behavior would be implemented here. The **`snaildap`** module also handles **`Disconnect`** requests when necessary.

Finally, in the **`debug`** function, we define behavior for the DAP capabilities that our debug adapter supports. Currently, Snail Language Support handles **`ConfigurationDone`**, **`Threads`**, **`Terminate`**, and **`Disconnect`** requests. This is enough for Snail Language Support to start and stop a debugging session via the VS Code UI, but not enough for a user to do any debugging to a snail program. Further work to include real debugging logic would be to implement behavior for a variety of DAP requests as shown in appendix I.

6 Good Practices in Software Development

6.1 Version Control

6.2 Documentation

7 Conclusions

References

[1] adpyke. Codesnap, Feb 2021.

[2] Kevin Angstadt. Version 1.0.0 released, February 2022.

[3] Kevin Angstadt. Introduction, May 2023.

[4] Apache. Apache netbeans releases, May 2023.

[5] IBM Archives, 2023.

[6] D. Arden, S. Best, F. Corbato, F. Helwig, J. McCarthy, A. Siegel, F. Verzuh, M. Watkins, and M. Weinstein. *Coding for the MIT-IBM 704 Computer*. Massachusetts Institute of Technology, 1957.

[7] AWS. What is debugging?, 2023.

[8] Michael Burgwin. History of development: Visual basic 1.0, May 2013.

[9] Electron. Build cross-platform desktop apps with javascript, html, and cs, 2023.

[10] Douglas Forer and Nicholas Petreley. Quickc is a one-stop development tool. *InfoWorld*, 13(46):113–114, Nov 1991.

[11] Eclipse Foundation. Eclipse installer 2023-03 r, Mar 2023.

[12] Zarko Gajic. Delphy history from pascal to embarcadero delphi xe 2, Mar 2017.

[13] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Observer*, page 293–303. Addison-Wesley, 1 edition, 1995.

[14] Alexander S Gillis and Valerie Silverthorne. What is integrated development environment (ide)?, Sep 2018.

[15] GitKraken. Gitlens, May 2023.

[16] GNU and Free Software Foundation. What is gdb?, May 2023.

[17] Kings Programming Education Tools Group, Sep 2022.

[18] Hackwaly. ocaml-dap, Sep 2021.

[19] Red Hat. Language support for java(tm), March 2023.

[20] Computer Hope. Regex. https://www.computerhope.com/jargon/r/regex.htm, Oct 2022.

[21] David Intersimone. Antique software: Turbo pascal v1.0. https://web.archive.org/web/20101221211755/http://edn.embarcadero.com/article/20693, December 2010.

[22] JetBrains. The state of developer ecosystem in 2019 infographic, 2019.

[23] JetBrains. Download intellij idea, Mar 2023.

[24] Mike Jones, Gordon Hogenson, J. Martens, Tom Pratt, William Anton Rohm, Genevieve Warren, Kraig Brockschmidt, Beth Harvey, Oscar Sun, and Kaycee. First look at the visual studio debugger, Sep 2022.

[25] John G. Kemeny and Thomas E. Kurtz. Dartmouth time-sharing. *Science*, 162(3850):223–228, 1968.

[26] Thomas E. Kurtz. *BASIC*, page 515–537. Association for Computing Machinery, New York, NY, USA, 1978.

[27] Bell Laboratories. *Unix Programmer's Manual*, volume 2. Bell Telephone Laboratories Inc., 7 edition, 1983.

[28] Mark A. Linton. The evolution of dbx. In *USENIX Summer*, 1990.

[29] MacroMates. Language grammars. https://macromates.com/manual/en/language_grammars, 2021.

[30] MacroMates. Regular expressions. https://macromates.com/manual/en/regular_expressions, 2021.

[31] Norman S. Matloff and Peter Jay Salzman. *The art of debugging with GDB and DDD: For professionals and students*. No Starch Press, 2008.

[32] Microsoft. Microsoft announces visual studio 97, a comprehensive suite of microsoft visual development tools, Jan 1997.

[33] Microsoft. Debug adapter protocol, 2021.

[34] Microsoft. Debug adapter protocol specification, 2021.

[35] Microsoft. Language server protocol, 2022.

[36] Microsoft. Download visual studio code, Apr 2023.

[37] Microsoft. Extension api, May 2023.

[38] Microsoft. Extensions for visual studio code, 2023.

[39] Microsoft. Language configuration guide. https://code.visualstudio.com/api/language-extensions/language-configuration-guide, Jun 2023.

[40] Microsoft. Language server extension guide. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide, Jun 2023.

[41] Microsoft. Language server protocol specification. https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/, June 2023.

