Usability of Alternative Semantic Highlighting Features

Charlie Ringström
E18, Lund University, Sweden
ch5573ri-s@lu.student.se

Jonathan Frisk
D17, Lund University, Sweden
jo1875fr-s@lu.student.se

Abstract

Today most programmers use some kind of active programming support when coding, such as auto completion and function references. One of these features is semantic highlighting. For this project, this is implemented with a language server that communicates with the text editor. This paper explores three different ways to semantically highlight the code. The first feature highlights Strings compared with the == operator to indicate that the code might be flawed. The other features highlights both for loop variables with unique colors, and also each bracket pair a unique color. By using the ExtendJ Java compiler, the proposed language server communicates with the text editors using LSP. The for loop variable highlighting and bracket coloring features were evaluated by showing code snippets to 12 test subjects, with or without semantic highlighting, and the time to solve each programming task was measured. For the two tests, the results showed a 5 % and a 13% reduction in time to solve each programming task with the semantic highlighting enabled. However, the standard deviation is too big to draw any conclusions. Additionally, it remains unknown how the results would change in real life usage.

1 Introduction

Most programmers today would expect some kind of active programming support when editing code in a modern text editor. Examples of such support may be syntax highlighting, function references and auto completion. While this is great for the user, it is also quite difficult for both language designers and text editor developers to implement. In 2016, Microsoft launched an open standard to facilitate this called the Language Server Protocol (LSP). Instead of having each editor target each language server they can all just implement the LSP protocol to make it work with any LSP compatible language [7].

This project has the goal to develop new semantic highlighting features and evaluate their usability by implementing a language server (called ExtendJ-lsp [3]) using the LSP protocol. Unlike most other code highlighting, ExtendJ-lsp doesn't highlight the code with regex pattern matching as is typically done. The language server instead constructs an Abstract Syntax Tree (AST) to semantically interpret the code. Although regex based highlighting is quicker and works well in most situations, semantic highlighting can highlight the code contextually, and could for example highlight a variable differently depending on its type.

This report also explores the option of using the Extendj [10] Java compiler to process the code in ExtendJ-lsp. Extendj is a Java compiler built with extensibility in mind. This makes it easy to integrate new features into the language server. The solution utilizes LSP4J [2] to implement LSP in Java. The text editors chosen for this project where Visual Studio code, and Neovim. To communicate with VS Code, a custom extension was made, and for Neovim a small configuration file was made.

Extendj-lsp has three different semantic highlighting features. The first feature highlights nested for loop variables with different colors. The second feature highlights a String to String comparison if it's done with == instead of `.equals()` to indicate that the code might be flawed. The third feature highlights brackets pairs with unique colors.

The features of the language server were tested on people to evaluate its usefulness. To do this, a comparison was performed to measure the time it took for the test subjects to find a bug in a code samples with and without semantic highlighting enabled.

The report is structured as follows. The *background* section aims to describe the underlying concepts and terminology behind the project. The *Extendj-lsp language server* section aims to describe more in depth how the language server works. The *evaluation* section describes in depth how the project was evaluated. Following, we present the *related work*, conclusions, *future work* and *acknowledgements.*

2 Background

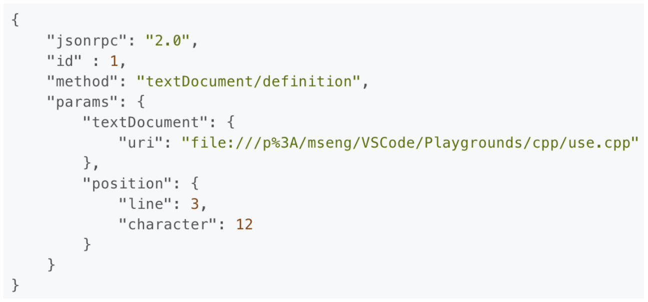
2.1 Language Server Protocol (LSP)

Language Server Protocol (LSP) standardizes the communication between language servers and text editors, using JSON-RPC to transfer information.

