# 개별연구 보고서

학생 개별 작성용

| 수행 학기 | 2022 학년도 1 학기 | | | |
|---|---|---|---|---|
| 교과목 정보 | 교과목명 | | 분반 | |
| 교과담당교수 | 소속 | | 교수명 | |
| | 이름 | | 학번 | |
| 학생정보 | 학과 | | 전화번호 | |
| | 학년 | | 이메일 | |

## 요약 보고서

| | |
|---|---|
| 작품명<br>(프로젝트명) | **Vscode 웹 버전 ( vscode.dev ) 을 위한 린트 플러그인 구현 ( web-lint )** |
| 1.<br>연구·개발<br>동기/목적/<br>필요성 및 개발<br>목표 | Vscode 공식 사이트에서는 vscode.dev 환경을 vscode 웹버전이라고 설명하고 있다. 이때, vscode.dev 환경은 웹 상에서 실행되기 때문에 기존 vscode 기반 플러그인들이 실행되는 환경인 node.js 와 관련된 디펜던시를 이용할 수 없다.<br>많은 lint 프로그램들이 node.js 환경에 종속된 fs, ps 등의 라이브러리를 사용하므로 웹 상에서 동작하는 vscode.dev 환경에서 실행되지 않는다.<br>따라서 웹 상에서 실행되는 vscode 린트 플러그인을 구현할 필요가 있다.<br>개별 설정을 통해 린트 규칙을 저장 및 설정 가능<br>설정한 린트 규칙을 이용하여 린트 패턴 위반 여부 알림 |
| 2.<br>최종 결과물 소개 | web-tree-sitter 라이브러리 기반으로 코드를 파싱하여 린트 규칙에 맞는 쿼리를 검사.<br>vscode 의 Memento API 을 이용하여 웹 브라우저 상에 쿼리를 json 형태로 저장. 캐시를 제거하기 전까지는 해당 내역이 유지됨.<br>사용자는 set-lint 명령 및 load-lint 명령을 통해 린트 규칙을 vscode.dev 상에 설정하거나, 로드 된 린트 규칙들을 불러올 수 있음. ( 개별 설정을 통해 린트 규칙 저장 및 설정 가능 )<br>캡처된 린트 규칙과 관련된 내용을 vscode 의 Diagnostic 에 기반하여 Problems 창 및 밑줄 형태로 표현. 대중적인 lint 플러그인의 동작을 묘사. ( 패턴 위반 여부 알림 ) |
| 3.<br>프로젝트 추진<br>내용 | web-tree-sitter 기반 CST 생성 및 쿼리 검사<br>vscode 의 Memento API 을 이용하여 웹 상에 쿼리 내용 저장.<br>Diagnostic 을 이용하여 Problems 혹은 밑줄을 통해 에러 내역 사용자에게 알림.<br>vscode 의 command 관련 API 을 이용하여 사용자가 쿼리 설정, 쿼리 로드 동작을 직접 수행할 수 있도록 구현. |
| 4.<br>기대효과 | vscode.dev 환경에서 사용자는 자신 혹은 타인이 현재 플러그인의 규칙 포맷에 맞게 만들어 둔 규칙을 불러와 현재 자신이 작성하는 코드가 lint 규칙에 맞는지 검사할 수 있다.<br>팀 단위로 vscode.dev 을 사용하는 경우 동일한 lint 규칙을 이용하여 코딩 스타일을 일관성 있게 관리할 수 있다. |

# 목차

# 1. 서론

vscode 의 웹 버전인 vscode.dev 은 2021 년 10 월 20 일에 처음 대중에게 공개되었다. 해당 사이트는 다양한 플러그인을 기반으로 동작하는 vscode IDE 을 웹 버전으로 포팅한 웹 IDE 로, 유저의 파일 시스템에 접근하여 코딩을 직접 수행할 수 있다는 장점을 가지고 있다.

그러나 vscode 와는 달리 vscode.dev 은 기존의 node.js 환경이 아닌 web 환경에서 실행되기 때문에 node.js 에 의존성을 가지는 동작을 수행할 수 없으며, 순수한 JavaScript 코드 및 Web Assembly 코드만 실행할 수 있다. 따라서 fs, ps 등 node.js 환경의 동작을 요구하는 코드를 포함하고 있는 대부분의 vscode extension 을 실행할 수 없다는 문제를 가지고 있다.

따라서 현재 과제에서는 vscode 의 웹 버전인 vscode.dev 환경에서 실행될 수 있는 lint 프로그램을 구현하기 위해 현재 환경에 대한 제약조건 및 요구사항을 분석하고 해당 내용을 기반으로 실제로 lint 프로그램을 구현하는 것을 목표로 한다.

# 2.1 관련/배경 연구

## 2.1.1 과제 요구사항 분석

현재 과제에 대해 제시된 요구사항을 알기 위해 제시된 공지사항을 분석해보자.

안녕하세요 모두들 중간고사는 잘 치루셨나요?
개별연구에 대한 공지사항을 안내드립니다.
본 개별연구는 http://shastra.dongguk.edu/ 에서 제공하는
온라인 프로그래밍 실습시스템의 기능을 향상시키는 주제를 다룹니다.

22 학년도 1 학기의 주제는 다음과 같습니다.

http://shastra.dongguk.edu/의 웹 IDE 시스템을 위한 plug-in 구현
샤스트라 웹 IDE 는 visual studio code 웹 버전으로 구현되어 있으며,
해당 버전의 모든 기능을 제공합니다.

수강생 분들은 visual studio code 웹 버전의 plug-in 을 구현하여 그 결과물과 함께 보고서를 제출하시면 됩니다.

plug-in 의 주제는 lint/clint/splint 와 유사한 visual studio code 웹 버전에서 작성 중인 코드의 코딩 스타일과 패턴을 분석하는 프로그램입니다. (github 등의 공개코드를 잘 활용하시기 바랍니다.)

plug-in 은 별도 설정을 통해 정규표현 또는 별도의 패턴 기술을 통해 분석하고자 하는 패턴 리스트를 저장할 수 있어야 하며, 분석은 저장된 패턴을 이용하여 해당 패턴을 잘 지켰는지, 또는 위배했는지 여부를 확인할 수 있어야 합니다.

1. 패턴 리스트 저장
2. 패턴 위반 여부 분석
- 어느 부분이 잘못됐는지 알려주기

참고용 코드 패턴은 CERT Secure Coding Standard 를 일부 참조하실 수 있으며,
분석 대상 언어는 C/C++, Java, Python 3 가지 언어로 한정하면 됩니다.

제시된 공지사항에 따르면, 현재 과제는 visual studio code web 버전을 위한 lint 플러그인을 구현하는 것을 목적으로 하고 있다. 이때 visual studio code for web 을 브라우저를 통해 검색해보면 vscode.dev 환경을 의미한다는 것을 알 수 있다[1]. visual studio code 의 개발사인 Microsoft 역시 웹 버전의 환경으로 vscode.dev[2] 환경을 소개하고 있으므로, 현재 과제는 **vscode.dev** 환경에서 동작하는 lint 플러그인의 구현으로 구체화할 수 있다.

Wikipedia 에 따르면 lint[3] 프로그램은 프로그래밍 오류, 버그, 스타일 오류 및 의심스러운 구성에 대한 플래그를 지정하는데 사용되는 정적 코드 분석 도구로, 현재 eslint[4], pylint[5] 등 다양한 언어에 대한 lint 플러그인 혹은 프로그램이 존재하고 있다.

언급한 lint 프로그램 중 eslint 는 자바스크립트 코드에 대한 패턴을 식별 및 보고하는 도구로, Espree[6] 라이브러리를 이용하여 분석하고자 하는 코드를 Abstract Syntax Tree 형태로 파싱한 이후 코드의 패턴을 평가하는 방식으로 동작한다. 각각의 패턴은 개별 파일 단위로 분리되어 있으며[7], Espree 을 통해 얻은 AST 노드를 입력 받아 각각의 토큰을 분석하는 방식으로 동작한다. 현재 과제에서는 eslint 의 동작을 참고하여 1) 각 코드를 파싱하고 2) 생성된 AST/CST 에 대한 패턴을 분석하기 위해 파서 라이브러리를 이용할 예정이다.