[42] Microsoft. Publishing extensions. https://code.visualstudio.com/api/working-with-extensions/publishing-extension, Jun 2023.

[43] Microsoft. Snippets in visual studio code. https://code.visualstudio.com/docs/editor/userdefinedsnippets, Jun 2023.

[44] Microsoft. Syntax highlight guide, Jun 2023.

[45] Microsoft. Why visual studio code?, May 2023.

[46] Ocsigen. Lwt manual, Feb 2023.

[47] Stack Overflow. Stack overflow developer survey 2022, May 2022.

[48] pgrandinetti. What is a programming language grammar?, Sep 2019.

[49] Robert Plant and Stephen Murrell. *An executive's Guide to Information Technology: Principles, business models, and terminology*. Cambridge University Press, 2007.

[50] Maya Posch. Revisiting borland turbo c and c++, Apr 2023.

[51] Reason. What and why. https://reasonml.github.io/docs/en/what-and-why, Dec 2020.

[52] RevDeBug. Debugging history - origins, Feb 2020.

[53] Kateryna Shyniaieva. The most popular ides for developers in 2023, Feb 2023.

[54] Sarika Sinha. Debugging the eclipse ide for java developers, Jun 2017.

[55] TypeScript. What is typescript?, 2023.

[56] Cornell University. Lexical analysis and regular expressions. https://www.cs.cornell.edu/courses/cs4120/2022sp/notes.html?id=lexing, Spring 2022.

[57] Slava Vaniukov. 16 best ides for software development: Overview for 2023, Apr 2023.

[58] Veracode. What is ide or integrated development environments?, Dec 2020.

[59] Tom Wadlow. Turbo pascal. *Byte*, 9(7):267–278, Jul 1984.

[60] Martin H. Weik. *compile-and-go*, pages 260–260. Springer US, Boston, MA, 2001.

[61] Yeoman. Getting started with yeoman. https://yeoman.io/learning/, 2023.

[62] Qin Zhao, Saman Amarasinghe, Rodric Rabbah, Larry Rudolph, and Weng Wong. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. volume 4959, 03 2008.

Appendix A Snail Language Support extension.ts

The following is the **extension.ts** file from Snail Language Support.