The protocol specifies notification requests that the text editor should respond to at certain events, and vice versa. Some examples are: initialization of

Course paper, EDAN70, Lund University, Sweden
January 22, 2022.

connection between server and editor, when user opens a document, or when a user edits the document.



```
{
    "jsonrpc": "2.0",
    "id" : 1,
    "method": "textDocument/definition",
    "params": {
        "textDocument": {
            "uri": "file:///p%3A/mseng/VSCode/Playgrounds/cpp/use.cpp"
        },
        "position": {
            "line": 3,
            "character": 12
        }
    }
}
```

Figure 1. Example of an LSP request

2.2 Semantic Highlighting

Semantic Highlighting is a way of highlighting code semantically, added to LSP 3.16 in December 2020. Semantic highlighting (usually) requires the language server to build an abstract syntax tree to process the code. By doing this, the language server could, for example, keep track of which variable is a `String`, `int` or `double` and color them accordingly, which wouldn't be possible with regex based highlighting. While semantic highlighting can do more advanced highlighting than regex based highlighting, it is less responsive and more resource intensive. Regex highlighting is also more forgiving, and can colorize the code even though the code has syntax errors, something which many semantic highlighters cannot do. Because of this, semantic highlighting and regex highlighting is almost always used in conjunction with each other.

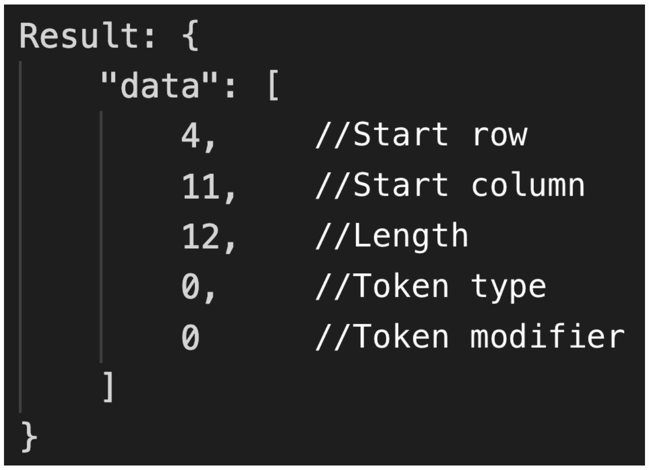
Data is sent from the language server as semantic tokens. These contain information indicating where and how the code should be colorized, as shown in figure 2.

2.3 Hover tooltip

Hover tooltips is a feature introduced in LSP when it was first released in version 1.0. A hover tooltip usually contains information about a code segment which the user hovers their mouse over. As shown later, this feature is not fully implemented in Extendj-lsp, and acts instead as proof that the language server works on editors that does not support semantic highlighting.

2.4 Extendj

ExtendJ-lsp uses the ExtendJ compiler to semantically process the code by building an Abstract Syntax Tree (AST). ExtendJ is itself built using the JastAdd [4] metacompiler. One of the advantages of JastAdd is its use of Reference Attribute Grammars[8]. (RAG)



```
Result: {
    "data": [
        4,      //Start row
        11,     //Start column
        12,     //Length
        0,      //Token type
        0       //Token modifier
    ]
}
```

Figure 2. Semantic token information

These attributes can be written using AspectJ-style syntax.

```
/* Example of an attribute that
returns the name of a variable.
VarAccess indicates that the
name() method should be
placed in the VarAccess class */
syn String VarAccess.name() = getID();
```

RAGs have several properties that are useful when writing a compiler. For example, instead of inheriting from a Java class hierarchy, attributes can inherit from nodes in the AST, which can change from program to program. JastAdd attributes also utilizes memoization, which reduces unnecessary code execution.

In JastAdd, both attributes and regular methods are placed in aspects. One advantage of using aspects is that it that related code can be placed into the same file, regardless of what class they belong to. For example, if each primitive datatype were to have a `getType()` method, one would need to write a unique `getType()` method in each of these classes, which can be problematic. This problem is circumvented by using aspect programming, which would allow a programmer to keep every `getType()` method in the same file. This is possible to do by other means as well, (visitor pattern, for example) but they can be hard to read and requires a lot of boilerplate code.

2.5 LSP4J

ExtendJ-lsp uses LSP4J [2] to acts as an intermediary between LSP's JSON-RPC messages and regular Java code.

For each defined LSP message, LSP4J provides a corresponding method to it via its Java interface.

