SAND2019-10279R

Sandia National Laboratories

24 January 2019

Snippet Open-Source Specification

Use and Application

Brian Rigdon, 05834
R&D S&E, Cybersecurity

Prepared by

Sandia National Laboratories
Albuquerque, New Mexico 87185

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly-owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Table of Contents

List of Figures

List of Tables

Snippet Open-Source Specification

1 Introduction

Snippet is a collaborative software code auditing tool developed at Sandia National Laboratories (Sandia) in conjunction with several sponsoring government agencies to enable software analysts, individually or in groups, to non-destructively audit large codebases, share files and comments from those audits, and develop Snippet plugins for use with other text editing software applications.

This specification defines the requirements and the reasoning behind the requirements for tools developed to be compatible with the technology and purpose of Snippet. A condensation of the requirements is given in the companion document, "Snippet Open-Source Specification Synopsis" also generated by Sandia National Laboratories.

Snippet's approach to browsing through codebases is similar to a web browser navigating the internet. Snippet software provides a code auditing environment with the following features:

- Project Definition – defines the audited source codebase and auditing team
- Annotation – analysts place annotations in line with the code to specifically locate their comments about the software being audited
  - Non-destructive – annotations do not corrupt the original file
  - Collaborative – annotations are shared among the audit team
  - Searchable – keywords within annotations are identified with a hashtag (#) to facilitate finding annotations made with the Snippet file
- Browsing Capability – an easy-to-use interface for reading, making comments to, and searching codebases
  - Searching – locates keywords within a file or the entire codebase
  - Result Sets – formalized search results can be saved and loaded, and enables other tools to provide search results that Snippet can ingest
  - Navigation – easily moves between locations within the codebase
- Analysis Result Handling
  - Code-level static analysis support via Result Sets
  - Plugin interface for extensibility
- Cartographer – a graphical navigation extension to Snippet providing a *visual map* of browser activity

As Snippet grows in complexity and popularity, it has become evident that the care and maintenance of Snippet is beyond the scope of a national laboratory. Many of the features in Snippet were developed based on either web browsers or Interactive Development Environments (IDE). Given the advanced state of most IDEs, it is not feasible for Snippet developers to remain up-to-date with continuous updates and demands for new features. However, the auditing concepts of Snippet and its plugins can be implemented for commercial-off-the-shelf (COTS) IDEs to provide most Snippet functionality within code analysts' existing development environments. This provides analysts with the familiarity and power of their COTS IDE with the benefits of Snippet's features.

Snippet's developers have determined that the best path forward for the lifecycle of Snippet is to develop an open standard that defines Snippet functionality. The Snippet executable is also being open-sourced as a reference implementation of Snippet functionality. Thus, Snippet is now both a tool and an open standard. Any software tool that complies with Snippet's standard is usable by analysts to collaborate with any other analyst using Snippet or another Snippet-compatible tool.

In contrast to most IDEs or text editors used for code auditing, Snippet is designed as an auditing tool. While current IDEs and editors display and navigate through code, those applications primarily generate code with priorities very different than those required for auditing. This reflects the difference between software development and software auditing. For example:

Software Development:	Software Auditing:
• Create, change files as needed	• Examine and explore files as is, modification is undesirable
• Keywords created to define new objects, structures, interfaces, etc.	• Keywords need to be understood regarding what they represent and where they are used
• Navigation is a secondary function of an editing environment	• Navigation is a primary function of auditing
• Comments are added to describe functionality	• Annotations do the same, but also capture concepts related to the purpose of the audit

Therefore, if a Snippet-compatible extension of an IDE is to be built, we strongly recommend that the plugin adopt not only the annotations of Snippet, but the auditing-specific characteristics of Snippet—that it meet the Snippet open standard.

To truly grasp what Snippet does comes from understanding how it enables auditing. Many actions built into Snippet have no associated data, nevertheless, they greatly simplify code auditing tasks (e.g., double-click navigation). Other actions have data artifacts that must be standardized to ensure cross-tool collaboration (e.g., searches resulting in Result Sets). The Snippet-standard features currently specified in the Snippet application are as follows:

- Project Definition – the directory that contains the source to be audited (optionally, the version and team members working the project, as well)
- Annotation – allows user-generated notes to be associated with code and displayed accordingly
- Navigation and Searching – the means to logically move within a multi-file/multi-directory codebase
- Result Sets – an artifact of searches (whether performed by Snippet or another tool)
- Migration – applies annotations to different versions of the audited codebase
- Customization – makes it easier for the auditor to modify the interface to Snippet
- Plugins – extends Snippet behavior to enable customization and external tool support

Cartographer (Snippet's graph-based navigation aid) is included with the open-source Snippet codebase; however, Cartographer is not a core Snippet capability and not part of the Snippet standard at the time of this writing.

Snippet compatibility will assume that Snippet and compatible tools provide a graphic user interface (GUI) that allows source code to be read, which is the definition of code auditing. All features in Snippet are focused around a GUI pane that displays the code to the auditor.

2 ANNOTATION

Snippet enables an auditor to make notes on the software being audited. Each note corresponds to a specific line in a specific file and represents the auditor's impressions, questions, remarks or suggestions during code review of that line or section of code. An annotation comprises the text of the note with relevant metadata to identify where the note belongs and who wrote it. To protect the integrity of the source code, annotations are stored in an SQL database that is independent of the source code and its directory hierarchy. A RESTful API has been developed by one of Sandia's collaborators to interface to the annotation database. This RESTful API uses JSON as the data transfer format, and its description serves as the data specification for Snippet's operation. Interfacing with a database is an absolute requirement for a Snippet-based application to be Snippet-standard compliant. This section presents the JSON representation and describes the RESTful API that Snippet supports.

2.1 ANNOTATION SPECIFICATION

Management of Snippet's annotations is one of two mandatory requirements for Snippet compatibility; the other being Result Set specification. Managing annotations includes creating, editing, storing, and deleting annotations as well as displaying them to the code auditor. The way Snippet-compatible applications use the data and present reviewer annotations is implementation dependent. Most of this document describes recommended application behaviors and what we've determined is useful for code auditors in our community. Any application must adhere to this JSON description to be Snippet-standard compliant. The actual Snippet JSON schema is provided in the appendices.

Table 1. Snippet Annotation Defined

Field	Data Type	Description
id	string	Database table index of this note (automatically generated)
filename	string	Relative path and filename of the file being annotated
linenum	integer	Line number (indexed from 1) associated with the note
txt	string	The notes' text
author	string	User id of the originating author
timestamp	string	Timestamp of the most recent edit of this note
ctimestamp	string	Creation timestamp of this note
tags	array of strings	A list of tags in the note
version	string	Used to associate this note with a particular revision of the code (e.g., git repository) (currently unused in Snippet reference implementation)

When an annotation is Created, it does not have an index; that is automatically inserted by the database. Subsequent access for that annotation, whether reads, updates, or deletions, will have the index specified. Snippet's behavior modifies the timestamps when the annotation is Created or Edited. If a Snippet-compliant tool is written to use the Database Server, that Database Server will manage timestamps automatically.