TODO mesh TypeScript and Json formats to look more uniform

```
import * as cp from 'node:child_process';
import * as f from 'node:fs';
import * as lc from 'vscode-languageclient/node';
import * as path from 'path';
import * as v from 'vscode';


let client: lc.LanguageClient;

let snailTerminal : v.Terminal;
let RUN_SNAIL_FILE_CMD = 'snail-language-support.runSnailFile';
let DEBUG_SNAIL_FILE_CMD = 'snail-language-support.debugSnailFile';
let OPEN_SETTINGS_ACTION = 'Open settings';
let OPEN_SETTINGS_CMD = 'workbench.action.openSettings';

export function activate(context: v.ExtensionContext) {

  let snailPath : string = v.workspace.getConfiguration('snailLanguageServer').
      snailPath;
  let resp = validateSnailPath(snailPath);
  if (resp.status == "ERROR") {

    const errorMessage = v.window.showErrorMessage(resp.message,
        OPEN_SETTINGS_ACTION);
    errorMessage.then(choice => {
        if (choice === OPEN_SETTINGS_ACTION) {
            v.commands.executeCommand(
                OPEN_SETTINGS_CMD,
                'snailLanguageServer.snailPath');
        }
    })
    return;
  }

  context.subscriptions.push(v.commands.registerCommand(RUN_SNAIL_FILE_CMD,
      runSnailFile));
  context.subscriptions.push(v.commands.registerCommand(DEBUG_SNAIL_FILE_CMD,
      debugSnailFile));

  // The server is implemented in node
  const serverModule = context.asAbsolutePath(
    path.join('client', 'out', 'server.js')
  );
  // The debug options for the server
  // --inspect=6009: runs the server in Node's Inspector mode so VS Code can
      attach to the server for debugging
  const debugOptions = { execArgv: ['--nolazy', '--inspect=6009'] };

  // If the extension is launched in debug mode then the debug server options are
      used
```

```
  // Otherwise the run options are used
  const serverOptions: lc.ServerOptions = {
    run: { module: serverModule, transport: lc.TransportKind.ipc },
    debug: {
      module: serverModule,
      transport: lc.TransportKind.ipc,
      options: debugOptions
    }
  };

  // Options to control the language client
  const clientOptions: lc.LanguageClientOptions = {
    // Register the server for plain text documents
    documentSelector: [{ scheme: 'file', language: 'snail' }],
    synchronize: {
      // Notify the server about file changes to '.clientrc files contained in the
          workspace
      fileEvents: v.workspace.createFileSystemWatcher('**/.clientrc')
    }
  };

  // Create the language client and start the client.
  client = new lc.LanguageClient(
    'snailLanguageServer',
    'Language Server for Snail',
    serverOptions,
    clientOptions
  );

  // Start the client. This will also launch the server
  client.start();
}

function validateSnailPath(path : string) {
  let error_code : string = "ERROR";
  let success_code : string = "OK";
  try {
    f.accessSync(path, f.constants.F_OK);
  } catch (err) {
    let message : string = "File path to snail doesn't exist: " + path;
    return {
      status: error_code,
      message: message,
      body: err
    }
  }

  try {
    f.accessSync(path, f.constants.X_OK);
  } catch (err) {
    let message = "User does not have execute privelege on snail path: " + path;
    return {
      status: error_code,
      message: message,
```

```
      body: err
    }
  }

  const snailCapabilities = cp.spawnSync(path, ['-h'])
    .stdout.toString()
    .split("\n")
    .map((item, _idx, _arr) => {
        return item.trim().split(' ')[0];
    });

  if (!snailCapabilities.includes('-s')) {
    let message : string = "This version of snail does not support language server
        capabilities: " + path;
    return {
      status: error_code,
      message: message,
      body: null
    }
  }

  let message : string = "yay";
  return {
    status: success_code,
    message: message,
    body: null
  }
}

function runSnailFile() {
  if (snailTerminal === undefined) {
    snailTerminal = v.window.createTerminal("Snail");
  }
  snailTerminal.show();

  const filePath : String | undefined = v.window.activeTextEditor?.document.
      fileName;
  let snailPath : String = v.workspace.getConfiguration('snailLanguageServer').
      snailPath;

  snailTerminal.sendText(snailPath + ' ' + filePath)
}

function debugSnailFile(resource : v.Uri) {

  if (!resource && v.window.activeTextEditor) {
    resource = v.window.activeTextEditor.document.uri;
  }

  let config : v.DebugConfiguration = {
    name: "Launch Snail Debug",
    request: "launch",
    type: "snail",
    program: resource.fsPath
```

```
  };

  // this line calls 'node client/out/debugAdapter.js'
  v.debug.startDebugging(undefined, config);
}

export function deactivate(): Thenable<void> | undefined {
  if (!client) {
    return undefined;
  }
  return client.stop();
}
```

Appendix B Snail Language Support package.json

The following is the **`package.json`** file from Snail Language Support.

```
{
  "name": "snail-language-support",
  "displayName": "Snail Language Support",
  "publisher": "cprein19",
  "description": "Extension providing useful language support for the Snail
     programming language",
  "version": "1.0.0",
  "engines": {
    "vscode": "^1.71.0"
  },
  "categories": [
    "Programming Languages",
    "Snippets"
  ],
  "activationEvents": [
    "onLanguage:snail",
    "onCommand:snail-language-support.runSnailFile"
  ],
  "main": "./client/out/extension",
  "contributes": {
    "languages": [
      {
        "id": "snail",
        "aliases": [
          "Snail",
          "snail"
        ],
        "extensions": [
          ".sl"
        ],
        "configuration": "./configurations/language-configuration.json",
        "icon": {
          "light": "./images/snail.png",
          "dark": "./images/snail.png"
        }
      }
    ],
    "grammars": [
      {
        "language": "snail",
        "scopeName": "source.sl",
        "path": "./syntaxes/snail.tmLanguage.json"
      }
    ],
    "snippets": [
      {
        "language": "snail",
        "path": "./snippets/snippets.json"
      }
    ],
    "commands": [
```

```
      {
        "command": "snail-language-support.runSnailFile",
        "title": "Run Snail Program",
        "category": "Snail",
        "enablement": "!inDebugMode",
        "icon": "$(play)"
      }
    ],
    "menus": {
      "commandPalette": [
        {
          "command": "snail-language-support.runSnailFile",
          "when": "resourceLangId == snail"
        }
      ],
      "editor/title/run": [
        {
          "command": "snail-language-support.runSnailFile",
          "when": "resourceLangId == snail",
          "group": "navigation@1"
        }
      ]
    },
    "configuration": {
      "type": "object",
      "title": "Snail Language Support",
      "properties": {
        "snailLanguageServer.maxNumberOfProblems": {
          "scope": "resource",
          "type": "number",
          "default": 100,
          "description": "Controls the maximum number of problems produced by the
              server."
        },
        "snailLanguageServer.trace.server": {
          "scope": "window",
          "type": "string",
          "enum": [
            "off",
            "messages",
            "verbose"
          ],
          "default": "off",
          "description": "Traces the communication between VS Code and the
              language server."
        },
        "snailLanguageServer.snailPath": {
          "scope": "application",
          "type": "string",
          "default": "snail",
          "description": "Path to snail executable to use for Language Server"
        }
      }
    },
```

```
    "breakpoints": [
      {
        "language": "snail"
      }
    ]
  },

  "scripts": {
    "vscode:prepublish": "npm run compile",
    "compile": "tsc -b",
    "lint": "eslint ./client/src --ext .ts,.tsx",
    "postinstall": "cd client && npm install && cd ..",
    "test": "sh ./scripts/e2e.sh"
  },
  "devDependencies": {
    "@types/glob": "^8.0.1",
    "@types/mocha": "^9.1.0",
    "@types/node": "^16.11.7",
    "@typescript-eslint/eslint-plugin": "^5.30.0",
    "@typescript-eslint/parser": "^5.30.0",
    "eslint": "^8.13.0",
    "mocha": "^9.2.1",
    "typescript": "^4.8.4"
  },

  "repository": {
    "type": "git",
    "url": "https://github.com/snail-language/snail-language-support"
  },
  "bugs": {
    "url": "https://github.com/snail-language/snail-language-support/issues"
  }
}
```