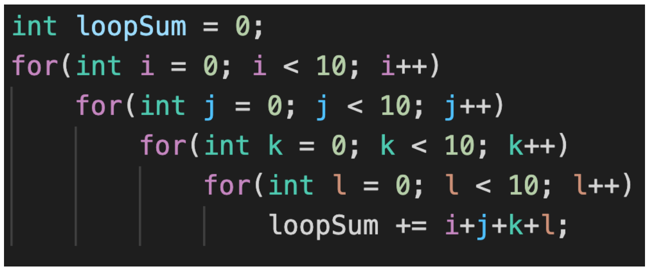
3 Extendj-lsp language server

3.1 Features

Extendj-lsp can do the following:

3.1.1 Semantic Highlight for for loop variables

In order facilitate finding bugs in for loops, Extendj-lsp uses semantic highlighting to assign different colors to each nested for loop variable depending on their depth.

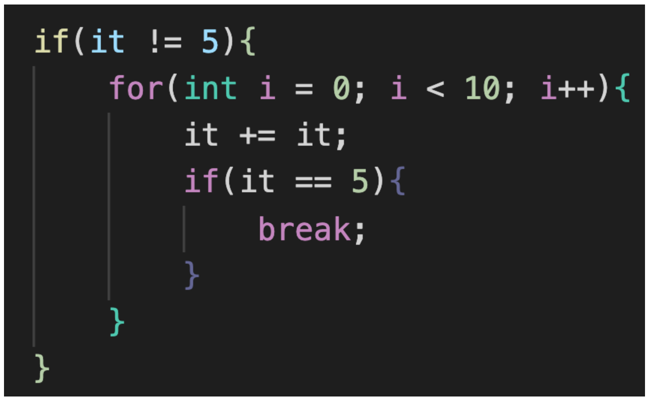


```
int loopSum = 0;
for(int i = 0; i < 10; i++)
    for(int j = 0; j < 10; j++)
        for(int k = 0; k < 10; k++)
            for(int l = 0; l < 10; l++)
                loopSum += i+j+k+l;
```

Figure 3. For loop variable coloring

3.1.2 Semantic highlighting for brackets

Some programmers sometimes gets confused trying to identify bracket pairs when dealing with nested code. To facilitate this, Extendj-lsp provides unique colors for each bracket pair.



```
if(it != 5){
    for(int i = 0; i < 10; i++){
        it += it;
        if(it == 5){
            break;
        }
    }
}
```

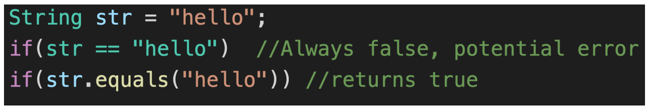
Figure 4. Bracket color matching

3.2 Experimental Features

To test a wider range of features, but with less practical usefulness, Extendj-lsp also has the ability to do the following test features:

3.2.1 Semantic Highlighting for String equality comparisons

Strings are equality checked in Java by using the `equal()` method. Some inexperienced Java programmers might try to equality check using the == operator instead, which compares the object IDs of the Strings. To warn the programmer about this, Extendj-lsp uses an experimental feature to highlight the String comparison that is recommended to be changed. This feature is more commonly implemented with a simple warning message in other language servers.

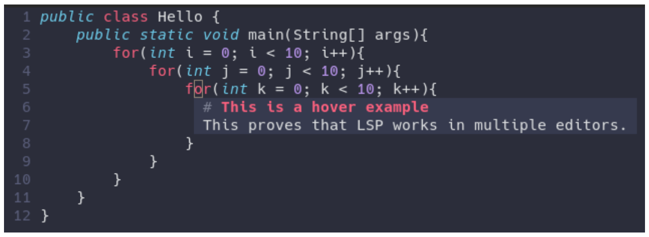


```
String str = "hello";
if(str == "hello")  //Always false, potential error
if(str.equals("hello")) //returns true
```

Figure 5. Bad String equality check highlighting

3.2.2 Hover tool tip placeholder

One requirement for the project was to prove that the language server actually works on multiple code editors. Since semantic highlighting is a relatively new feature, there is a lack of code editors that supports this feature. To circumvent this, hover tool tips were added instead to demonstrate the portability of the language server. As figure 6 shows, the tooltip provides no useful information to the user, and merely acts as a placeholder, proving that the language server works in Neovim.



```
public class Hello {
    public static void main(String[] args){
        for(int i = 0; i < 10; i++){
            for(int j = 0; j < 10; j++){
                for(int k = 0; k < 10; k++){
                 # This is a hover example
                 This proves that LSP works in multiple editors.
                }
            }
        }
    }
}
```

Figure 6. Hover tool tip placeholder in Neovim

3.3 Implementation

3.3.1 Overview

The language server consists of multiple parts which is described in figure 7.