2.2 Annotation Database Interaction

As mentioned above, one of Sandia's Snippet development partners has implemented a Snippet-compatible SQL database web server. This *Annotation Server* allows auditors to use Snippet-enabled editors or IDEs to collaborate with each other's annotations. To support this database, a RESTful API was defined that greatly simplifies database compatibility and integrity. Third party tools interface more easily through this API rather than directly to Snippet's database. Multiple projects can utilize the same annotation service by sharing a common database.

Table 2. Snippet's Extended Use of This RESTful API That Leverages the Annotation Service

Category	REST type	URL ({inputs})	Returns
Administration	GET	`/api/v0.2/list_projects`	List of project names
	PUT	`/api/v0.2/{new_project}/{version}/create_project`	N/A
	DELETE	`/api/v0.2/{project}`	N/A
Annotations	GET	`/api/v0.2/{project}/{version}/get_notes`	List of annotation data for all files in the project
	GET	`/api/v0.2/{project}/count_notes`	Integer number of notes in the project
	GET	`/api/v0.2/{project}/{version}/get_files_with_notes`	List of filenames which have annotations
	GET	`/api/v0.2/{project}/{version}/get_notes_for_file/{filename}`	List of annotations associated with the specified file
	POST	`/api/v0.2/{project}/{version}/add_note`	N/A
	POST	`/api/v0.2/{project}/{version}/add_notes`	N/A
	PUT	`/api/v0.2/{project}/update_note/{idx}`	N/A
	DELETE	`/api/v0.2/{project}/delete_note/{idx}`	N/A
	GET	`/api/v0.2/{project}/{version}/stream`	Initiates a stream from the server with updates to the project

Table 3. Remainder of the Annotation Server API (Not used by Snippet)			
Category	**REST type**	**URL ({inputs})**	**Returns**
Access Control	GET	`/api/v0.2/{project}/get_auth_required`	True if authentication is required for this project
	PUT	`/api/v0.2/{new_project}/set_auth_required`	N/A
	PUT	`/api/v0.2/{project}/set_auth_not_required`	N/A
	GET	`/api/v0.2/{project}/list_members`	List of user ids
	GET	`/api/v0.2/{project}/list_accessors`	List of user ids who attempted unauthorized access
	PUT	`/api/v0.2/{new_project}/add_member/{username}`	N/A
	PUT	`/api/v0.2/{project}/add_member/{username}/{organization}`	N/A
	DELETE	`/api/v0.2/{project}/remove_member/{username}`	N/A
	DELETE	`/api/v0.2/{project}/remove_member/{username}/{organization}`	N/A
Versioning of Annotations	PUT	`/api/v0.2/{project}/send_diff//{commit_a}/{commit_b}`	N/A
	GET	`/api/v0.2/{project}/list_versions`	List of version strings
	GET	`/api/v0.2/{project}/{version}/list_missing_diffs`	Differences missing from the server for a given version of the project code
	GET	`/api/v0.2/{project}/list_missing_diffs`	Differences missing from the server for the project code

2.3 Annotation Behaviors

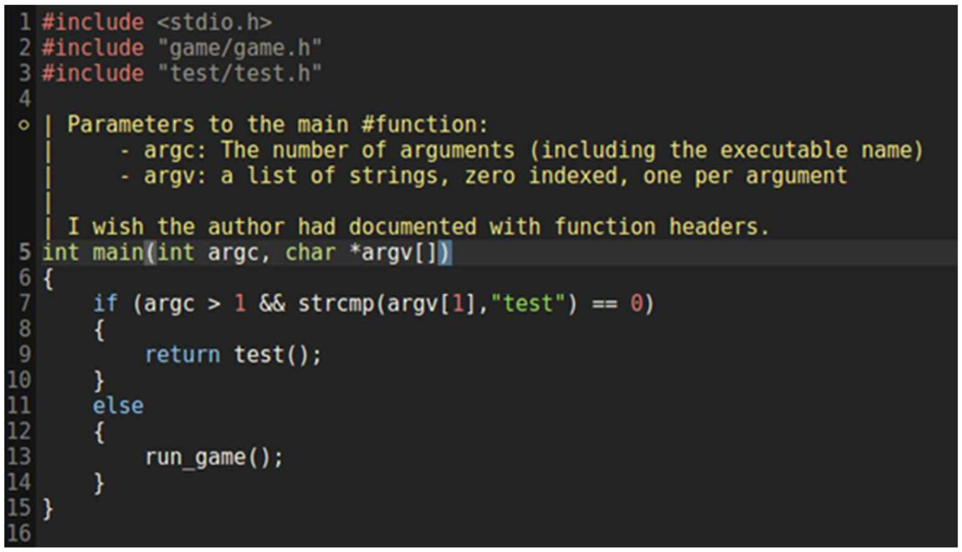
Snippet's key feature is the ability to display annotations in line with the code with the annotation presented in such a way as to differentiate it from the source.

Any tool compatible with the Snippet2.4 standard specification must have a mechanism for displaying annotations. Ideally, this mechanism would display annotations in line, as Snippet does, but anything that automatically or readily brings annotations into view is acceptable.

> *Note: A quirk of the QT text display widget is that Snippet's annotations cannot be placed ahead of the first line of code. Other tools have issues with the last line. These implementation nuances do not affect compliance with the Snippet-standard specification.*

2.4.1 Annotation Display

When an annotation is placed in the code, it is inserted into the text above the line it describes. The annotations are slightly indented, uniquely colored, and prepended with a vertical bar character to distinguish the annotation from the code. The code line numbers are interrupted to keep code lines enumerated correctly. A screen capture of a Snippet annotation is given in figure 1, which displays a document with multiple users' annotations.



```
#include <stdio.h>
#include "game/game.h"
#include "test/test.h"

| Parameters to the main #function:
|     - argc: The number of arguments (including the executable name)
|     - argv: a list of strings, zero indexed, one per argument
|
| I wish the author had documented with function headers.
int main(int argc, char *argv[])
{
    if (argc > 1 && strcmp(argv[1],"test") == 0)
    {
        return test();
    }
    else
    {
        run_game();
    }
}

```

Figure 1. Snippet Annotation Display

2.4.2 Annotation Creation and Editing

The auditor creates an annotation in a variety of ways: (1) via drop-down menu, (2) right-click menu on the line to be annotated, or (3) using the ';' ('semicolon') key binding shortcut. Snippet makes the process of adding annotations very simple and straightforward for code annotation to become almost reflexive to the auditor. When annotations are created/edited, reviewers can do so in line or via a pop-up window per user preference. Saving and closing annotations is accomplished using the GUI or a 'shift-enter' key binding. Annotations can likewise be deleted through menu operations.

> *Note: The annotation key binding is a prime example of the difference between Snippet as a pure auditing tool and either an IDE or editor. The semicolon (';') key was chosen as it is very easy to use (under the right pinky finger of a touch typist) and because it has the same function in a binary auditing tool that is popular in our community. All our auditors use that tool and are accustomed to using the semicolon for this purpose. In a code editor, the semicolon simply inserts a ';'.*

2.4.3 Tags in Annotations

The annotation shown in figure 1 has a Snippet tag defined in it. These tags become keywords for Snippet annotations. In the first line of the annotation, the word *function* is prepended with a hashtag symbol ('#'). This identifies it as a tag for special operations in Snippet. Snippet uses these tags to search for all the instances of this tag. The auditor uses tags to associate thoughts or identify common code features and to find those references more easily when needed.