Appendix C Snail Language Support language-configuration.json

The following is the **`language-configuration.json`** file from Snail Language Support.

```
{
  "comments": {
    "lineComment": "//",
    "blockComment": [ "/*", "*/" ]
  },
  "brackets": [
    ["{", "}"],
    ["[", "]"],
    ["(", ")"]
  ],
  "autoClosingPairs": [
    {"open": "{","close": "}"},
    {"open": "[", "close": "]"},
    {"open": "(", "close":")"},
     {"open": "\"", "close": "\"", "notIn": ["comment", "string"]},
    {"open": "'", "close": "'", "notIn": ["comment", "string"]}
  ],
  "surroundingPairs": [
    ["{", "}"],
    ["[", "]"],
    ["(", ")"],
    ["\"", "\""],
    ["'", "'"]
  ],
  "folding": {
    "markers": {
      "start": "^.*{\\s*\\b",
      "end": "^.*}\\s*"
    }
  }
}
```

Appendix D Snail Language Support snippets.json

The following is the **`snippets.json`** file from Snail Language Support.

```
{
  "if": {
    "prefix": "if",
    "body": [
      "if ($1) {",
      "\t$2",
      "} else {",
      "\t$3",
      "};"
    ],
    "description": "An if-else conditional"
  },
  "while": {
    "prefix": "while",
    "body": [
      "while ($1) {",
      "\t$0",
      "};"
    ]
  },
  "class": {
    "prefix": "class",
    "body": [
      "class $1 {",
      "\t$0",
      "};"
    ]
  },
  "class-inherits": {
    "prefix": "class-inherits",
    "body": [
      "class $1 : $2 {",
      "\t$0",
      "};"
    ]
  },
  "main": {
    "prefix": ["class M", "class m", "main", "Main"],
    "body": [
      "class Main {",
      "\n\tmain() {",
      "\t\t$0",
      "\t};\n",
      "};"
    ]
  },
  "main-inherits": {
    "prefix": ["main-inherits", "Main-inherits"],
    "body": [
      "class Main : $1 {",
      "\n\tmain() {",
```

```
      "\t\t$0",
      "\t};\n",
      "};"
    ]
  },
  "method-def": {
    "prefix": ["method-def"],
    "body": [
      "${1:method_name}($2) {",
      "\t$0",
      "};"
    ]
  },
  "let": {
    "prefix": "let",
    "body": [
      "let $1;"
    ]
  },
  "let-def": {
    "prefix": "let-def",
    "body": [
      "let $1 = $2;"
    ]
  }
}
```