3.3.2 Implementation of Extendj-lsp

To implement Extendj-lsp, ExtendJ was added as a dependency in a seperate submodule, inspired by the ExtendJ example analysis project [1]. Custom attributes are added to the AST by adding `.jrag` files. Subsequently, when building the submodule, custom AST class attributes becomes available for the language server to use after parsing a file.

To implement the semantic highlighting features using ExtendJ, following was performed:

The three semantic highlighting features in this report (for-loop, brackets, string equals string) implements a `Set`

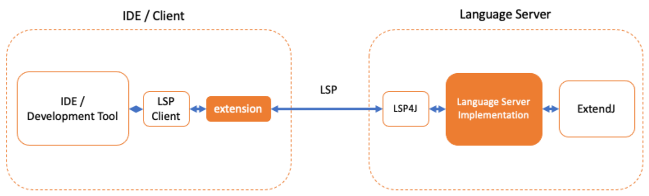
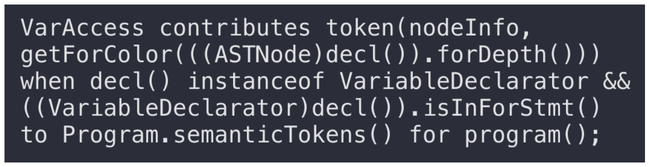


Figure 7. Overview of architecture for a Language Server connected to the text editor with LSP. To the right is the Language Server. It handles both LSP communication by using LSP4J as well as static analysis by using ExtendJ. To the right is the text editor that has a LSP client and an extension. Shapes filled in orange are developed specifically for this project, whereas the transparent shapes are dependencies.

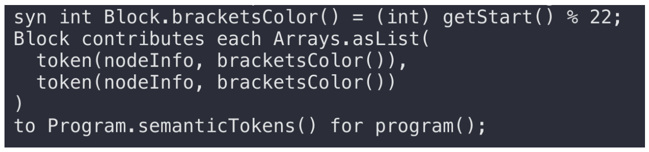
collection attribute in the AST called `semanticTokens()`. All semantic highlighting features contributes with new tokens to the `Set` when a specific condition is met. The tokens are objects of a custom data class that contain information about the semantic token.

The implementation for each feature are shown in figure 8, 9, and 10. All code is implemented in JastAdd's `.jrag` files. Additionally, to simplify the code, node data from the AST, such as line and column number, is written in this report as *nodeInfo*.



```
VarAccess contributes token(nodeInfo,
getForColor(((ASTNode)decl()).forDepth()))
when decl() instanceof VariableDeclarator &&
((VariableDeclarator)decl()).isInForStmt()
to Program.semanticTokens() for program();
```

Figure 8. Implementation of for-loop coloring. This collection attribute adds a token with color information to the `semanticTokens()` set in the `Program` root node. The color is decided by calculating the depth of the for-loop nesting.

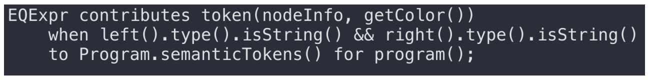


```
syn int Block.bracketsColor() = (int) getStart() % 22;
Block contributes each Arrays.asList(
  token(nodeInfo, bracketsColor()),
  token(nodeInfo, bracketsColor())
)
to Program.semanticTokens() for program();
```

Figure 9. Implementation of bracket coloring. This collection attribute adds a token with color information to the `semanticTokens()` set in the `Program` root node.

3.3.3 Implementation of Hover Tool Tip

Since hover tool tip is implemented as a placeholder, nothing is computed in the AST. Instead, a hard coded string is sent as the tool tip text.



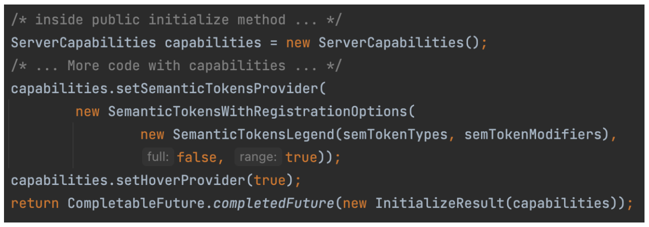
```
EQExpr contributes token(nodeInfo, getColor())
    when left().type().isString() && right().type().isString()
    to Program.semanticTokens() for program();
```

Figure 10. Implementation of String equality check coloring. This contributes a semantic token if an == token has a `String` object on its left and right side. getColor() returns a predefined color as an int.