The choice of keyword is determined by the auditor or team. Any word or sequence of alphanumeric characters prepended with the hashtag is a valid tag.

2.4.4 Annotation Search and Filtering

Snippet provides an optional *Annotation Info* GUI pane to display all the tags each author has used. The Annotation Info pane presents the tags and users with checkboxes. By selecting or deselecting the tag, all annotations with that tag will be either displayed or hidden. Likewise selecting or deselecting the user will display or hide all that user's annotations.

2.4.5 Code Integrity

It is impossible to modify the underlying codebase while auditing with Snippet; there is no mechanism whereby the code can be modified. Snippet's display panes do not allow any modification of their contents. The annotations are edited separately (via either dialog boxes or special in-line editors) from their display window. Thus, Snippet protects the integrity of the code as it interacts on top of the code's display and not directly with the code. This is a critically important distinction between Snippet and a typical IDE. If an IDE is to be successfully extended to Snippet-standard compatibility, the extension should incorporate Snippet's approach that prevents modifying files being audited. If possible, the IDE edit widget should be disabled from changing the displayed code text.

3 CODE BROWSING

Snippet's primary function is to browse through code during auditing. Written to be a browser rather than an editor/IDE, Snippet's expectations for user interaction are significantly different than most software editing tools. Nevertheless, many intuitive, common key bindings and shortcuts are mapped directly between editor/IDEs and Snippet. Snippet's key bindings are also customizable, enabling users to align their Snippet experience with their own preferences (See *Customization* below).

Navigation is an intrinsic function of editors and IDEs, and, as such, users have strong opinions as to what constitutes *appropriate* navigation. Snippet's approach and style of navigation is, therefore, an optional but *recommended* component of the Snippet-standard's specifications.

As Snippet's intention is to facilitate the auditor's understanding of a codebase, navigation is optimized to explore the *symbols* present in a program. By *symbols*, we mean constructs defined by the code author: types, variables, functions, include files, etc. Snippet provides many ways to find uses and definitions of symbols, giving each user individualized options to best perform her audit:

- Double-click – jumps from a symbol definition to a use, or from a use to a definition
- Right-click – menu options including finding references and opening in new tabs or splits
- Searching – string-based text-matching within a file
- Outline View – a GUI section that displays structs, typedefs, constants, functions, etc. – selecting any of these symbols will navigate to the definition in the current file
- Omnibar – a GUI element that selectively searches based on user input – selecting from the Omnibar navigates the user to files, typedefs, functions, etc.
- Go-to Line – as one would expect

- Result Set Navigation – Result Sets are presented to the user – selecting a result will navigate the open browser window to display the file/line specified in the result

If there is only a single result to a navigation query, Snippet will jump directly to that location. Navigation results satisfied by multiple locations (e.g., the uses of a typedef or calls to printf) are presented to the user as a Result Set. Result Sets are an intrinsic part of Snippet's navigation. Because Result Sets are a special case of navigation and they form a mandatory requirement of the Snippet-standard specification, they are described in their own section below.

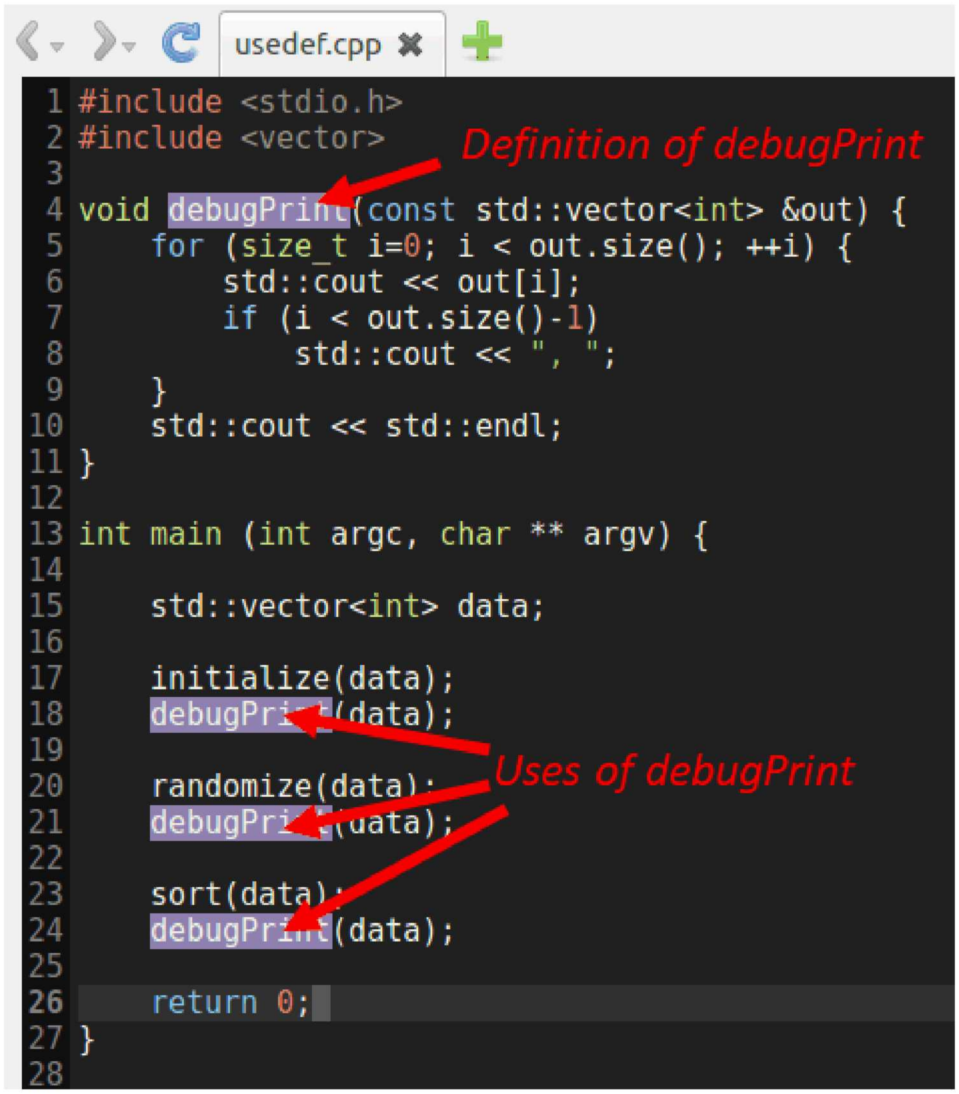
Snippet navigation is based on a cross-reference database built using Snippet's underlying *context* utility ("ctx"). Cross-references are logical connections between labels in the code, such as a variable declaration and all its uses. If the Snippet utility is being mapped to another tool, it is assumed that tool has cross-referencing capabilities to be used for navigation.

3.1 Double-click Navigation

Double-click navigation is probably the most significant difference between Snippet and a code editor/IDE. While an IDE interprets a double-click as the selection of a word in the text, Snippet responds to that double-click by performing cross-reference-based navigation on the codebase for the symbol being selected.

When the user double-clicks a cross-referenced symbol (figure 2), she is navigating *away* from that instance of the symbol. The navigation *to* is dependent on what she navigates *from*. If we are navigating from a use of a symbol (e.g., a function call or the type of a variable declaration), the navigation destination will be the definition of that symbol (the function declaration or typedef in our example). If the user double-clicks a definition, the navigation will be toward the use of that symbol (figure 2). If there are more than one potential destination, a Result Set will be generated and presented to the user.