현재 과제에서 vscode.dev 기반 lint 플러그인에 대해 요구하는 사항을 나타내면 다음과 같다.

- 구현 환경 : vscode.dev
- 수행하는 동작
  - 패턴 리스트 저장
  - 패턴 위반 여부 분석
- 분석 대상 언어 : C/C++, Java, Python
- 동작 방식
  - 파서 라이브러리를 이용하여 대상 언어 코드를 AST/CST 형식으로 파싱
  - 파싱된 트리를 기반으로 패턴 분석

---

[1] visual studio code for web 검색 결과 : https://bit.ly/3Nt95xj

[2] https://code.visualstudio.com/blogs/2021/10/20/vscode-dev

[3] https://en.wikipedia.org/wiki/Lint_(software)

[4] https://eslint.org/

[5] https://pypi.org/project/pylint/

[6] https://github.com/eslint/espree

[7] https://github.com/eslint/eslint/tree/main/lib/rules

### 2.1.2 vscode.dev 환경 분석

과제 요구사항 분석에서 언급했듯이 현재 과제는 vscode studio code web 버전, 즉 vscode.dev 환경에서의 lint 프로그램 구현을 목표로 하고 있다. 이때 Microsoft 사에서 제공하는 문서에 따르면 vscode.dev 에서 사용되는 플러그인은 web extension 이라고 부르며 일반적인 vscode 환경에서 사용하는 플러그인에 비해 일부 제약이 존재한다. 이때 web extension guide[8] 및 정보를 탐색하여 분석한 환경 및 제약 정보는 다음과 같다.

- ◯ 일반 확장과는 달리 browser 속성에 의해 정의된다.
- ◯ 스크립트는 브라우저의 Web Worker[9] 환경의 웹 확장 호스트에서 실행되므로 nodejs 환경에서 구동되는 일반 확장과 일부 차이점이 존재한다.
  - ■ 'fs', 'path', 'process' 등 nodejs 기반의 API 를 이용할 수 없다.
  - ■ 타 모듈에 접근하는 것을 허용하지 않으며, Worker global scope 에서 사용 가능한 importScripts 을 통한 모듈 사용도 불가능하다. 따라서 코드는 webpack 등을 통해 단일 파일 형태로 패키징 되어야 한다.
  - ■ 작업 공간 (유저의 파일이 있는 공간) 은 웹 상의 File System Access API[10] 을 이용하여 접근한다. 확장 공간의 파일은 코드 상에서 ExtensionContext 을 통해 접근한다.
  - ■ File System 은 File System Provider 인터페이스를 구현하는 경우 fs 을 통해 접근할 수 있다[11]. 웹 확장에서는 File System Access API 을 이용하여 직접 해당 인터페이스를 구현하는 경우 사용할 수 있을 것으로 판단된다.[12]
- ◯ JavaScript 및 Web Assembly 기반의 코드만 실행할 수 있다. 따라서 JavaScript 혹은 C/C++/Rust 코드를 Web Assembly 로 컴파일 한 경우에 대해서만 사용 가능하다.
- ◯ vscode 의 확장은 client 및 server 가 통신하는 형식으로 구현된다.[13] 웹 확장에서 서버는 Web Worker 환경에서 실행되어 위 언급한 제약 조건에 해당된다.

따라서 vscode.dev 환경에서는 node js 환경에 의존하지 않는 순수 JavaScript 라이브러리 혹은 Web Assembly 로 컴파일 가능한 종류의 코드만 사용할 수 있다. 만약 유저의 파일 시스템에 접근하고 싶은 경우, File System Provider 인터페이스를 직접 구현해야 한다.

---

[8] https://code.visualstudio.com/api/extension-guides/web-extensions

[9] 웹의 window 환경과 다른 전역 컨텍스트에서 동작하는 스레드. DOM, window 등 일부 메서드 및 속성은 사용할 수 없다는 제약 조건이 있다. https://developer.mozilla.org/ko/docs/Web/API/Web_Workers_API

[10] https://developer.mozilla.org/en-US/docs/Web/API/File_System_Access_API

[11] https://stackoverflow.com/questions/50198150/how-to-work-with-filesystemprovider-file-systems-in-visual-studio-code

[12] FileReader API : https://developer.mozilla.org/en-US/docs/Web/API/FileReader

[13] https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

### 2.1.3 요구사항 정리

전체 요구사항 및 제약 조건을 정리하면 다음과 같다.

- 구현 환경 : vscode.dev
- 수행하는 동작
  - ■ 패턴 리스트 저장
  - ■ 패턴 위반 여부 분석
- 분석 대상 언어 : C/C++, Java, Python
- 동작 방식
  - ■ 파서 라이브러리를 이용하여 대상 언어 코드를 AST/CST 형식으로 파싱
  - ■ 파싱된 트리를 기반으로 패턴 분석
- node js 환경에 의존하지 않는 순수 JavaScript 라이브러리 혹은 Web Assembly 로 컴파일 가능한 종류의 코드만 사용 가능.
- 유저의 파일 시스템에 접근할 수는 있으나, File System Provider 인터페이스를 직접 구현해야 한다.

위와 같은 요구 사항 및 제약 조건을 기반으로 대안과 라이브러리에 대한 탐색을 진행한다.

## 2.2 대안 탐색 및 선택

### 2.2.1 라이브러리 탐색

현재 과제는 eslint 의 동작을 묘사하여 유저의 코드를 파싱하여 생성되는 AST/CST 노드들을 기반으로 패턴을 분석하는 것을 목표로 하고 있다. 이때 해당 파서는 분석 대상 언어인 C, C++, Java, Python 에 대한 언어 분석을 지원해야 하며, JavaScript 혹은 Web Assembly 기반의 코드로 작성되어야 한다.

이때 라이브러리에서 가장 중요한 요건은 '파서' 기능 여부이므로, 우선 파서 기능을 수행하는 라이브러리를 npm[14], GitHub[15] 등 사이트로부터 탐색하였다.

#### espree[16]

espree 는 ECMAScript[17]에 대한 파서 기능을 지원하는 라이브러리로, eslint 라이브러리에서 AST 을 생성하여 패턴을 분석하기 위해 사용된다. 자바스크립트 문법을 이미 내부적으로 보유하고 있는 라이브러리이므로, 다른 언어에 대한 파싱 기능은 지원하지 않는다.

#### Chevrotain[18]

Chevrotain 은 자바스크립트 언어 기반의 라이브러리로, 파서 구축을 위한 툴킷 기능을 수행한다. 특정 언어에 대해 미리 생성된 파서는 존재하지 않는 것으로 보이나, 각각의 토큰 및 패턴을 수동으로 지정할 수 있으므로 다양한 언어에 대한 파서를 구현하는데 사용될 수 있다. 실제로 현재 라이브러리를 이용하여 구현한 자바 언어에 대한 Prettier 플러그인이 존재[19]한다.

#### Tree-sitter[20]

Tree-sitter 은 확장성을 위해 C 언어 기반으로 작성된 파서 제너레이터로, JavaScript,(node/ Web Assembly), Python, Java 등 다양한 언어에 대한 바인딩 버전이 있으며, , C, C++, Java, Python 등의 언어에 대한 미리 구현된 파서가 존재한다. 토큰 및 문법 규칙을 설정하여 파서를 제작하거나, 파싱을 통해 생성된 CST 에 대해 검사하고자 하는 패턴을 지정하여 특정 패턴이 존재하는지 여부를 알 수도

---

[14] node package manager. node.js 기반의 패키지 매니저. https://www.npmjs.com/

[15] https://github.com/

[16] https://github.com/eslint/espree

[17] 자바스크립트 언어에 대한 표준. https://en.wikipedia.org/wiki/ECMAScript

[18] https://chevrotain.io/docs/

[19] https://github.com/jhipster/prettier-java

[20] https://tree-sitter.github.io/tree-sitter/

있다. 웹 개발 환경으로 채택되는 ATOM IDE 에서 코드를 하이라이팅할 때 사용되고 있다.[21]

### 2.2.2 탐색된 라이브러리 비교 및 선택

탐색된 라이브러리들을 다음과 같이 비교했다. 현재 라이브러리들은 모두 자바스크립트 기반의 라이브러리이므로 vscode.dev 상에서 사용 가능할 것으로 보인다.

| 라이브러리 | 파싱 대상 언어 | 파서 생성 | 패턴 지정 |
|---|---|---|---|
| espree | ECMAScript | X | O |
| Chevrotain | 미리 존재 X | O | O |
| Tree-sitter | C, C++, Java, Python... | O | O |

espree 라이브러리는 현재 상용 중에 있는 eslint 라이브러리의 중추 역할을 하는 ECMAScript 에 대한 파서로, AST 을 얻거나 패턴을 지정하는 동작이 가능하지만 현재 과제에서 대상이 되는 C, C++, Java, Python 등의 언어에 대한 파싱 기능은 지원하지 않으며 파서를 생성할 수도 없으므로 사실상 사용할 수 없다.