Appendix E Snail Language Support TextMate Grammar

The following is the **`snail.tmLanguage.json`** file from Snail Language Support.

```
{
  "$schema": "https://raw.githubusercontent.com/martinring/tmlanguage/master/
     tmlanguage.json",
  "name": "Snail",
  "patterns": [
    {
      "include": "#comments"
    },
    {
      "include": "#classes"
    },
    {
      "include": "#keywords"
    },
    {
      "include": "#features"
    },
    {
      "include": "#expressions"
    },
    {
      "include": "#blocks"
    }
  ],
  "repository": {
    "classes": {
      "patterns": [
        {
          "match": "(?i)(class)\\s([a-zA-z_0-9]+)\\s*(:\\s*[a-zA-z_0-9]+)?\\s*",
          "captures": {
            "1": { "name": "storage.type.class.snail"},
            "2": { "name": "entity.name.class.snail"}
          }
        }
      ]
    },
    "features": {
      "patterns": [
        {
          "begin": "\\b([a-zA-Z_0-9]+)\\b\\(",
          "beginCaptures": {
            "1": { "name": "entity.name.method.snail"}
          },
          "patterns": [
            {
              "name": "variable.parameter.snail",
              "match": "[a-zA-z_0-9]+"
            },
            {
              "include": "$self"
            }
```

```
        ],
        "end": "\\)"
      },
      {
        "begin": "\\b(?i)(let)\\s([a-zA-Z_0-9]+)\\b(=)?\\b",
        "beginCaptures": {
          "1": { "name": "storage.type.variable.snail"},
          "2": { "name": "variable.name.other.snail"}
        },
        "patterns": [
          {
            "include": "$self"
          }
        ],
        "end": ";"
      }
    ]
  },
  "expressions": {
    "patterns": [
      {
        "begin": "\\b([a-zA-Z_0-9]+)\\b\\(",
        "beginCaptures": {
          "1": { "name": "entity.name.function.snail"}
        },
        "patterns": [
          {
            "include": "$self"
          }
        ],
        "end": "\\);"
      },
      {
        "begin": "\\(",
        "patterns": [
          {
            "include": "$self"
          }
        ],
        "end": "\\)"
      },
      {
        "begin": "([a-zA-z_0-9]+)\\[",
        "beginCaptures": {
          "1": { "name": "variable.other.snail" }
        },
        "patterns": [
          {
            "include": "$self"
          }
        ],
        "end": "\\]"
      },
      {
```

```
        "name": "constant.numeric.snail",
        "match": "\\b[0-9]+[.]?[0-9]*\\b"
      },
      {
        "name": "variable.other.snail",
        "match": "\\b[a-zA-z_0-9]+\\b"
      },
      {
        "name": "string.quoted.double.snail",
        "begin": "\"",
        "end": "\"",
        "patterns": [
          {
            "name": "constant.character.escape.snail",
            "match": "\\\\(n|t)"
          }
        ]
      }
  ]
  },
  "keywords": {
    "patterns": [
    {
      "name": "keyword.control.snail",
      "match": "\\b(?i)(if|else|while|for|new)\\b"
    },
    {
      "name": "constant.language.snail",
      "match": "\\b(?i)(true|false|isvoid)\\b"
    },
    {
      "name": "storage.type.class.snail",
      "match": "\\b(?i)(class)\\b"
    },
    {
      "name": "storage.type.binding.snail",
      "match": "\\b(?i)(let)\\b"
    }
    ]
  },
  "comments": {
    "patterns": [
      {
        "name": "comment.line.double-slash.snail",
        "match": "\/\/.*"
      },
      {
        "name": "comment.block.snail",
        "begin": "\/[*]",
        "end": "[*]\/",
        "patterns": [
          {
            "include": "#comments"
          }
```

```
        ]
      }]
    },
    "blocks": {
      "begin": "{",
      "patterns": [
        {
          "include": "$self"
        },
        {
          "include": "#comments"
        },
        {
          "include": "#blocks"
        },
        {
          "include": "#expressions"
        }],
      "end": "}"
    }
  },
  "scopeName": "source.sl"
}
```