> *Note: Snippet can map double-click navigation to* text searching *rather than* cross-reference searching*, but that is not a typical use of double-click.*



```cpp
#include <stdio.h>
#include <vector>

void debugPrint(const std::vector<int> &out) {
    for (size_t i=0; i < out.size(); ++i) {
        std::cout << out[i];
        if (i < out.size()-1)
            std::cout << ", ";
    }
    std::cout << std::endl;
}

int main (int argc, char ** argv) {

    std::vector<int> data;

    initialize(data);
    debugPrint(data);

    randomize(data);
    debugPrint(data);

    sort(data);
    debugPrint(data);

    return 0;
}

```

Figure 2. Sample Cross-referenced Symbol

3.2 Right-click Navigation

When users right-click a symbol, Snippet presents a menu of potential actions that can be taken from that point in the file with the following options pertaining to navigation.

- Find declaration in this file – finds the first definition matching the symbol name in this file (Note: This ignores scope: “int i;” may be declared multiple times but only one will be found)
- Find declaration in this function – finds the definition of the symbol in this function
- Find definition – jumps to the definition of the symbol (similar to double-click navigation)
- Find references – jumps to the use of the symbol (if only one) or presents a Result Set of all uses (similar to double-click navigation)

3.3 Searching

Many software and text editors provide quick search capabilities by typing a *ctrl-f*. Users of *vi*, on the other hand, use the */* or *?* keys. Snippet supports the capability to use these commands when searching through a file being audited (See *Customization* on how to change key bindings).

Once the string is found, pressing *n* or *N* will navigate the cursor to the next location of the string. The *n* key takes the auditor in the direction of the search; *N* takes the auditor in the opposite direction.

As Snippet navigates to the next search term, instances of that term are highlighted. If desired, the user may change the color highlighting the search term.

3.4 Outline View Navigation

Snippet has an *Outline View*—a GUI pane that displays a high-level view of the file being audited. Features such as functions, macros, typedefs, variables, etc. are alphabetically organized by category in the Outline View. Double-clicking an entry in the outline will navigate the cursor to the definition of that keyword.

3.5 Omnibar Navigation

At the top of the Snippet window is the *Omnibar*. This is a type-in widget that quickly finds matching files, functions, classes, etc. for the text as it is being entered. The auditor can quickly find all matches to her search, refining the search as she types. The Omnibar behavior is modeled after the URL entry bar of most web browsers. As search information is typed, these browsers provide recommendations based on what has been entered so far.

3.6 Go to Line

Like most IDEs and editors, Snippet provides a mechanism to jump to a specific line. By default, this is the *ctrl-g* key binding. A dialog box opens that prompts the auditor for the desired line. Upon entry, the cursor navigates to that line and the view is adjusted accordingly.

4 Result Sets

Auditing is one of many methods for analyzing software. Numerous automated tools provide different approaches to inspecting software and extracting features. Snippet can integrate these automated tools if their search results can be extracted and re-composed for use by Snippet's navigation tools to identify locations within the codebase.

Snippet navigates by identifying locations within a file; specifically, using the filename (including path), line number, and column number (optional) for the requested location. Result Sets of requested locations are typically generated from search results; however, sets of locations can be generated by other analysis tools. Externally generated Result Sets can be passed to Snippet through a plugin or read in from a file. Snippet displays Result Sets to the auditor in a table format. If the auditor double-clicks on a result within the Result Set, Snippet navigates to that result's location.

4.1 Result Set Formatting

Snippet internally generates Result Sets based on information provided by the ctx index database. Depending on the tool used to generate the index (CScope, Ctags, etc.), Snippet builds a data type comprising the important location information. This data is presented to the auditor using a GUI element.

For externally generated Result Sets, Snippet supports two representations of data: *hierarchical* and *notes* schemas. Specification schema for these result representations are given in Appendix B.

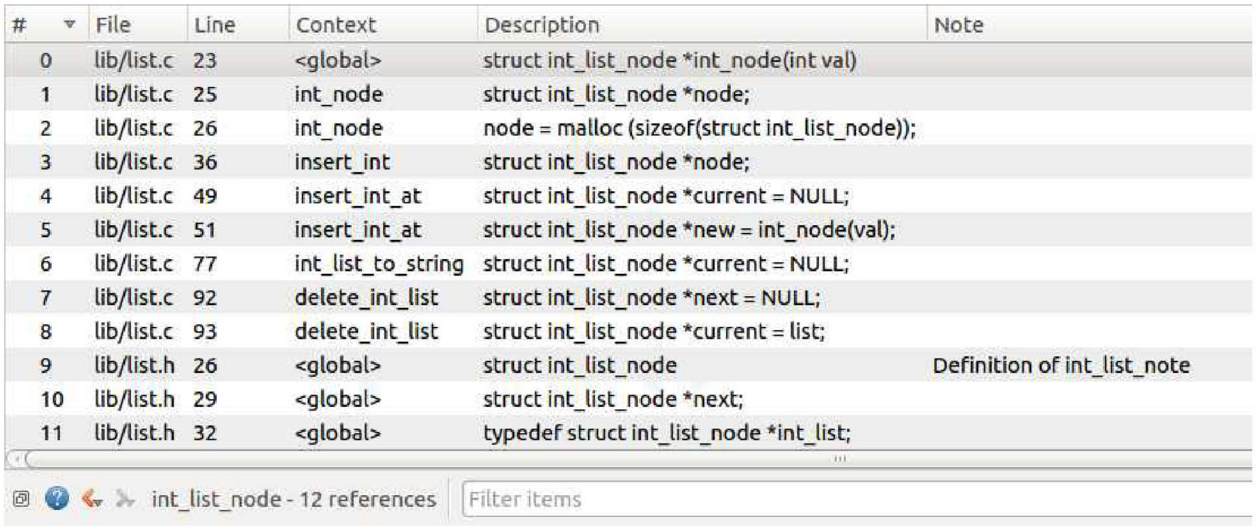
> *Note: There are two other legacy schemas for which Result Sets can be parsed, but these are essentially deprecated.*

After Snippet was developed, an industry standard was generated to represent the results of source code static analysis: the Static Analysis Result Interchange Format (SARIF (https://gitub.com/sarif-standard)). Data represented in SARIF is a superset of data that Snippet needs. Therefore, a Snippet auditor importing a SARIF file would likely parse out the file and location information in the Result Set to display the data. At the time of this writing, Snippet does not ingest SARIF files.

Any tool that is Snippet-standard compliant must be able to generate and absorb Result Set data. Such tools will be able to export Result Sets in a format that can be loaded by other Snippet-compliant tools. We recommend the SARIF format be adopted for loading and storing Result Set information.

4.2 Result Set Display

Result Sets are presented to the auditor in table format as seen in in figure 3.