Chevrotain 은 파서 구축을 위한 툴킷 기능을 수행하는 라이브러리로, 상용 중인 prettier-java 플러그인에서 Syntax Tree 을 생성하는데 사용되고 있다. 특정 언어에 대한 파서가 미리 존재하지는 않지만, 파서를 생성할 수 있으며, 패턴 지정 및 검사도 가능하므로 필요한 경우 C, C++, Python, Java 언어에 대한 토큰 및 문법 규칙을 기반으로 각 언어에 대한 파서를 생성하는 경우 현재 과제에서도 사용할 수 있을 것으로 보인다.

Tree-sitter 은 확장성을 위해 C 언어 기반으로 작성된 파서 제너레이터로 과제에서 제시한 C, C++, Java, Python 언어에 대한 미리 구현된 파서가 존재한다. 또한 파싱 결과로 생성된 CST 에 대해 패턴을 라이브러리 측면에서 지정하여 분석할 수 있으므로 제시된 라이브러리들 중 현재 과제에 가장 적합하다고 볼 수 있다.

따라서 현재 과제는 Tree-sitter 의 미리 구현된 파서 및 지원하는 패턴 기능을 수행하여 lint 기능을 구현할 예정이다.

### 2.2.3 선택된 라이브러리 분석

현재 선택된 라이브러리는 Tree-sitter 이다. 공식 홈페이지[22]에 따르면 Tree-sitter 은 Web Assembly 기반의 JavaScript 바인딩을 지원하는데, 해당 버전은 npm 상에서 web-tree-sitter[23] 이라는 명칭을 통해 구분되어 나타난다. node_modules 경로에서 실제 해당 라이브러리의 위치를 찾아가면 실제

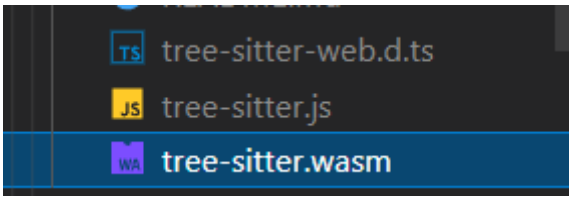
---

[21] https://flight-manual.atom.io/hacking-atom/sections/creating-a-grammar/

[22] https://tree-sitter.github.io/tree-sitter/

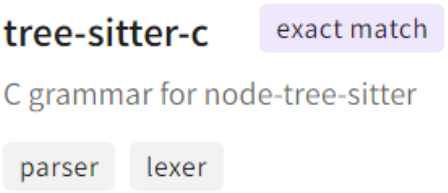
[23] https://www.npmjs.com/package/web-tree-sitter

라이브러리가 tree-sitter**.wasm** 의 형태라는 사실을 볼 수 있는데, 현재 라이브러리 자체도 Web 에서 실행될 수 있도록 Web Assembly 으로 변환되었음을 알 수 있다.




**1 tree-sitter.wasm**

각 언어에 대한 미리 구현된 파서는 tree-sitter 라이브러리와 독립되어 존재하며, npm 을 통해 tree-sitter-[언어이름] 형태로 설치하여 사용할 수 있다. 이때 해당 파서들을 web-tree-sitter 환경에서 이용하기 위해서는 해당 라이브러리 자체를 Web Assembly 의 형태로 변환해야 한다.




**2 C 언어에 대한 tree-sitter 파서**

web-tree-sitter 의 사용법은 대략 다음과 같다.

```
import * as Parser from 'web-tree-sitter'; // 파일 가져오기

await Parser.init({
            locateFile() {
                return uri;
            }
        }); // tree-sitter.wasm 파일 로드

const parser = new Parser(); // 파서 생성

const lang = await Parser.Language.load(uri); // 특정 언어 가져오기

parser.setLanguage(lang); // 특정 언어에 대해 파서 설정


const tree = parser.parse(str); // 입력 문자열 (str)을 파싱하여 CST 생성
const query = lang.query(str_query); // 쿼리 문자열 입력

const qc = query.captures(tree.rootNode); // 쿼리에 대한 결과 배열 가져오기
// 이후 결과를 분석
```

web-tree-sitter 에서는 Parser, Language, Query, QueryCapture 클래스를 이용한다.

- Parser : 실제 tree-sitter 라이브러리와 관련된 메인 모듈로, web-tree-sitter 의 경우 tree-

sitter.wasm 파일을 init( ) 함수를 통해 로드한 후 사용할 수 있다.

- 특정 언어에 대해 파싱을 원하는 경우, setLanguage( ) 메서드를 이용하여 해당 언어에 대한 Language 을 설정한다.
- CST 을 얻고 싶은 경우 parse(str) 메서드를 이용하여 파싱을 진행한다.

- Language : tree-sitter-[언어이름] 에 대응되는 클래스로, 실제 언어에 대한 파싱 규칙 등 다양한 정보를 가지고 있다.
  - query(str) 메서드를 이용하여 입력 받은 문자열에 대한 쿼리를 얻을 수 있다.
- Query : Language 클래스의 query(str) 메서드를 이용하여 만든 쿼리이다. CST 에서 어떤 부분이 입력 받은 쿼리와 매칭되는지 알아보기 위해 사용된다.
  - captures(tree.rootNode) 메서드는 Parser 클래스의 parse( ) 메서드를 통해 만든 CST 의 루트 노드를 입력 받고, 현재 쿼리와 매칭되는 부분이 있는지 여부를 반환한다.
  - 입력 받는 쿼리문은 S-expression[24] 형태로 구성된다.[25] @name 형태로 해당 패턴의 이름을 지정할 수 있으며, 실제로 패턴이 발견되는 경우 QueryCapture 클래스의 name 프로퍼티와 대응되므로, 해당 이름을 기반으로 패턴을 검사할 수 있다.
- QueryCapture : Query 에 대해 captures(tree.rootNode) 메서드를 실행했을 때 얻을 수 있는 정보로 name 및 node 프로퍼티를 통해 쿼리 정보를 알려준다.
  - name : 발견된 패턴의 이름. 쿼리 상에 @**name** 형태로 지정된 이름과 동일하다.
  - node : 패턴에 대응되는 CST 상의 노드. 문서 상의 좌표 등 정보를 알 수 있다.

분석된 정보를 활용하여 lint 플러그인을 구성해보자.