Appendix F Snail Language Support server.ts

The following is the **server.ts** file from Snail Language Support.

```
import * as cp from 'node:child_process';
import * as f from 'node:fs';
import * as ls from 'vscode-languageserver/node';
import * as os from 'node:os';
import * as path from 'node:path';

import {
	TextDocument
} from 'vscode-languageserver-textdocument';


// Create a connection for the server, using Node's IPC as a transport.
// Also include all preview / proposed LSP features.
const connection = ls.createConnection(ls.ProposedFeatures.all);

// Create a simple text document manager.
const documents: ls.TextDocuments<TextDocument> = new ls.TextDocuments(
    TextDocument);

let hasConfigurationCapability = false;
let hasWorkspaceFolderCapability = false;
let hasDiagnosticRelatedInformationCapability = false;

connection.onInitialize((params: ls.InitializeParams) => {
	const capabilities = params.capabilities;

	// Does the client support the `workspace/configuration` request?
	// If not, we fall back using global settings.
	hasConfigurationCapability = !!(
		capabilities.workspace && !!capabilities.workspace.configuration
	);
	hasWorkspaceFolderCapability = !!(
		capabilities.workspace && !!capabilities.workspace.
		    workspaceFolders
	);
	hasDiagnosticRelatedInformationCapability = !!(
		capabilities.textDocument &&
		capabilities.textDocument.publishDiagnostics &&
		capabilities.textDocument.publishDiagnostics.relatedInformation
	);

	const result: ls.InitializeResult = {
		capabilities: {
			textDocumentSync: ls.TextDocumentSyncKind.Incremental,
		}
	};
	if (hasWorkspaceFolderCapability) {
		result.capabilities.workspace = {
			workspaceFolders: {
				supported: true
			}
```

```
                };
        }
        return result;
});

connection.onInitialized(() => {
        if (hasConfigurationCapability) {
                // Register for all configuration changes.
                connection.client.register(ls.DidChangeConfigurationNotification.
                    type, undefined);
        }
        if (hasWorkspaceFolderCapability) {
                connection.workspace.onDidChangeWorkspaceFolders(_event => {
                        connection.console.log('Workspace folder change event
                            received.');
                });
        }
});

// The example settings
interface ExtensionSettings {
        maxNumberOfProblems: number;
        snailPath: string;
}

// The global settings, used when the 'workspace/configuration' request is not
    supported by the client.
// Please note that this is not the case when using this server with the client
    provided in this example
// but could happen with other clients.
const defaultSettings: ExtensionSettings = { maxNumberOfProblems: 1000, snailPath:
     "snail" };
let globalSettings: ExtensionSettings = defaultSettings;

// Cache the settings of all open documents
const documentSettings: Map<string, Thenable<ExtensionSettings>> = new Map();

connection.onDidChangeConfiguration(change => {
        if (hasConfigurationCapability) {
                // Reset all cached document settings
                documentSettings.clear();
        } else {
                globalSettings = <ExtensionSettings>(
                        (change.settings.snailLanguageServer || defaultSettings)
                );
        }

        // Revalidate all open text documents
        documents.all().forEach(validateTextDocument);
});

function getDocumentSettings(resource: string): Thenable<ExtensionSettings> {
        if (!hasConfigurationCapability) {
                return Promise.resolve(globalSettings);
```

```
99          }
100         let result = documentSettings.get(resource);
101         if (!result) {
102                 result = connection.workspace.getConfiguration({
103                         scopeUri: resource,
104                         section: 'snailLanguageServer'
105                 });
106                 documentSettings.set(resource, result);
107         }
108
109         return result;
110 }
111
112 // Only keep settings for open documents
113 documents.onDidClose(e => {
114         documentSettings.delete(e.document.uri);
115 });
116
117 // The content of a text document has changed. This event is emitted
118 // when the text document first opened or when its content has changed.
119 documents.onDidChangeContent(change => {
120         validateTextDocument(change.document);
121 });
122
123 async function validateTextDocument(textDocument: TextDocument): Promise<void> {
124         // TODO is there another way to do it?
125         // In this simple example we get the settings for every validate run.
126         const settings = await getDocumentSettings(textDocument.uri);
127
128         // get path to snail from extension settings
129         const snailPath = settings.snailPath;
130
131         // run the current snail file and return error messages
132         const text: string = textDocument.getText().replace(/\n/gm, "\n");
133
134         // create temp dir and temp file
135         const osTmpDir : string = os.tmpdir();
136         const tmpDir: string = f.mkdtempSync(path.join(osTmpDir));
137         const filename: string = path.join(tmpDir, 'tmp.sl');
138         f.writeFileSync(filename, text);
139
140         try {
141
142                 const diagnostics: ls.Diagnostic[] = [];
143
144                 // run the snail file
145                 const snailPath = settings.snailPath;
146                 const child = cp.spawnSync( snailPath, ['-s', filename]);
147                 const err_msg = child.stdout.toString();
148
149                 // extract error information:
150                 const err_json = JSON.parse(err_msg);
151
152                 let problems = 0;
```

```
                if (err_json.status == 'ERROR' && problems < settings.
                    maxNumberOfProblems) {
                        problems++;
                        const err_start = err_json.location.offset_start
                        const err_end = err_json.location.offset_end
                        const diagnostic: ls.Diagnostic = {
                                severity: ls.DiagnosticSeverity.Error,
                                range: {
                                        start: textDocument.positionAt(err_start),
                                        end: textDocument.positionAt(err_end)
                                },
                                message: err_json.message,
                                source: "Snail " + err_json.type
                        };
                        diagnostics.push(diagnostic);
                }

                // Send the computed diagnostics to VSCode.
                connection.sendDiagnostics({ uri: textDocument.uri, diagnostics })
                    ;
        } catch (e) {
                throw e;
        } finally {
                // remove our temporary directory
                f.rmSync(tmpDir, { recursive: true, force: true });
        }

}

connection.onDidChangeWatchedFiles(_change => {
        // Monitored files have change in VSCode
        connection.console.log('We received an file change event');
});

// Make the text document manager listen on the connection
// for open, change and close text document events
documents.listen(connection);

// Listen on the connection
connection.listen();
```