#	File	Line	Context	Description	Note
0	lib/list.c	23	<global>	struct int_list_node *int_node(int val)	
1	lib/list.c	25	int_node	struct int_list_node *node;	
2	lib/list.c	26	int_node	node = malloc (sizeof(struct int_list_node));	
3	lib/list.c	36	insert_int	struct int_list_node *node;	
4	lib/list.c	49	insert_int_at	struct int_list_node *current = NULL;	
5	lib/list.c	51	insert_int_at	struct int_list_node *new = int_node(val);	
6	lib/list.c	77	int_list_to_string	struct int_list_node *current = NULL;	
7	lib/list.c	92	delete_int_list	struct int_list_node *next = NULL;	
8	lib/list.c	93	delete_int_list	struct int_list_node *current = list;	
9	lib/list.h	26	<global>	struct int_list_node	Definition of int_list_note
10	lib/list.h	29	<global>	struct int_list_node *next;	
11	lib/list.h	32	<global>	typedef struct int_list_node *int_list;	

Figure 3. Result Set Display

Snippet generates the Result Set table with a list of headers using data provided to Snippet. In addition to headers, the data contains a list of items from which table information can be extracted. Each item must, at a minimum, contain the filename and line number to facilitate navigation. As seen in figure 2, one of the optional items displayed is an annotation associated with line 26 of `lib/list.h`.

The GUI widget enables double-click callback on any of the table elements. This is the mechanism whereby Snippet performs navigation. Snippet uses each data listing's table information to navigate to the location for its display.

Snippet-compliant tools do not need to duplicate this display; however, they require some mechanism to display the Result Set information to the auditor. A mechanism to use Result Set data for navigation is also required for successful compliance.

5 Additional Features

5.1 Customization

Snippet has many customizable features to individualize auditing experiences. Snippet's capability to map key bindings to specific operations is one of its foremost features. Key bindings, primarily those enabling navigation, are essential to the auditing experience. Snippet binds certain keys and mouse actions to specific navigation features. These associations are intended to make code browsing as intuitive as possible so that our auditors focus on auditing code, not operating the code auditing software. Snippet is capable of matching key bindings to different editor environments to take advantage of previous familiarity with specific key bindings. For example, if an auditor is accustomed to using Visual Studio, searching is initiated by a *ctrl-f*, whereas in *vi*, searching is initiated with the */* character. Snippet provides key bindings based on several editors and enables users to change any key binding per user preference.

We recommend that any tool developed to the Snippet standard provides customization of key bindings to users.

All Snippet key bindings are listed in Appendix C.

5.2 Plugins

Snippet is extensible through its plugin interface, which essentially gives Snippet users the ability to hook into Snippet internals. Typically, plugin interfaces extend the user interface for new tools or analyses. In Snippet, the plugin interface integrates several independent analysis tools into the Snippet auditing environment.

Snippet developed its plugin environment for a broad spectrum of use in the research arena. Many Snippet users developed custom plugins to facilitate their work. Tools based on the Snippet specification, however, may not need a plugin interface. Annotation capabilities that are being added to IDEs are often implemented as plugins. In these cases, the IDE's plugin interface supersedes that of Snippet.

Plugin capability is not a requirement of the Snippet-standard specification.

5.3 Cartographer

Cartographer is an extension of Snippet that provides a graph-based visualization of the auditor's navigation history. Cartographer exists as a separate pane from the Snippet GUI. The Cartographer GUI displays navigation locations as nodes in a graph. As the user navigates away from their current location in the code, Cartographer inserts a new node with the edge showing the fact that a navigation event happened.

The graph is interactive; selecting a node in the graph will navigate Snippet back to the location represented by that node. Nodes can also be arranged as needed by the user.

Cartographer has been shown to be useful helping auditors with their processes. However, it is not a key capability of Snippet and not a part of the Snippet-standard specification.

6 Snippet Reference Implementation

The most recent version of Snippet is provided as a representative implementation of the concepts presented in this requirements specification. Users are encouraged to install and run Snippet to familiarize themselves with the behaviors described in this document. Snippet’s documentation is also supplied to give supplemental information to this specification.

APPENDIX A - TECHNICAL DESCRIPTION OF ANNOTATIONS

A.1 ANNOTATION JSON

Annotations are stored in a database, but are imported or shared via JSON. The JSON transaction mechanism allows the various annotation implementations to have customizable implementation using a common sharing mechanism.

In addition to the import and export of annotations, the JSON representation of an annotation is the correct data implementation when exercising the Annotation Server's RESTful API.

A.1.1 Annotation JSON Schema

Annotations are supplied to the Annotation Service with JSON conforming to the schema as shown in the text box below.

Annotation JSON Schema

```
{
  "title": "Annotation",
  "type": "object",
  "properties": {
    "id":{ "type":"string" },
    "filename":{ "type":"string", "minLength": 0, "maxLength": 40000 },
    "linenum": { "type": "integer", "minimum": 1, "maximum": 4000000 },
    "txt":    { "type": "string", "minLength": 0, "maxLength": 40000 },
    "author":    { "type": "string", "minLength": 0, "maxLength": 40 },
    "tags": {
      "type": "array",
      "items": { "type": "string", "maxLength": 40 },
      "uniqueItems": True,
      "maxItems": 10
    },
    "timestamp": {
      "type": "string",
      "format": "date-time"
    },
    "ctimestamp": {
      "type": "string",
      "format": "date-time"
    }
  },
  "required": [ "filename", "linenum", "txt", "author", "tags" ]
 }
```

The *id* field found in the text box is automatically generated. When an annotation is created, it is passed to the database without an *id*. The database action of saving the annotation will assign the annotation its *id*. Subsequent operations with this annotation will then contain its *id*.

Additionally, JSON arrays of annotations can be built to handle multi-annotation interactions with the Annotation Service.

Fields that are not specified in this document will be ignored by the current version of the Annotation Service. The Annotation Service, however, may be extended after the time of this writing.

A.1.2 Annotation JSON Example

A truncated JSON export of a Snippet annotation database is below. The JSON is suitable to be imported by Snippet or the Annotation Database.

Annotation JSON Example

```
{
  "data": [
    {
      "author": "user1",
      "ctimestamp": "2018-09-10T10:38:36.834246",
      "filename": "#/main.c",
      "linenum": 23,
      "tags": [
        "release"
      ],
      "timestamp": "2018-09-10T10:39:30.210634",
      "txt": "Primary code - this is what is in the #release version"
    },
    {
      "author": "user1",
      "ctimestamp": "2018-09-11T08:07:04.138652",
      "filename": "#/main.c",
      "linenum": 43,
      "tags": [
        "This",
        "annotation",
        "many",
        "tags",
        "like",
        "not"
      ],
      "timestamp": "2018-09-11T08:07:04.138652",
      "txt": "#This is a #annotation with #many #tags. #like it or #not;"
    },
    {
      "author": "user2",
      "ctimestamp": "2018-09-10T10:35:08.224153",
      "filename": "#/test.c",
      "linenum": 23,
      "tags": [],
      "timestamp": "2018-09-10T10:35:08.224153",
      "txt": "Debug version of get_user_input. It includes all the debug
prints"
    }
  ],
  "schema": {
    "format": "notes",
    "version": 1
  }
 }
```

APPENDIX B - TECHNICAL DESCRIPTION OF RESULT SETS

Snippet reads Result Set information to give the analyst a visual representation of the Result Set. Snippet uses a table display as this visual representation. This table can then be used to navigate to the locations specified in the Result Set.