3.3.4 Implementation of LSP4J in Language Server

Extendj-lsp uses LSP4J in order to convert JSON-RPC messages to simple Java method calls. In order to implement it, two main parts are needed. Firstly, an instance of *ServerCapabilities* for the server, and secondly, an instance of *TextDocumentService*, which are interfaces defined in LSP4J.

A **ServerCapabilities** instance is defined and returned when the client calls *initialize* over LSP, which calls the initialize method defined in the LSP4J interface.



```
/* inside public initialize method ... */
ServerCapabilities capabilities = new ServerCapabilities();
/* ... More code with capabilities ... */
capabilities.setSemanticTokensProvider(
        new SemanticTokensWithRegistrationOptions(
                new SemanticTokensLegend(semTokenTypes, semTokenModifiers),
                full: false, range: true));
capabilities.setHoverProvider(true);
return CompletableFuture.completedFuture(new InitializeResult(capabilities));
```

Figure 11. Creation of a ServerCapabilities object.

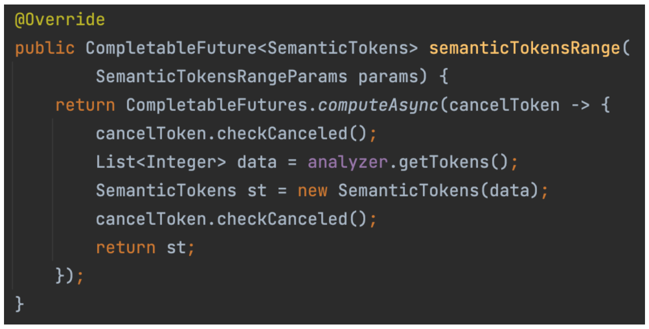
A **TextDocumentService** instance is created upon initialization. It has methods overiding the *didOpen*, *didChange*, *hover*, *semanticTokensRange* methods in TextDocumentService. A text document is parsed and analyzed when *didOpen* or*didChange* is called. For example, when the text editor/IDE sends a semantic highlighting request, *semanticTokensRange* methods gets called respectively. With the attributes defined in the AST, these `semanticTokensRange()` returns semantic highlighting tokens, which LSP4J translates to LSP replies.

In order to send the data, the methods semanticTokensRange() and hover() returns a CompletableFuture<T> which can be cancelled in the future, which minimizes unnecessary computations.

4 Evaluation

4.1 Usability of Semantic Highlighting features

In order to evaluate the usability of the semantic highlighting features, two tests were given to a number of test subjects. They were instructed to complete two tests, shown in figure 13 and 14 with semantic highlighting enabled or disabled randomly. These tested loop variable feature (figure

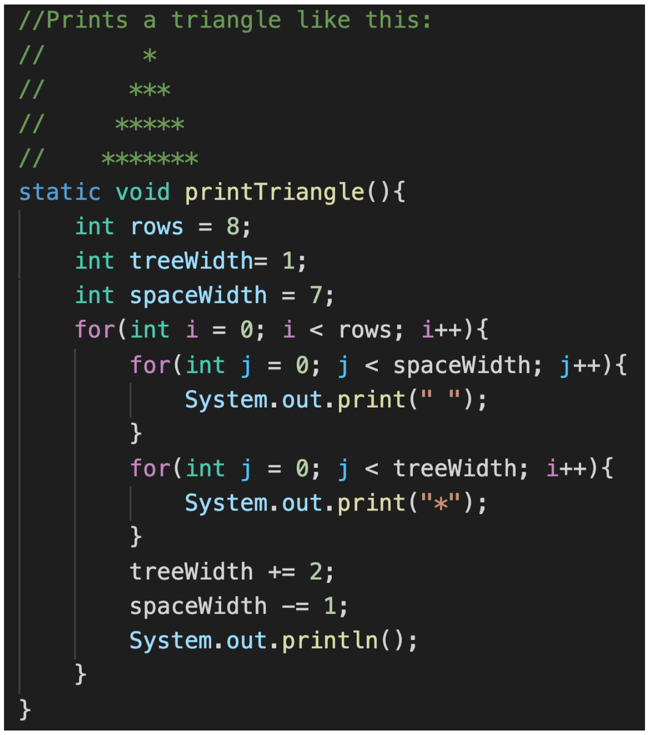


```
@Override
public CompletableFuture<SemanticTokens> semanticTokensRange(
        SemanticTokensRangeParams params) {
    return CompletableFutures.computeAsync(cancelToken -> {
        cancelToken.checkCanceled();
        List<Integer> data = analyzer.getTokens();
        SemanticTokens st = new SemanticTokens(data);
        cancelToken.checkCanceled();
        return st;
    });
}
```