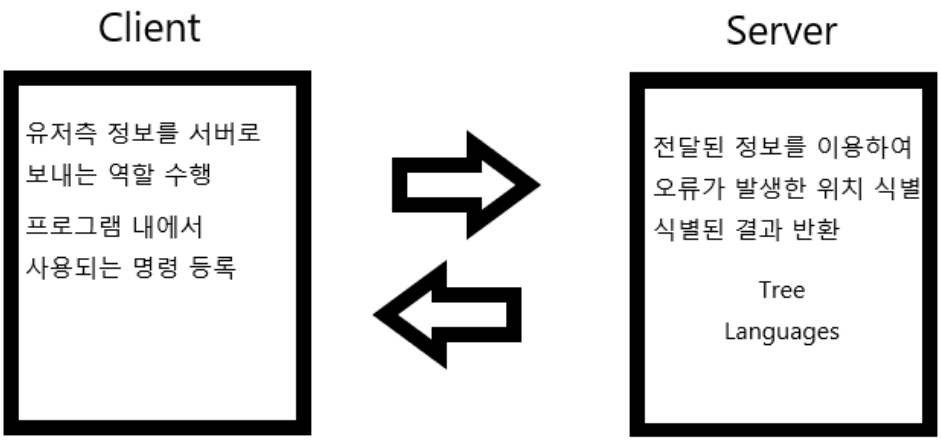
---

[24] https://en.wikipedia.org/wiki/S-expression

[25] https://tree-sitter.github.io/tree-sitter/using-parsers#pattern-matching-with-queries

# 3. 본론 및 개발 내용

## 3.1 전체 프로그램 구조




**3 현재 프로그램의 개략적인 구조. Client-Server 모델을 따르고 있다.**

visual studio code 의 공식 문서에 따르면, 플러그인을 개발할 때 언어 기능을 구현하고 싶은 경우 당사에서 지정한 **Language Server Protocol**[26] 을 통해 코드 에디터와 통신하는 Language Server 을 별도로 두고, 해당 공간에서 여러 처리 작업을 진행할 것을 권장하고 있다.[27] 따라서 현재 과제 역시 LSP 프로토콜 기반의 Client – Server 모델을 통해 개발하였으며, 클라이언트 및 서버 사이의 통신 방식 및 여러가지 설정은 visual studio code 에서 공식적으로 제공하는 튜토리얼을 참고하였다.[28]

LSP 모델에서 클라이언트는 vscode IDE 와 관련된 다양한 작업을 수행할 수 있게 하는 vscode API[29]에 직접적으로 접근할 수 있는데, 이러한 기능에는 특정 명령을 등록하는 등의 동작도 포함된다. 클라이언트 측은 언급한 API 기능을 활용하여 서버와 통신하거나, vscode IDE 을 통해 유저로부터 데이터를 입력 받는 등 다양한 동작을 수행할 수 있어, 커맨드 등을 통해 유저의 반응에 대응하는 코드를 등록하는 구조로 구성되어 있다. 이외로는 서버에 유저나 확장과 관련된 정보를 전달하는 역할을 수행한다.

서버 측은 클라이언트로부터 전달받은 데이터에 대해 일련의 처리를 수행하고, 해당 결과를 클라이언트 측에 반환하여 클라이언트가 유저에게 반응할 수 있도록 한다. 언어와 관련된 모든 기능은 통상적으로 서버 측에서 수행되며, 현재 프로젝트에서 lint 관련 작업은 대부분 서버 측에서 수행된다.

---

[26] https://microsoft.github.io/language-server-protocol/

[27] https://code.visualstudio.com/api/language-extensions/language-server-extension-guide

[28] https://github.com/microsoft/vscode-extension-samples/tree/main/lsp-web-extension-sample

[29] https://code.visualstudio.com/api/references/vscode-api

클라이언트 및 서버는 별개의 패키지로 분리되어 구현된다. 이때 vscode.dev 환경은 다른 파일을 import 할 수 없게 정의되어 있으므로 webpack 을 통해 import 관계를 하나의 파일로 합쳐 사용한다.

## 3.2 공통 개발 사항

### 각 언어에 대한 파서를 Web Assembly 로 컴파일

vscode.dev 환경은 JavaScript 혹은 Web Assembly 코드의 실행만을 허용한다. 따라서 web-tree-sitter 라이브러리에서 사용되는 C, C++, Python 및 Java 언어에 대한 미리 구현된 파서는 wasm 형식으로 컴파일 된 후 web-tree-sitter 의 Language 로 로딩되어야 웹 환경에서 실행될 수 있다. 이때 wasm 파일을 생성하는 과정은 다음과 같다.

1. emscripten 라이브러리를 로컬 영역에 설치한다.

```
# 현재 명령어는 리눅스 기준으로 기술되었다.
$ git clone https://github.com/emscripten-core/emsdk.git
$ cd emsdk
$ git checkout 2.0.17
$ ./emsdk install latest
$ ./emsdk activate latest
$ source ./emsdk_env.sh
```

이때 특징적인 부분은 emscripten 2.0.17 버전을 사용한다는 점이다. 현재 emscripten 의 버전은 3.x 이며, 최근 버전을 사용하여 tree-sitter-[언어이름] 을 wasm 으로 컴파일하는 경우 잘못된 파일을 생성한다. 이때 잘못된 파일을 web-tree-sitter 의 Language 로 로딩하는 경우 에러가 발생하여 원활하게 실행되지 못하는 버그가 존재한다.

이와 관련된 내용은 web-tree-sitter 의 공식 문서에는 나타나지 않으나, github 상에 제시된 issue 중 하나에서 이 현상에 대해 설명하고 있다.[30] wasm 컴파일이 잘못된 파일을 만드는 현상은 emscripten 2.0.18 이상의 버전이 이전 버전과 호환되지 않아 발생하는 문제로 알려져 있으며, 현재 알려진 해결책은 2.0.17 이전 버전을 사용하는 것이다. 따라서 현재 프로젝트에서도 언어 파서를 wasm 으로 컴파일할 때 2.0.17 버전을 사용한다.

2. tree-sitter-cli 및 원하는 언어에 대한 파서를 npm 을 통해 설치한다.

```
$ npm install --save-dev tree-sitter-cli
$ npm install -save tree-sitter-c tree-sitter-cpp tree-sitter-python tree-sitter-java
```

[30] https://github.com/tree-sitter/tree-sitter/issues/1098

3. tree-sitter-cli 을 이용하여 wasm 파일을 생성한다.

```
$ npx tree-sitter build-wasm node_modules/tree-sitter-c
$ npx tree-sitter build-wasm node_modules/tree-sitter-cpp
$ npx tree-sitter build-wasm node_modules/tree-sitter-python
$ npx tree-sitter build-wasm node_modules/tree-sitter-java
```

4. 생성된 파일을 적절한 위치로 이동시켜 관리한다. 위 과정은 단순히 wasm 파일을 얻기 위한 것이므로, 모든 작업을 마친 이후에는 wasm 파일을 제외한 프로젝트 파일을 모두 제거하도록 한다.

### ILint, ILangLint 인터페이스

ILint 인터페이스는 각각의 린트 규칙을 의미하는 인터페이스다. eslint 의 린트 파일은 현재 린트 규칙의 타입, 메시지, 매칭되는 패턴 등의 다양한 정보를 담고 있다.[31] 현재 프로젝트의 린트 규칙 역시 이러한 구조를 묘사하는 ILint 인터페이스를 두고 있다.

```
export interface ILint {
    /**메시지 타입 */
    type : 'error'|'hint'|'information'|'warning';
    /**출력할 메시지 */
    message: string;
    /**tree-sitter query 에 대응되는 쿼리 */
    query: string;
    /** query 에서 잡히는 이름. 해당 이름을 기반으로 나중에 동작하며, 쿼리상의
이름과 같아야 함*/
    node_name : string;
}
```

- type : 현재 린트 규칙의 타입. 서버 측에서 다시 설명한다.
- message : 린트 패턴이 감지되었을 때 출력할 메시지.
- query : tree-sitter 에서 특정 패턴을 감지하는데 사용되는 S-expression. 쿼리의 이름은 @name 형식으로 지정되며, 현재 인터페이스의 node_name 과 대응되어야 한다.
- node_name : query 에 대응되는 노드가 선택되었을 때 나타나는 노드의 이름. 서버 측에서는 node_name 을 키로 사용하여 린트 규칙을 탐색하기 때문에 쿼리 상의 명칭과 동일해야만 정상적으로 동작한다.