For a Result Set to be ingested by Snippet or a Snippet-compliant application, it must be supplied as a JSON object. The Result Set is displayed to the user in table format. Snippet currently supports two JSON schemas. The first is a *hierarchical* format that enables source analysis tools to present nested results for Snippet's use. The second format, called a *notes* schema, is based on an export of the Snippet annotation database.

Result Sets are presented as both data and metadata. The metadata describes how Snippet parses and displays the data. Both the data and the metadata are represented differently in the JSON for each schema. Each schema is described below.

SARIF files are not yet supported by Snippet. It is a straightforward effort to ingest a SARIF object, then build a Snippet Result Set object from that information. That task has simply yet to be written.

B.1 HIERARCHICAL RESULT SET SCHEMA

Snippet builds a table out of each result provided in the schema. Each row in the table is a different result from the Result Set. The hierarchical schema provides a mechanism to group the rows. This grouping is described in the "`data`" element description below.

There are four fields in the hierarchical Result Set schema. The list below gives the JSON name strings for each of these fields along with a description of these fields.

- "`schema`" – identifies the hierarchical results schema
- "`table_headers`" – a list of the table headers Snippet will display (the order of this list describes how the data field will be extracted)
- "`fields`" – the JSON names associated with each data member in the "`data`" field (this list must be ordered exactly as the "`table_headers`" field for the data to be correctly extracted and displayed)
- "`data`" – the location information associated with the results

The "`table_headers`", "`fields`", and "`data`" members all work together to build the table. Each data element is defined as a *parent* in its JSON. This is to allow other data to be nested under that data element. The nested data is presented as a list underneath the *children* of that data element. In turn, each child element is defined as a parent, and can have children underneath it as well. This is explained in deeper detail below.

B.1.1 Hierarchical "**`schema`**" Element

The "`schema`" element must appear as follows to identify a hierarchical Result Set:

```
"schema": { "version" : 1, "format", "hierarchical_results"}
```

B.1.2 Hierarchical "**`table_headers`**", "**`fields`**", and "**`data`**" Elements

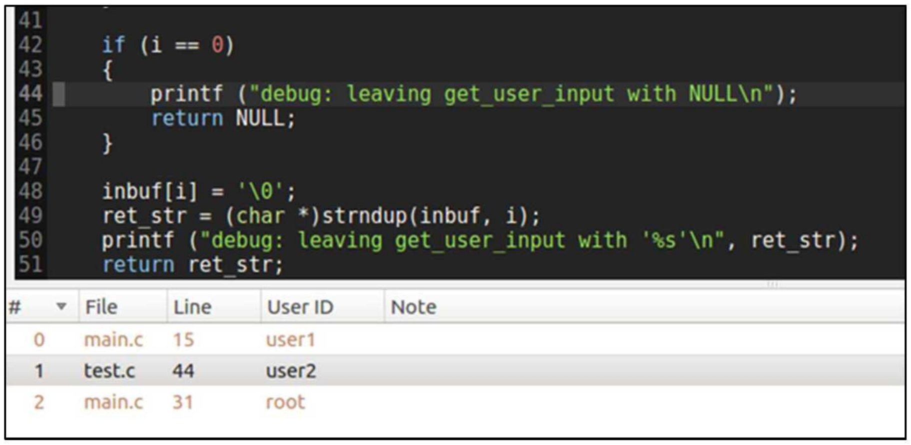
The "`table_headers`" and "`fields`" elements are ordered lists of strings. Their ordering is crucial because headers and fields are matched based on this ordering. The "`table_headers`" provide the headers for each column in the table that is presented to the user. Each string in the table headers corresponds to a string in the "`fields`" element; the first *table header* is associated with the first *field*, the second with the second, etc. Each element in the "`fields`" element is a name for the list of name/value pairs in the "`data`" element. There are no restrictions on the headers or fields displayed by Snippet.

Each element in the "`data`" array **MUST** have a "`parent`" object. The value associated with the "`parent`" name is a JSON element containing the information for this result. The *parent* object **MAY** include data for each of the fields specified in the overarching fields array. If it does not have data for that field, that table entry will be left blank. Additionally, to generate a navigation location for each result, the *parent* object **MUST** include "`filename`" and "`linenum`" fields, whether or not these are included in the "`table_headers`" and "`fields`" arrays. The parent object may also contain an **OPTIONAL** "`colnum`" field to further resolve the location (the meaning and effect of the "`children`" array is explained below). The "`colnum`" field places the Snippet cursor on the specified column if this field is present. If the "`colnum`" field is not present, Snippet places the cursor on the first column (line numbers are indexed from 1; column numbers are indexed from 0).

As an example, the JSON in the text box below will generate the table in figure 6.

Hierarchical Results JSON

```
{
  "schema": {"version":1, "format": "hierarchical_results"},
  "table_headers": ["File", "Line", "User ID", "Column", "Note"],
  "fields": ["filename", "linenum", "author", "colnum", "txt"],
  "data": [
      { "parent": {
          "author": "user1", "filename": "main.c", "linenum": 15} },
      {  "parent": {
          "author": "user2", "filename": "test.c", "linenum": 44} },
      {  "parent": {
          "author": "root", "filename": "main.c", "linenum": 31} }
    ]
 }
```



```
    if (i == 0)
    {
        printf ("debug: leaving get_user_input with NULL\n");
        return NULL;
    }

    inbuf[i] = '\0';
    ret_str = (char *)strndup(inbuf, i);
    printf ("debug: leaving get_user_input with '%s'\n", ret_str);
    return ret_str;
```

#	File	Line	User ID	Note
0	main.c	15	user1	
1	test.c	44	user2	
2	main.c	31	root	

Figure 4. Hierarchical Results Display

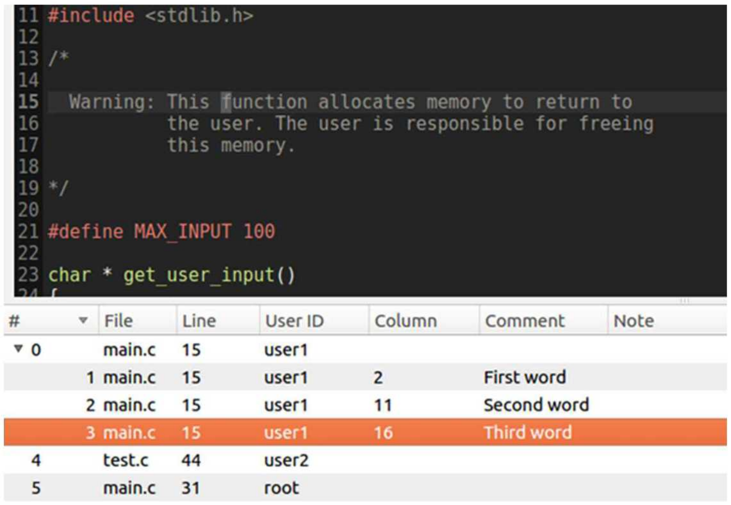
B.1.3 Nested Hierarchical Results

Snippet's results nesting feature is provided so that analysis tools can generate groupings of results. In the examples in figures 7 and 8, we show a grouping based on a line of code. We have expanded the JSON above to include annotations and a grouping.

Grouping is accomplished with the "`children`" array in each data element. The "`children`" element is an array holding more results. These results are identical in form to all the other results. That is, they must have "`parent`", "`filename`", and "`linenum`" fields ("`colnum`" is optional). Additionally, they must support the field names listed in the overall "`fields`" array. The "`children`" array is an extension of a parent object.