Figure 12. A simplified example of the LSP4J method semanticTokensRange(), overridden from TextDocumentService.

8) and the brackets feature (figure 9).

In the first test the test subjects were instructed to find a small bug in an otherwise working piece of code. As can be seen in figure 13, one of the loop variables has been changed from a `j` to an `i`.

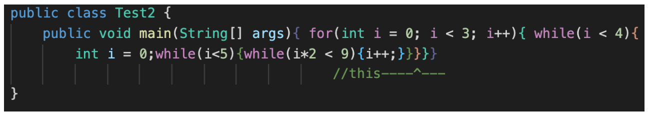


```
//Prints a triangle like this:
//       *
//      ***
//     *****
//    *******
static void printTriangle(){
    int rows = 8;
    int treeWidth= 1;
    int spaceWidth = 7;
    for(int i = 0; i < rows; i++){
        for(int j = 0; j < spaceWidth; j++){
            System.out.print(" ");
        }
        for(int j = 0; j < treeWidth; i++){
            System.out.print("*");
        }
        treeWidth += 2;
        spaceWidth -= 1;
        System.out.println();
    }
}
```

Figure 13. The test subjects were instructed to find a big in this code. (wrong variable used in one of the for loops)

As expected, most test subjects first tried to find logical errors in the code. The idea was to let the semantic highlighting guide them to find the error more easily.

For the next test people were instructed to locate the the first while loop's ending bracket.



```
public class Test2 {
    public void main(String[] args){ for(int i = 0; i < 3; i++){ while(i < 4){
        int i = 0;while(i<5){while(i*2 < 9){i++;}}}}}
                                      //this----^---
}
```

Figure 14. Code of brackets test.

Unfortunately, this test was somewhat confusing for many of the participants, so some of the tests had to be excluded because of that.

4.2 Results

The tests were conducted both psychically and over the internet, and all participants knew how to program in Java. Six people tested the test in figure 13 with semantic highlighting and six people tested without. Four people did the test in figure 14 with semantic highlighting, and four did without semantic highlighting.

The results (seen in figure 15) shows that the average time for the triangle test is 2:37 min for semantic highlighting and 2:45 min for the people without semantic highlighting. Additionally, for the Brackets test, the results was 0:57 min with semantic highlighting whereas the other got 1:06 min without.

A comparison between highlighted and non-highlighted shows that semantic highlighting decreased the time with 5% for the Triangle test and with 13 % for the Brackets test.

The standard deviation for the triangle test with semantic highlighting was 1m 46, and without semantic highlighting it was 1m 58s. For the brackets test with semantic highlighting the standard deviation was 30 seconds, and without semantic highlighting it was 44 seconds.
For comments on the results, see the conclusions section.

5 Related work

There are other projects which uses Extendj to make a language server for LSP. Examples of these are Joakim Ericson's language server,[5] and Fredrik Siemund & Daniel Tovesson's language server.[6] Both language servers use the LSP protocol in a more traditional way, and uses information provided by the Extendj compiler to highlight errors in the code.

6 Conclusion and Future Work

The test results seemed to be in favor of to the semantic highlighter, and people generally completed the tests quicker with highlighting enabled. However, the evaluation is far from perfect and no conclusions can be scientifically drawn, based on the hypothesis test [9]. The reason is the large standard deviation of test times, in conjunction with

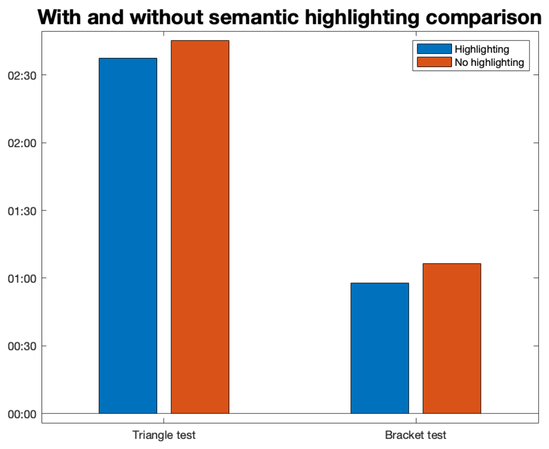


Figure 15. Results for the Triangle test (left) and Brackets test (right). The blue bars represents data from test subjects that used semantic highlighting and the orange represents without semantic highlighting.

the small sample size of test subjects.