Appendix G Snail Language Support debugAdapter.ts

The following is the **debugAdapter.ts** file from Snail Language Support.

```
// this is EXTERNAL to vscode
// i.e. we do not have access to vscode apis, because this
// is running separately in node

import * as f from 'fs';
import * as s from 'net';
import * as path from 'path';

// FIXME debugging
const base = path.join(__dirname, "../../");
const response_file = `${base}/stderr.txt`;
const sent_file = `${base}/stdin.txt`;
f.writeFileSync(response_file, 'Debug Output\n');
f.writeFileSync(sent_file, 'Debug Input\n');


const PORT_NUM = 9999;
// start our socket client
var client = s.connect(PORT_NUM, 'localhost', () => {
    f.appendFileSync(response_file, "debugAdapter connected\n")
});

client.on('data', (buff) => {
    const content : String = buff.toString('utf-8');
    console.log(content);
    f.appendFileSync(response_file, content.toString() + "\n");
})

client.on('error', (err) => {
    f.appendFileSync(response_file, "Error!\n");
    f.appendFileSync(response_file, err.toString() + "\n");
})


// register input from vscode
process.stdin.on('data', (buff) => {
    const content : String = buff.toString('utf-8');
    client.write(content.toString());
    f.appendFileSync(sent_file, content.toString() + "\n");
})
```