ILangLint 인터페이스는 타겟이 되는 언어 및 해당 언어에 대한 린트 규칙을 묶어 놓은 형태로, 서버

[31] https://eslint.org/docs/developer-guide/working-with-rules#rule-basics

상에 린트 규칙을 전달할 때 사용된다.

```
export interface ILangLint {
    /** 타겟이 되는 언어 */
    target : string;
    lints : ILint[];
}
```

## 3.3 클라이언트 측 개발

클라이언트는 특정 명령을 등록하거나, 유저 측에서 전달된 정보를 서버로 넘기는데 사용된다. 클라이언트 코드의 엔트리는 activate 함수로, 클라이언트에서 수행해야 하는 모든 기능은 이 함수 내에서 작성해야 한다.

```
export function activate(context: vscode.ExtensionContext) {
    const languages = ['python', 'java', 'c', 'cpp'];

    const documentSelector = languages.map(lang => { return { language:
lang }; });
    const InitOptions = getInitOptions(context);

    const clientOptions: LanguageClientOptions = {
        documentSelector,
        synchronize: {},
        initializationOptions: InitOptions
    };
```

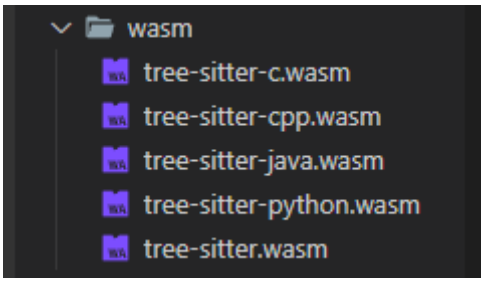
클라이언트는 유저 혹은 확장 환경의 정보를 가지고 있다. 이때 클라이언트는 서버로 LanguageClientOptions 인터페이스를 통해 해당 환경 정보를 초기에 전달할 수 있다. 현재 프로그램에서는 getInitOptions 을 통해 서버에 전달할 정보를 생성한 후, 해당 정보를 clientOptions 에 적재하고 있다.

**getInitOptions**

```
export const getInitOptions = (context: vscode.ExtensionContext) => {
    const _treeuri = vscode.Uri.joinPath(context.extensionUri,'wasm/tree-
sitter.wasm');
    const treeSitterWasmUri = 'importScripts' in globalThis ?
_treeuri.toString() :  _treeuri.fsPath;
    const paths = {
        c: 'wasm/tree-sitter-c.wasm',
        cpp:'wasm/tree-sitter-cpp.wasm',
        python:'wasm/tree-sitter-python.wasm',
        java:'wasm/tree-sitter-java.wasm'
    };

    const lang_uri = new Map<string,string>();

    for(const [lang, uri] of Object.entries(paths))
```

```
    {
        const pathuri = vscode.Uri.joinPath(context.extensionUri,uri);
        const str_uri = 'importScripts' in globalThis ?
pathuri.toString() :  pathuri.fsPath;
        lang_uri.set(lang,str_uri);
    }

    return {
        treeSitterWasmUri,
        lang_uri
    };
};
```

서버는 vscode API 에 직접적으로 접근할 수 없다. 이때 web-tree-sitter 은 Parser 및 Language 객체를 생성하기 위해 wasm 파일이 존재하는 위치를 요구하므로, 클라이언트 측에서 vscode.ExtensionContext 을 통해 확장 환경에 존재하는 wasm 파일의 위치를 서버 측에 제공해야 한다. 현재 함수는 vscode.Uri.joinPath 메서드를 통해 클라이언트 측에서 실제 파일의 경로를 구하고, 이를 객체 형태로 반환한다. 참고로, 모든 wasm 파일은 프로젝트 루트 기준 wasm 폴더에 존재한다.




**4 wasm 파일들을 모아둔 모습**

### createWorkerLanguageClient

```
function createWorkerLanguageClient(context: vscode.ExtensionContext,
clientOptions: LanguageClientOptions) {

    const serverMain = vscode.Uri.joinPath(context.extensionUri,
'dist/server.js');
    const worker = new Worker(serverMain.toString(true));
    return new LanguageClient('web lint', 'web lint', clientOptions, worker);
}
```

vscode 환경 분석 단계에서 vscode.dev 의 서버는 Worker API 환경에서 구동된다고 언급했다.

createWorkerLanguageClient 함수는 서버 파일을 Worker API 상에 등록한 후, LSP 을 이용한 client-server 모델을 구축하는데 사용된다. vscode.Uri.joinPath 메서드를 이용하여 webpack 에 의해 가공된 'dist/server.js' 의 경로를 찾은 후, 해당 경로를 이용하여 서버를 Worker API 상에 적재한다. 이후 LanguageClient 클래스에 해당 worker 객체를 전달하여 language 기능을 위한 클라이언트를 생성 및 반환한다.

LanguageClient 클래스의 처음 두 인자는 각각 클라이언트의 id 및 name 을 의미하며, 클라이언트를 식별하는데 사용된다. 현재 과제의 주제인 web lint 로 채웠다.

### 설정 등록

```
const client = createWorkerLanguageClient(context, clientOptions);
    const disposable = client.start();
    context.subscriptions.push(disposable);
```

activate 함수 내에서 모든 동작은 push 메서드를 이용하여 등록한다. 해당 동작들은 특정 조건을 만족했을 때 실행되도록 구현되며 push 을 통해 구독 목록에 포함되는 경우에만 실행된다. 위 코드에서는 생성된 클라이언트를 등록하고 있다.

### getUserInput & web-lint.setlint

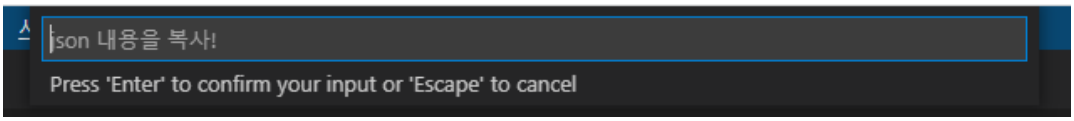
```
async function getUserInput(context: vscode.ExtensionContext) {
    const input = await vscode.window.showInputBox({ placeHolder: 'json 내용을
복사!' });
    const json_lint : ILangLint[] = JSON.parse(input); // ILint 타입으로 파싱.
    // 이거에 안맞는거 들어오면 에러로 취급되서 진행 자체가 안됨.

    for(const lang_lint of json_lint) // 각 언어에 대한 정보 저장.
    {
        await context.globalState.update(`weblint-${lang_lint.target}`,
lang_lint.lints);
    }
    await vscode.commands.executeCommand('web-lint.loadlint');
}
```

**getUserInput** 함수는 사용자로부터 lint 설정을 텍스트 형태로 입력 받는 동작을 구현한다.

구현 초기 단계에서는 설정 파일을 json 형태로 유저의 파일 시스템에 저장한 후 플러그인이 실행되면 해당 파일들의 정보를 읽어오는 방식으로 구현하고자 했다. 이때 공식 문서에서는 유저의 파일 시스템을 vscode.workspace.fs 을 통해 접근할 수 있다고 설명했으나, 실제로는 대응되는 File System Provider 인터페이스를 개별적으로 구현한 경우에만 사용할 수 있었다. 따라서 파일 시스템 기반의 방식으로는 린트 규칙을 저장해 두는 것이 힘들다고 판단하여 다른 방법을 탐색했다.

두번째 방법으로는 린트 규칙을 텍스트 형식으로 유저로부터 직접 입력 받고, 해당 정보를 vscode IDE 내에 저장해 두는 방식을 생각했다. 다행히도 두가지 방법 모두 vscode API 상에서 지원하는 기능이었다.

- showInputBox: 유저로부터 메시지를 입력 받기 위한 창을 vscode IDE 환경에 띄운다. 유저가 입력한 메시지는 string 타입으로 반환된다.




**5 InputBox 의 모습**

- Memento API : 현재 vscode IDE 상에 정보를 저장할 수 있도록 지원하는 API 이다. key-value 구조를 통해 정보를 get 하거나 update 할 수 있다. 현재 코드에서는 **globalState** 라는 이름으로 나타나고 있으며, 전역 환경에 정보를 저장한다.

vscode 에서 지원하는 위 두가지 API 를 통해 **패턴 리스트 저장** 조건을 구현할 수 있었다.

showInputBox API 을 통해 입력 받은 문자열을 ILangLint[ ] 구조로 파싱한 후, 각 언어별로 다른 저장소에 업데이트한다. 이때 globalState 상에 저장된 정보들은 웹 브라우저의 캐시를 제거하는 경우 함께 제거된다는 특징을 가지므로, 캐시를 제거하지 않는 한 계속 린트 규칙이 유지된다.

린트 규칙을 모두 globalState 상에 작성한 이후에는 등록된 vscode 커맨드 중 하나인 'web-lint.loadlint' 을 실행한다.

```
    const disposable2 = vscode.commands.registerCommand('web-lint.setlint',
async () => {
        await getUserInput(context);
        vscode.window.showInformationMessage('린트 설정 성공');
    });
    context.subscriptions.push(disposable2);
```

현재 동작은 registerCommand 에 의해 **'web-lint.setlint'** 으로 등록되었다.