Another issue is that the tests are designed to prove that the language server can be useful in *some* situations. But highlighting too much of the text can also be detrimental. Unnecessary highlights can detract the programmer's attention from more important information.

In a real-world scenario, an Extendj-lsp user would know what features their LSP has, and might therefore know what to look for when fixing bugs. When asked, several of the test subjects didn't even notice that the for loop variable colors were different.

A potential flaw is that the test in figure 14 was somewhat contrived, and most programmers use line breaks to make their code easier to read.

Another probable issue is how Extendj-lsp highlights bad String comparisons in java (figure 5) by highlighting it. A more suitable way to do this would be to use a simple warning message instead.

This project uses LSP's semantic highlighting feature in a rather unorthodox manner. This can be seen in the specification itself, which currently does not have any way to hard code colors. Instead, LSP has different token types and modifiers, such as Class, Struct and Readonly. Extendj-lsp instead picks a semantic token type which happens to look right with the default VS Code theme. While this method of highlighting code works, it may not be optimal.

Extendj showed promising signs to be a good backend for a language server. One reason was because of how easy it was to add new features. This project may be helpful for future LSP developers in order to widen the scope of potential candidates of backends for an LSP. Based on this, future work may be interesting in order to compare ExtendJ-lsp with the current well known language servers.

To improve Extendj-lsp in the future, more LSP features could be added to it, such as jump to definition and warning messages. As previously mentioned, this LSP uses semantic highlighting in a rather unorthodox manner, and a more traditional semantic highlighter could instead be added.

Acknowledgements

Thanks to Anton Risberg Alaküla, for his enthusiastic and helpful approach as our supervisor.

References

[1] 2021. analysis-template. (2021). https://bitbucket.org/extendj/analysis-template/src/master/
[2] 2021. Eclipse LSP4J. (Apr 2021). https://projects.eclipse.org/projects/technology.lsp4j
[3] 2022. extendj-lsp. (2022). https://bitbucket.org/edan70/lsp-charlie-jonathan/src/master/
[4] Torbjörn Ekman and Görel Hedin. 2007. The Jastadd Extensible Java Compiler. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '07)*. Association for Computing Machinery, New York, NY, USA, 1–18. https://doi.org/10.1145/1297027.1297029
[5] Joakim Ericson. 2018. 2018-languageserver-joakim. (2018). https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2018/Reports/Ericson.pdf
[6] Daniel Tovesson Fredrik Siemund. 2018. Language Server Protocol for ExtendJ. (2018). https://fileadmin.cs.lth.se/cs/Education/edan70/CompilerProjects/2018/Reports/SiemundTovesson.pdf
[7] N. Gunasinghe and N. Marcus. 2021. *Language Server Protocol and Implementation: Supporting Language-Smart Editing and Programming Tools*. Apress. https://books.google.se/books?id=zeuezgEACAAJ
[8] Görel Hedin. 2000. Reference Attributed Grammars. *Informatica* 24, 3 (2000), 301 – 317. http://ludwig.lub.lu.se/login?url=https://search.ebscohost.com/login.aspx?direct=true&db=edsswe&AN=edsswe.oai.lup.lub.lu.se.42369945.8949.4506.a212.280893694207&site=eds-live&scope=site
[9] R. Peck and J.L. Devore. 2011. *Statistics: The Exploration & Analysis of Data*. Cengage Learning. https://books.google.se/books?id=NYcIAAAAQBAJ
[10] Jesper Öqvist. 2018. ExtendJ: Extensible Java Compiler. In *Conference Companion of the 2nd International Conference on Art, Science, and Engineering of Programming (Programming'18 Companion)*. Association for Computing Machinery, New York, NY, USA, 234–235. https://doi.org/10.1145/3191697.3213798