Appendix H Snail Language snaildap.re

The following is the **snaildap.re** file from the snail language interpreter.

```
// learning how to use the debug adapter protocol (dap)

open Lwt_unix;

open SnailLib;
open Debug_protocol_snail;
open Execute;
open Settings;

let launch = (rpc, launch_args) => {
  let open Launch_command.Arguments;
  input_file := launch_args.program;
  let%lwt _ = execute();
  let%lwt _ = Debug_rpc.send_event(rpc, (module Terminated_event),
      Terminated_event.Payload.make());
  Debug_rpc.send_event(rpc, (module Exited_event), Exited_event.Payload.make(0));
};

let uninitialized = (rpc) => {
  let (promise, resolver) = Lwt.task(());
  let prevent_renter = () => Debug_rpc.remove_command_handler(rpc, (module
      Initialize_command));
  Debug_rpc.set_command_handler(rpc, (module Initialize_command), (args) => {
    prevent_renter();
    // send VSCode the debugging capabilities that
    // this debugger supports
    let caps = Capabilities.make(
      ~supports_terminate_request=(Some(true)),
      ~supports_configuration_done_request=(Some(true)),
      ());
    Lwt.wakeup_later(resolver, (args, caps));
    Lwt.return(caps);
  });
  promise;
};

let initialized = (rpc, init_args, capabilities) => {
  let (promise, resolver) = Lwt.task();
  let prevent_renter = () => {
    Debug_rpc.remove_command_handler(rpc, (module Launch_command));
    Debug_rpc.remove_command_handler(rpc, (module Attach_command));
  }
  Debug_rpc.set_command_handler(rpc, (module Launch_command),
    (launch_args) => {
      open Launch_command.Arguments;
      prevent_renter();
      let%lwt _ = Debug.send_console_event("Launching: %s\n", launch_args.program)
          ;
      let launched = launch(rpc, launch_args);

      // FIXME how do we use this wakeup later?
```

```
      Lwt.wakeup_later(resolver, (launch_args, launched));
      Lwt.return_unit
    });
  Debug_rpc.set_command_handler(rpc, (module Attach_command),
    // FIXME configure to just launch and then attach instead?
    (attach_args) => {
      prevent_renter();
      Lwt.fail_with("The attach command is not supported");
    });
  Debug_rpc.set_command_handler(rpc, (module Disconnect_command),
    (_) => {
      Debug_rpc.remove_command_handler(rpc, (module Disconnect_command));
      Lwt.wakeup_later_exn(resolver, Exit);
      Lwt.return_unit;
    });

  promise;
};

// this is where we register our debugging capabilities
let debug = (rpc, init_args, launch_args, dbg) => {
  let (promise, resolver) = Lwt.task();

  Lwt.pause();

  let send_initialized_event = () => {
    Debug_rpc.send_event(rpc, (module Initialized_event), ());
  }

  Debug_rpc.set_command_handler(rpc, (module Configuration_done_command), (_) => {
    // send a "stopped event"
    // Debug_rpc.send_event(rpc, (module Stopped_event),
    //   Stopped_event.Payload.make(~reason=Entry, ~thread_id=Some(0), ()));
    Lwt.return_unit;
  });

  // provide access to the threads
  Debug_rpc.set_command_handler(rpc, (module Threads_command), (_) => {
    let main_thread = Thread.make(0, "main");
    Lwt.return(Threads_command.Result.make(~threads=[main_thread], ()));
  });

  Debug_rpc.set_command_handler(rpc, (module Terminate_command), (_) => {
    Debug_rpc.remove_command_handler(rpc, (module Terminate_command));
    // FIXME stop the execution
    let%lwt _ = Debug.send_console_event("Terminate request\n");
    // Send back an event indicating that we have terminated
    Debug_rpc.send_event(rpc, (module Terminated_event),
      Terminated_event.Payload.make(~restart=None, ()));
  });

  Debug_rpc.set_command_handler(rpc, (module Disconnect_command), (_) => {
    Debug_rpc.remove_command_handler(rpc, (module Disconnect_command));
    Lwt.wakeup_later_exn(resolver, Exit);
```

```
    Lwt.return_unit;
  })

  // wait for one of these items to occur
  Lwt.join([send_initialized_event(), promise]);
};

let debugger = (in_, out) => {
  // create a dap conneciton
  let rpc = Debug_rpc.create(~in_, ~out, ());
  let cancel = ref(() => ());

  Debug.set_debugger_active(true);
  Debug.set_debug_adapter(rpc);

  Lwt.async(() => {
    try%lwt {
      let%lwt _ = Debug.send_console_event("state uninitialized\n");
      let%lwt (init_args, capabilities) = uninitialized(rpc);
      let%lwt _ = Debug.send_console_event("state initialized\n");
      let%lwt (launch_args, dbg) = initialized(rpc, init_args, capabilities);
      let%lwt _ = Debug.send_console_event("state debug\n");
      let%lwt _ = debug(rpc, init_args, launch_args, dbg);
      fst(Lwt.task());
    } {
      | Exit => Lwt.return_unit
    };
    cancel^();
    Lwt.return_unit;
  });

  // after we initialize our debug session and debug
  // capabilities, start debugging
  let loop = Debug_rpc.start(rpc);
  cancel := () => Lwt.cancel(loop);

  // our issue is that this loop will block until input
  // (communication from vscode (through debugadapter)) closes
  // but we need to send vscode (through debugadapter) some responses
  try%lwt (loop) {
    | Lwt.Canceled => Lwt.return_unit
  }
};

let on_connection = (~client, in_, out) => {
  // let%lwt _ = Debug.send_console_event("Client %s connected\n", client);

  let%lwt _ = debugger(in_, out);


  // let%lwt _ = Debug.send_console_event("Client %s disconnected\n", client);
  Lwt.return();
};
```

```
let serve = (port) => {
  let addr = Unix.ADDR_INET(Unix.inet_addr_loopback, port);
  // open a tcp server for communication between snail and debug adapter
  let%lwt _ = Lwt_io.establish_server_with_client_address(addr,
        (addr, (in_chan, out_chan)) => {
          let cl = Unix.string_of_inet_addr(Unix.inet_addr_loopback);
          on_connection(~client=cl, in_chan, out_chan);
      });

    let%lwt _ = Lwt_io.printf("Debug adapter server listening at port %d\n", port)
        ;
    fst (Lwt.wait ())
};

Lwt_main.run(serve(9999));

//this line won't be reached
```

Appendix I Future snaildap Debug Adapter Protocol Requests

The **`snaildap`** debug adapter module allows a VS Code client to start and stop a snail debugging session through the VS Code UI. In order to perform debugging-specific actions such as step in/out behavior and variable inspection, a number of DAP requests must be processed and responded to in the **`snaildap`** module. Those requests are listed here, organized by the functionality they are meant to support. They are described in more detail on the Debug Adapter Protocol documentation: https://microsoft.github.io/debug-adapter-protocol/specification

1. Breakpoints
    - BreakpointLocations Request
    - SetBreakpoints Request
2. Program execution start/stopping
    - Pause Request
    - Continue Request
3. Step through
    - Next Request
    - StepInTargets Request
    - StepIn Request
    - StepOut Request
4. Call stack display
    - StackTrace Request
5. Variable/expression inspection
    - Scopes Request
    - Variables Request
    - Evaluate Request