### getqueries & web-lint.loadlint

```
async function getqueries(context: vscode.ExtensionContext, langs : string[]){
    const all_lints : ILangLint[] = [];

    for(const lang of langs) // 각 언어에 대한 정보 저장.
    { // 언어 가져오기
        const lints : ILint[] = await context.globalState.get(`weblint-
${lang}`);
        if(lints)
        {
            all_lints.push({target: lang, lints});
        }
    }
    return all_lints; // 린트 모두 반환.
}
```

**getqueries** 함수는 globalState 상에 저장된 쿼리 정보를 취합하여 반환한다. 현재 프로그램은 globalState 상에 린트 규칙들을 저장할 수 있다. 이때 프로그램이 동작하는 시점에 해당 린트 규칙들을 가져와 서버로 보냄으로써 서버의 린트 규칙을 활성화할 필요가 있으므로 쿼리를 가져오는 함수를 따로 분리하여 구성하였다.

```
const disposable3 = vscode.commands.registerCommand('web-lint.loadlint', async
() => {
        const lints = await getqueries(context,languages);
        client.sendNotification('lint-config-change', lints);
        vscode.window.showInformationMessage('린트 로드 성공');
        // 클라이언트 측에서 서버측으로 메시지를 보내도록 만든다.
    });
    context.subscriptions.push(disposable3);
```

**'web-lint.loadlint'** 명령에 대응되는 함수는 getqueries 을 통해 린트 규칙들을 가져온 후, 서버에 해당 정보를 전달하는 동작을 가진다. 서버로 정보를 전달하는 것은 sendNotification 함수를 이용한다.

- sendNotification : 서버로 정보를 전달할 때 사용되는 함수로, 2 개의 인자를 가진다.
  - method : 클라이언트 및 서버에서 정보를 공유할 때 사용되는 이름을 지정한다. 위 코드의 경우 'lint-config-change' 의 이름을 통해 클라이언트 및 서버가 서로 소통할 수 있게 된다. 서버 측에서는 'lint-config-change'에 대해 대기하는 코드를 작성하여 정보를 제공받을 수 있다.
  - params : json 으로 파싱될 수 있는 객체. 클라이언트가 전달한 객체 그대로 서버에서 사용할 수 있다.

현재 함수에서는 sendNotification 을 통해 린트 규칙들을 서버로 넘긴 후, 린트가 로드되었다는 메시지를 showInformationMessage 함수를 통해 출력한다.


**6 showInformationMessage 함수를 통해 IDE 상에서 출력한 메시지**

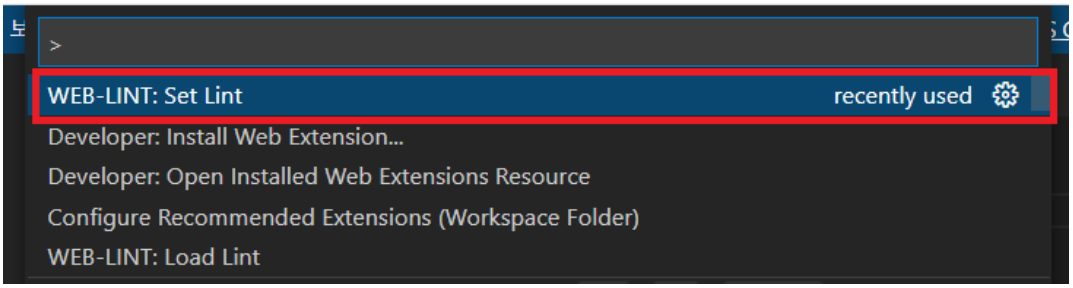
```
    client.onReady().then(() => {
        vscode.window.showInformationMessage('클라이언트 준비. 이제 set lint
명령으로 린터를 설정하세요!');
        vscode.commands.executeCommand('web-lint.loadlint')
        .then();
    });
}
```

전체 설정을 등록하면 client.onReady 함수를 통해 클라이언트를 로딩한다. 클라이언트가 로딩되면 'web-lint.loadlint' 명령을 실행하여 globalState 상에 저장된 린트 규칙을 서버에 전달함으로써 서버에 쿼리 객체들을 생성한다.

이때 사용자 측에서 위 등록한 web-lint.loadlint 및 web-lint.setlint 명령을 수행하기 위해서는 해당 명령들이 package.json 파일의 contributes -> commands 에 등록되어야 한다.

```
"commands": [
            {
                "command": "web-lint.setlint",
                "title": "Set Lint",
                "category": "WEB-LINT"
            },
            {
                "command": "web-lint.loadlint",
                "title": "Load Lint",
                "category": "WEB-LINT"
            }
```

**"command"**는 위에서 제시한 커맨드에 해당한다. 커맨드는 vscode 내에서 사용되는 명령이므로, 사용자 측에서 해당 명령을 사용할 때는 "title"에 등록된 이름을 통해 접근한다. "category"는 명령의 큰 분류를 지정한다.




**7 사용자 측에서 커맨드 팔레트를 통해 명령에 접근 가능한 모습**

## 3.4 서버 측 개발

서버는 사용자 및 vscode IDE 로부터 정보를 받고, 해당 정보들을 기반으로 CST 을 만들어 사용자 측으로부터 전달받은 쿼리를 시험한 후, 쿼리에 들어 맞는 사항이 있는 경우 해당 결과를 반환하는 역할을 수행한다.

### Tree 클래스

Tree 클래스는 사용자로부터 전달된 유저의 코드를 tree-sitter 의 CST 로 파싱하는 역할을 수행한다. 내부적으로 파서 및 트리에 대한 static 프로퍼티를 가지며, static 클래스처럼 동작한다.

- _parser : web-tree-sitter 에서 실제 파서 역할을 수행하는 모듈
- _tree : 코드를 파싱하여 생성된 CST

```
export class Tree {
    private static _parser: Parser;
    private static _tree: Parser.Tree | null;

    static async init(uri: string) // 트리시터 초기화
    {
        await Parser.init({
            locateFile() {
                return uri;
            }
        });

        this._parser = new Parser(); // 파서 생성
    }
```

**init** 메서드는 tree-sitter.wasm 파일의 주소를 입력 받아 web-tree-sitter 의 파서 기능을 초기화한다. 파서를 초기화하는데 사용되는 Parser.init 은 인자로 emscripten 생성 코드에 대해 전달할 내용을 가지며, locateFile 은 tree-sitter.wasm 의 위치를 전달하는데 사용되고 있다.[32]

```
    static attach_lang(lang_Id: string) {

        const lang = Languages.getLang(lang_Id);
        this.clear_tree(); // 트리 초기화
        this._parser.setLanguage(lang); // 언어 갈아 끼기
    }
```

**attach_lang** 메서드는 Languages 클래스로부터 특정 언어에 대한 파서를 입력 받아 현재 파서를 해당 언어에 대해 지정하는 역할을 수행한다.

---

[32] https://emscripten.org/docs/api_reference/module.html

```
    static parse(str: string) {
        this._tree = this._parser.parse(str);
    }
```

parse 메서드는 현재 지정된 언어에 대해 유저의 코드를 파싱하고, 트리에 저장하는데 사용된다.

```
    static get_tree() {
        return this._tree;
    }

    static clear_tree() {
        this._tree = null; // 트리 초기화
    }
}
```

**get_tree / clear_tree** 메서드는 _tree 에 대한 getter / clear 동작을 수행한다. 외부에서는 get_tree 을 이용해야만 생성된 트리에 접근할 수 있다.

### Languages 클래스

Languages 클래스는 클라이언트 측에서 web-lint.setlint 명령을 수행할 때 전달하는 ILangLint[ ] 을 받아 각 언어에 대한 쿼리들을 생성하여 저장해두고, 필요한 경우 전달된 CST 을 이용하여 매칭된 쿼리를 반환하는 역할을 수행하는 클래스로, 언어와 관련된 정보 저장 및 쿼리 캡쳐 반환을 주 역할로 한다. 현재 지원하는 언어는 조건에 따라 c, cpp, python, java 이지만, 클라이언트 측에서 일부 코드 및 특정 언어에 대한 wasm 파일만 추가하면 충분히 확장할 수 있도록 설계되었다.