Nested Hierarchical Results JSON

```
{
  "schema": {"version":1, "format": "hierarchical_results"},
  "table_headers": ["File", "Line", "User ID", "Column", "Note"],
  "fields": ["filename", "linenum", "author", "colnum", "txt"],
  "data": [
      { "parent": {
          "author": "user1", "filename": "main.c", "linenum": 15},
        "children": [
          { "parent": {
              "author": "user1", "filename": "main.c", "linenum": 15,
              "colnum": 2, "txt": "First word"} },
          { "parent": {
              "author": "user1", "filename": "main.c", "linenum": 15,
              "colnum": 11, "txt": "Second word"} },
          { "parent": {
              "author": "user1", "filename": "main.c", "linenum": 15,
              "colnum": 16, "txt": "Third word"} }
          ] },
      {  "parent": {
          "author": "user2", "filename": "test.c", "linenum": 44} },
      {  "parent": {
          "author": "root", "filename": "main.c", "linenum": 31} }
    ]
 }
```



```
#include <stdlib.h>

/*

  Warning: This function allocates memory to return to
           the user. The user is responsible for freeing
           this memory.

*/

#define MAX_INPUT 100

char * get_user_input()
```

#	File	Line	User ID	Column	Comment	Note
▼ 0	main.c	15	user1			
1	main.c	15	user1	2	First word	
2	main.c	15	user1	11	Second word	
3	main.c	15	user1	16	Third word	
4	test.c	44	user2			
5	main.c	31	root			

Figure 5. Nested Hierarchical Results Display

Note: The Column and Comment fields are blank in the table for the results not containing these data in their JSON parent elements.

Any element in the children array can, in turn, have its own list of children. The nesting is arbitrarily deep.

B.2 Notes Result Set Schema

The *notes* Result Set schema is based on the information stored in the annotation database. If one were to export the database to JSON, it would be loadable as a Result Set. The *notes* Result Set JSON must only contain two elements: (1) the "`schema`" from which the parsing is indicated, and (2) the "`data`" array.

B.2.1 Notes "**`schema`**" Element

JSON for the *notes* Result Set must include the following "schema" element to ensure Snippet parsing of the Result Set:

```
"schema": {"format": "notes", "version": 1}
```

B.2.2 Notes "**`data`**" Element

The "`data`" element is an array of JSON elements. Each element is composed of the fields listed below. Each element in the "`data`" array must contain "`filename`" and "`linenum`" values; all other values are optional. Snippet will display a table with *File*, *Line*, *Author,* and *Note* columns.

- "`author`" – displayed in *Author* column (optional)
- "`ctimestamp`" – creation timestamp (ignored, optional)
- "`filename`" – displayed in *File* column (required)
- "`linenum`" – displayed in *Line* column (required)
- "`tags`" – tags in the annotation (ignored, optional)
- "`timestamp`" – last edit timestamp (ignored, optional)
- "`txt`" – annotation string displayed in *Notes* column (optional)

An export of a demo annotation database resulted in the JSON in the following textbox. The display of these as a Result Set is provided in figure 10.

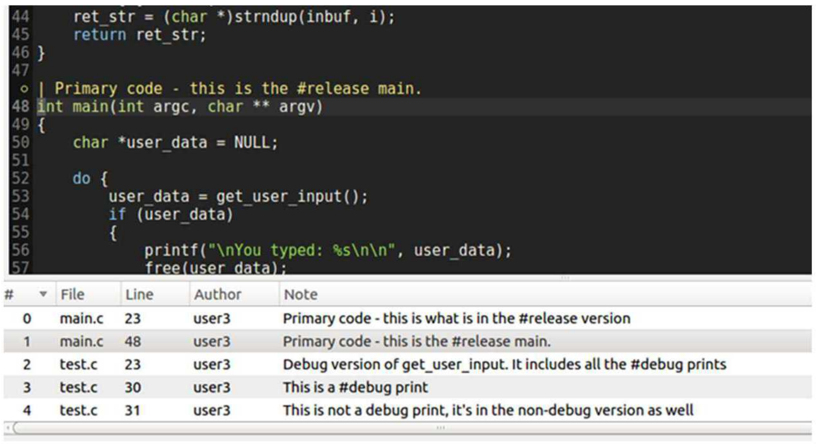
Flat Result Set JSON (Notes Database Export)

```
{
  "data": [
    {
      "author": "user3",
      "ctimestamp": "2018-09-10T10:38:36.834246",
      "filename": "#/main.c",
      "linenum": 23,
      "tags": [
         "release"
      ],
      "timestamp": "2018-09-10T10:39:30.210634",
      "txt": "Primary code - this is what is in the #release version"
    },
    {
      "author": "user3",
      "ctimestamp": "2018-09-10T10:38:54.154072",
      "filename": "#/main.c",
      "linenum": 48,
      "tags": [
         "release"
      ],
      "timestamp": "2018-09-10T10:38:54.154072",
      "txt": "Primary code - this is the #release main."
    },
    {
      "author": "user3",
      "ctimestamp": "2018-09-10T10:35:08.224153",
      "filename": "#/test.c",
      "linenum": 23,
      "tags": [
         "debug"
      ],
      "timestamp": "2018-09-10T10:35:08.224153",
      "txt": "Debug version of get_user_input. It includes all the #debug prints"
    },
```

(cont.)

Flat Result Set JSON (Notes Database Export) (cont.)

```
{
      "author": "user3",
      "ctimestamp": "2018-09-10T10:35:49.324205",
      "filename": "#/test.c",
```



```
44     ret_str = (char *)strndup(inbuf, i);
45     return ret_str;
46 }
47
 o | Primary code - this is the #release main.
48 int main(int argc, char ** argv)
49 {
50     char *user_data = NULL;
51
52     do {
53         user_data = get_user_input();
54         if (user_data)
55         {
56             printf("\nYou typed: %s\n\n", user_data);
57             free(user_data);
```

#	File	Line	Author	Note
0	main.c	23	user3	Primary code - this is what is in the #release version
1	main.c	48	user3	Primary code - this is the #release main.
2	test.c	23	user3	Debug version of get_user_input. It includes all the #debug prints
3	test.c	30	user3	This is a #debug print
4	test.c	31	user3	This is not a debug print, it's in the non-debug version as well

Figure 6. Flat Result Set Display

APPENDIX C – KEY BINDING MAP

Snippet's key bindings are listed in the tables below. These have been refined over time per Sandia's auditor preferences. Snippet provides four sets of *standard* key bindings that map to the default Snippet key bindings and different editor environments: vim, emacs and Windows. Auditors often prefer a specific editor, and it makes sense that the key bindings closest to the auditor's expectation will be more optimal for their usage. Snippet provides the user with the capability to personalize the key bindings through the preferences dialog box.