- _languages : 현재 프로젝트에서 사용되는 언어(C, C++, java, python)을 저장해두는 Map 자료구조이다.
- _queries : 각각의 언어에 대한 쿼리 목록.
- _linter : 유저가 입력한 ILint[] 목록으로, 내부적으로 쿼리가 정상적으로 생성되지 않는 상황이 다수 존재하여 해당 상황에 쿼리를 다시 생성할 수 있도록 저장하고 있다.

```
export class Languages {
    private static readonly _languages = new Map<string, Parser.Language>();
    private static readonly _queries = new Map<string, Parser.Query[]>();
    private static _linter: Map<string, ILint[]>;
    /**
     * wasm을 읽어서 각 언어에 대한 파서를 생성한다.
     */
    public static async init(lang_uri: Map<string, string>) {
        for (const [lang, uri] of lang_uri) {
            console.log(uri);
            const tree_lang = await Parser.Language.load(uri);
            this._languages.set(lang, tree_lang);
        } // language 들을 삽입한다.
    }
```

**init** 메서드는 각 언어에 대한 주소를 얻은 후, 각각의 주소에 대응되는 언어 파서를 wasm 파일로부터 로드 및 _languages 내에 저장하는데 사용된다.

```
    public static getLang(lang_Id: string) {
        const lang = this._languages.get(lang_Id);
        if (lang) {
            return lang;
        }
        else {
            console.warn(`There is no lang ${lang_Id}`);
            return null;
        }
    }
```

**getLang** 메서드는 요청된 특정 언어를 외부에 전달한다. 현재 프로젝트에서는 Parser 의 언어를 지정하기 위해 사용된다.

```
    public static setLintQueries(linters: Map<string, ILint[]>) {
        this.setLinter(linters);
        this.setQueries();
    }
```

클라이언트로부터 전달받은 쿼리 규칙들을 내부에 저장하고, 쿼리 객체를 생성한다.

```
    public static setQueries() {
        this._queries.clear(); // 이전에 있었던 내용을 모두 날린다.
        for (const [target, lints] of this._linter) {
            const queries: Parser.Query[] = [];

            const lang = this.getLang(target); // 언어 가져오기
            if (lang) {
                for (const lint_info of lints) {
                    try {
                        const query = lang.query(lint_info.query);
                        queries.push(query);
                    }
                    catch(e) {
                        console.log(e);
                        continue;
                    }
                }
                this._queries.set(target, queries);
            }
        }
    }
```

**setQueries** 메서드는 _queries 프로퍼티에 각각의 언어에 대응되는 쿼리 객체를 생성하여 저장하는데 사용된다. 클래스 내부의 _linter 객체에 저장된 ILint 객체의 query 프로퍼티를 추출하여 각각 Query 클래스를 생성한 후 배열 형태로 저장한다.

```
    public static getQueries(lang_Id: string) {
        const query = this._queries.get(lang_Id);

        if (query) {
            return query;
        }
        else {
            this.setQueries(); // 쿼리 설정
            return this._queries.get(lang_Id);
        }
    }
```

getQueries 메서드는 특정 언어에 대한 쿼리를 전달하는데 사용된다. 실제로 프로그램을 테스팅 하는 상황에서 간헐적으로 쿼리가 생성되지 않는 현상이 발생했기 때문에, if 문을 이용하여 쿼리가 없는 경우 쿼리를 다시 생성할 수 있도록 구현했다.

```
    public static getQueryCaptures(lang_Id: string, tree: Parser.Tree | null){
        const queries = this.getQueries(lang_Id);
        const queryCaptures: Parser.QueryCapture[] = [];

        // 대응되는 쿼리와 트리가 존재할 때
        if (queries && tree) {
            // 각각의 쿼리에 대한 캡쳐 결과 삽입.
            for (const query of queries) {
                try {
                    const qc = query.captures(tree.rootNode);
                    queryCaptures.push(...qc); // 쿼리가 잘못될 수도 있음.
                }
                catch(e) {
                    console.log(e);
                }

            }

            return queryCaptures;
        }
        else {
            return null;
        }
    }
```

getQueryCaptures 은 특정 언어에 대해 생성된 CST 을 query.captures 을 통해 매칭시킨 후 매칭된 결과를 얻어 반환하는 역할을 한다. CST 또는 쿼리가 존재하지 않으면 제대로 된 결과를 생성할 수 없으므로 if 문을 통해 체크하고, 대응되는 언어에 대해 쿼리를 모두 비교하여 대응되는 값을 하나의 배열 형태로 모아 반환하고 있다.

**Server**

서버가 클라이언트와 통신하거나, 특정 동작을 등록하는 코드가 주를 이루는 파일이다.

```
const messageReader = new BrowserMessageReader(self);
const messageWriter = new BrowserMessageWriter(self);

const connection = createConnection(messageReader, messageWriter);
```

서버가 클라이언트를 참조하기 위해 사용하는 코드이다. 클라이언트 측에 대한 Reader 및 Writer 을 설정하여 클라이언트 측에서 오는 정보를 읽거나, 클라이언트에게 정보를 전달할 수 있도록 설정하는 것으로, 서버측 코드에서는 connection 객체를 통해 클라이언트와 연결된다.

```
connection.onInitialize(async (params: InitializeParams):
Promise<InitializeResult> => {
    const initData: InitOptions = params.initializationOptions;
    console.log('tree-sitter : ', initData.lang_uri);

    await Tree.init(initData.treeSitterWasmUri); // 트리 시터 초기화
    await Languages.init(initData.lang_uri); // 언어 목록 초기화.

    const capabilities: ServerCapabilities = {
    };
    console.log("server ready");
    return { capabilities };
});
```

**onInitialize** 은 클라이언트가 실행되었을 때 발동한다. 인자로 들어오는 params 은 클라이언트 측에서 LanguageClient 을 설정할 때 전달했던 params 와 대응되므로, 서버에서는 params 정보를 통해 wasm 파일의 주소를 얻을 수 있도록 구현되었다. 현재 코드 상에서는 해당 주소를 기반으로 Parser 및 Language 들을 초기화 하고 있다.

```
connection.onNotification('lint-config-change', (val: ILangLint[]) => {
    console.log(`서버 린터 설정`);

    const temp = new Map<string, ILint[]>();
    val.forEach(
        (lint) => {
            lint.lints.forEach(
                l => {
                    let bef = l.query;
                    let cur = l.query.replace('\\', '');
                    while (bef !== cur) {
                        bef = cur;
                        cur = cur.replace('\\', '');
                    } // \ 이 더 이상 없을 때까지 모두 제거한다.
                    l.query = bef;
                }
```

```
            );
            temp.set(lint.target, lint.lints);
        }
    );
    linter = temp;
    Languages.setLintQueries(temp); // 쿼리 설정.
});
```

onNotification 메서드는 클라이언트가 sendNotification 을 통해 서버에 특정 정보를 전달하는 경우 발동된다. 클라이언트에서는 'lint-config-change' 을 통해 클라이언트가 유저로부터 얻은 린트 규칙들을 서버로 제공했다. 서버에서는 해당 메서드에 대응되는 onNotification 을 지정하여 데이터를 제공받는다.

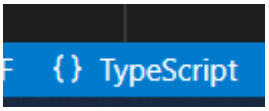
인자로 제공받은 객체는 ILangLint[ ] 타입으로, 언어 및 대응되는 린트 규칙들을 가지고 있다. 이때 json 내에서는 큰따옴표를 사용할 수 없어 ₩" 형태로 표현했기 때문에, 역 슬래시를 제거하는 과정을 거친 후 Languages.setLintQueries 메서드를 통해 각 언어에 대한 쿼리를 생성한다.

```
const documents = new TextDocuments(TextDocument);

documents.listen(connection);
```

IDE 에 활성화된 문서 정보는 TextDocuments 클래스를 통해 받고 있다. 이후 문서와 직접적으로 관련된 정보는 documents 을 통해 얻을 수 있다.

```
documents.onDidChangeContent(async change => {
    const doc = change.document;
    const lang_id = doc.languageId;
    const text = doc.getText();

    Tree.attach_lang(lang_id); // 언어 교체
    Tree.parse(text); // 트리 파싱

    const tree = Tree.get_tree(); // 트리 가져오기
    // 패턴 매칭이 되는 놈이 있으면 찾아서 없앤다는 마인드.

    let problems = 0;
    const diagnostics: Diagnostic[] = [];
    // 패턴에 맞는게 있으면 안됨!

    const captures = Languages.getQueryCaptures(lang_id, tree);
    // console.log(captures);
```

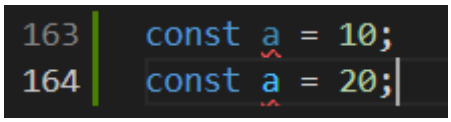
documents.onDidChangeContent 메서드는 현재 vscode IDE 상에 열려 있는 문서에 변화가 발생할 때마다 실행된다. change.document 을 통해 현재 변화된 문서 정보 및 사용된 언어 정보를 얻을 수 있다. 이를 기반으로 문서의 텍스트 정보 및 사용된 언어의 id 를 받아온다. 언어 정보는 IDE 의 우측 하단에 출력되는 것과 같다.