Table 4. Annotation Key Binding

Action	Snippet default	VIM	Emacs	Windows
Add/Edit Annotation	;	;	;	;
Add New Annotation	Ctrl+;	Ctrl+;	Ctrl+;	Shift+;
Show Annotations	A	A	A	F8
Update Annotations	Ctrl+R, F5	Ctrl+R, F5	Ctrl+R, F5	Ctrl+R, F5
Update All Annotations	Ctrl+Shift+R, Ctrl+F5	Ctrl+Shift+R, Ctrl+F5	Ctrl+Shift+R, Ctrl+F5	Ctrl+Shift+R, Ctrl+F5
Go to Next Annotation	V	V	V	F2
Go to Previous Annotation	Shift+V	Shift+V	Shift+V	Shift+F2
Annotation Info	Ctrl+'	Ctrl+'	Ctrl+'	F9
Open Inbox	Ctrl+/	Ctrl+/	Ctrl+/	Ctrl+/
Rename Tag				

Table 5. Cross Reference Navigation Key Bindings

Action	Snippet default	VIM	Emacs	Windows
Open Cross Reference Dialog	Ctrl+\	Ctrl+\	Ctrl+\	Ctrl+\
Smart Search	Ctrl+H	Ctrl+H	Ctrl+H	Return
Open in New Tab				
Open in New Split	Ctrl+Shift+S	Ctrl+Shift+S	Ctrl+Shift+S	Ctrl+Shift+S
Search for Definitions	Ctrl+]	Ctrl+], Ctrl+Shift+-,g	Ctrl+]	F12
Search for References	Ctrl+[	Ctrl+[, Ctrl+Shift+-,c	Ctrl+[	Shift+F12
Search for File	Alt+]	Alt+]	Alt+]	F11
Pop XRef Location	Ctrl+T	Ctrl+T	Ctrl+T	Shift+Backspace
Go to Next Result	Ctrl+N	Ctrl+N	Ctrl+N	Ctrl+Alt+Down
Go to Previous Result	Ctrl+P	Ctrl+P	Ctrl+P	Ctrl+Alt+Up
Go to Newer Result	Ctrl+.	Ctrl+.	Ctrl+.	Ctrl+Alt+Right
Go to Older Result	Ctrl+,	Ctrl+,	Ctrl+,	Ctrl+Alt+Left

Table 6. Code Viewer Navigation Key Bindings

Action	Snippet default	VIM	Emacs	Windows
Search	/	/	/ Ctrl+S	/ Ctrl+F
Search Backward	?	?	?	?
Search Current Word	Shift+8	Shift+8	Shift+8	Ctrl+S
Search Current Word Backward	Shift+3	Shift+3	Shift+3	Ctrl+Shift+S
Search Next	N	N	N	N F3
Search Previous	Shift+N	Shift+N	Shift+N	Shift+N Shift+F3
History Forward	Alt+Right	Ctrl+I Alt+Right	Alt+Right	Alt+Right
History Backward	Alt+Left	Ctrl+O (letter) Alt+Left	Alt+Left	Alt+Left Backspace
Go to Line	Ctrl+G	Ctrl+G	Ctrl+G	Ctrl+G
Show Outline	O (letter)	O (letter)	O (letter)	F6
Show Results	R	R	R	F7
Focus Omnibar	Ctrl+K	Ctrl+K	Ctrl+K	Ctrl+K Ctrl+L
Focus Code Viewer	`	`	`	`
Go to Bracket	Shift+5	Shift+5	Shift+5	Ctrl+]
Global declaration	G, Shift+D	G, Shift+D	G, Shift+D	Shift+Space
Local declaration	G, D	G, D	G, D	Space

Table 7. Misc. UI Shortcut Key Bindings

Action	Snippet default	VIM	Emacs	Windows
Next Result	Ctrl+Shift+J	Ctrl+Shift+J	>	Ctrl+Down
Previous Result	Ctrl+Shift+K	Ctrl+Shift+K	<	Ctrl+Up
New Tab	Ctrl+Shift+T	Ctrl+Shift+T	Ctrl+Shift+T	Ctrl+T
Close Tab	Ctrl+W	Ctrl+W	Ctrl+W	Ctrl+W Ctrl+F4
Next Tab	Ctrl+PgDown	Ctrl+PgDown	Ctrl+PgDown	Ctrl+Tab Ctrl+PgDown
Previous Tab	Ctrl+PgUp	Ctrl+PgUp	Ctrl+PgUp	Ctrl+Shift+Tab Ctrl+PgUp
Open File	Ctrl+Shift+O (letter)	Ctrl+Shift+O (letter)	Ctrl+Shift+O (letter)	Ctrl+Shift+O (letter)

Table 7. Misc. UI Shortcut Key Bindings

Action	Snippet default	VIM	Emacs	Windows
Open File in New Tab				
Show Console	Shift+1	Shift+1	Shift+1	Shift+1
Highlight Code	Ctrl+1	Ctrl+1	Ctrl+1	Ctrl+1
Highlight Code	Ctrl+2	Ctrl+2	Ctrl+2	Ctrl+2
Highlight Code	Ctrl+3	Ctrl+3	Ctrl+3	Ctrl+3
Highlight Code	Ctrl+4	Ctrl+4	Ctrl+4	Ctrl+4
Highlight Code	Ctrl+5	Ctrl+5	Ctrl+5	Ctrl+5
Highlight Code	Ctrl+6	Ctrl+6	Ctrl+6	Ctrl+6
Highlight Code	Ctrl+7	Ctrl+7	Ctrl+7	Ctrl+7
Clear Highlights	Ctrl+9	Ctrl+9	Ctrl+9	Ctrl+9
Highlight as Comment	Ctrl+0 (zero)	Ctrl+0 (zero)	Ctrl+0 (zero)	Ctrl+0 (zero)
Fold	F	F	F	F
Unfold	Shift+F	Shift+F	Shift+F	Shift+F

Table 8. Mark Navigation Key Bindings

Action	Snippet default	VIM	Emacs	Windows
Set Mark	M	M	M	M
Jump to Mark	‘	‘	‘	‘
Set Anon Mark	M, Ctrl+M	M, Ctrl+M	M, Ctrl+M	M, Ctrl+M
Cycle Anon Marks	Ctrl+M	Ctrl+M	Ctrl+M	Ctrl+M
Clear Anon Marks	Ctrl+Shift+M	Ctrl+Shift+M	Ctrl+Shift+M	Ctrl+Shift+M

Table 9. Global Code Viewer Movement Key Bindings

Action	Snippet default	VIM	Emacs	Windows
Right	L	L Space	Ctrl+F	
Left	H	H Backspace	Ctrl+B	
Up	K	K	Ctrl+P	
Down	J	J	Ctrl+N	
Begin	G, G	G, G	G, G	
End	Shift+G	Shift+G	Shift+G	
Start of Line	0 (zero)	0 (zero)	Ctrl+A	
End of Line	Shift+4	Shift+4	Ctrl+E	
Start of Word	D	D	D	

Table 9. Global Code Viewer Movement Key Bindings				
Action	**Snippet default**	**VIM**	**Emacs**	**Windows**
End of Word	E	E	E	
Word Left	B	B	B	
Word Right	W	W	W	
View Top	Shift+H	Shift+H	Shift+H	Shift+H
View Middle	Shift+M	Shift+M	Shift+M	Shift+M
Center	Ctrl+L	Ctrl+L	Ctrl+L	Shift+L
Page Up	Ctrl+B	Ctrl+B	(Could not bind Ctrl+Z)	
Page Down	Ctrl+F	Ctrl+F	Ctrl+V	
Line Up	Ctrl+Y	Ctrl+Y		
Line Down	Ctrl+E	Ctrl+E		