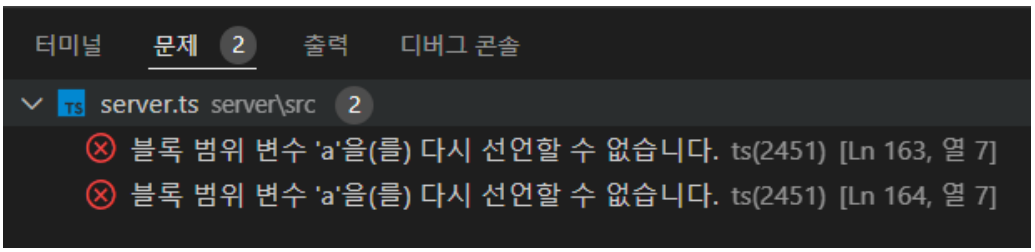

**8 TypeScript 언어를 사용하는 문서**

위에서 얻은 정보를 기반으로 Tree 의 _parser 의 언어를 lang_id 가 가리키는 언어로 교체한 후 언어 파싱을 수행한다. 파싱 결과 생성된 CST 는 Languages 클래스의 getQueryCpatures 에 전달되고, 쿼리를 만족하는 대상 노드를 돌려받는다.

이때, 코드 중 Diagnostic 인터페이스를 볼 수 있다. Diagnostic 인터페이스[33]는 vscode IDE 의 PROBLEMS 패널 및 밑줄을 통해 특정 정보를 알리고 싶은 대상을 가리키는 인터페이스이다. 예를 들어 eslint 의 규칙을 만족하지 않는 경우 그림 9, 10 과 같은 에러 메시지 및 밑줄이 발생할 수 있다. 현재 과제에서는 Diagnostic 을 이용하여 lint 규칙을 IDE 상에 나타낸다.




**9 const 변수 재선언에 대해 에러 메시지를 출력하는 모습**




**10 PROBLEMS 패널에서 해당 에러를 설명해주는 모습**

```
if (captures) // captures 객체가 실제로 존재하는 경우
    {
        for (const cap of captures) {
            problems += 1;
            if(problems > 100) // 최대 문제 개수 제한
            {
                break;
            }
            const name = cap.name;
            const node = cap.node;
            let message = '';
            let severty: DiagnosticSeverity = DiagnosticSeverity.Warning;
            const range = Range.create(node.startPosition.row,
node.startPosition.column, node.endPosition.row, node.endPosition.column);
```

[33] Diagnostic 사용법을 설명하는 vscode 공식 문서 : https://code.visualstudio.com/api/language-extensions/programmatic-language-features#provide-diagnostics

```
            const lints = linter.get(lang_id); // lang_id 에 대응되는 린터
가져오기.
```

captures 객체가 존재하는 경우 ( 쿼리 및 트리가 유효한 경우 ) 각각의 캡처의 정보를 얻는다. 각각의 QueryCapture 객체 cap 은 name 및 node 을 가지는데, name 은 캡처된 쿼리의 이름과 대응되어 해당 이름을 기준으로 린트 메시지를 얻으며, node 는 해당 캡처가 발견된 텍스트 상의 위치 정보를 담고 있으므로 해당 정보를 기반으로 Diagnostic 객체의 위치를 지정한다.

```
const lints = linter.get(lang_id); // lang_id 에 대응되는 린터 가져오기.
            if (lints) { // 린터 객체가 존재할 때
                for (const l of lints) { // 각각의 린터에 대해
                    // console.log(`${name}, ${l.node_name}`);

                    if (name === l.node_name) { // 대응되는 린터를 발견한 경우
                        // 대응되는 린터가 없는 경우는 관여 X
                        message = l.message;
                        switch (l.type) { // 린터의 타입을 읽어 와서 설정.
                            case 'error':
                                severty = DiagnosticSeverity.Error;
                                break;
                            case 'hint':
                                severty = DiagnosticSeverity.Hint;
                                break;
                            case 'warning':
                                severty = DiagnosticSeverity.Warning;
                                break;
                            case 'information':
                                severty = DiagnosticSeverity.Information;
                                break;
                        }

                        // 린터를 문제 창에 출력하기 위해 Diagnostic 객체 생성.
                        const diagnostic: Diagnostic = {
                            severity: severty,
                            range: range,
                            message: message,
                            source: 'web-lint'
                        };
                        diagnostics.push(diagnostic); // 객체를 삽입한다
                        break; // 린터가 중복될 수는 없음.
                    }
                }
            }
        }
    }
    connection.sendDiagnostics({ uri: doc.uri, diagnostics });
```

이후 문서의 언어에 대응되는 lint 규칙을 가져와 캡처된 노드의 이름과 대응되는 린트 객체를 찾는다. 대응되는 린트 규칙은 ILint 인터페이스를 따르는 객체로 해당 린트 규칙에 대한 메시지 및 타입 정보를 가지고 있는데, 해당 정보를 기반으로 현재 캡처된 노드에 대응되는 Diagnostic 객체에 대한 타입 및

메시지를 설정한다.

만들어진 전체 Diagnostic 객체들은 connection.sendDiagnostics 을 통해 클라이언트 측에 전달되며, 클라이언트는 대응되는 문서 및 PROBLEMS 패널에 해당 메시지들을 출력한다.

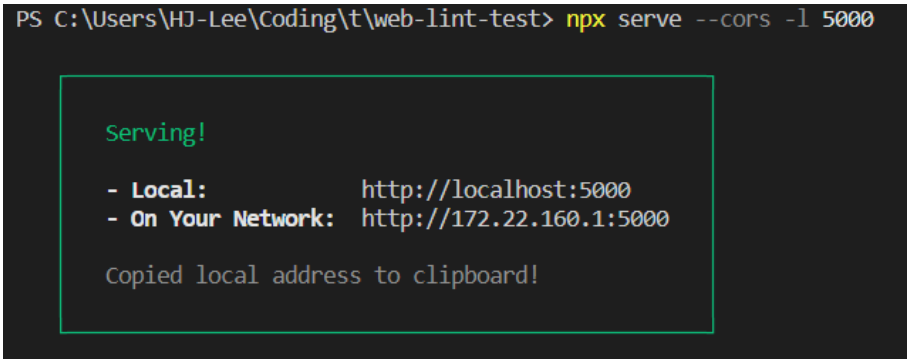
# 4 실험 결과

실험 결과에서는 현재 플러그인의 사용법 및 테스트용 쿼리의 실제 동작 여부를 보이고 있다.

## 4.1 플러그인 이용 방법

현재 과제에서 구현한 플러그인을 실행하는 방법은 vscode 공식 문서에서 제공하는 vscode web extension 테스팅 방식을 사용하고 있다.[34]

1. 동봉된 코드 파일의 압축을 해제한다.
2. 제공된 코드의 루트 폴더로 터미널을 이동시킨다.
3. 터미널 상에 npm install 을 입력하여 디펜던시 파일들을 설치한다.
4. 터미널 상에 **npm run watch** 을 입력하여 webpack 을 실행시킨다.
5. 새로운 터미널에서 **npx serve –cors -l 5000** 을 입력하여 로컬 서버를 연다.




**11 제시된 코드를 실행하면 나오는 화면**

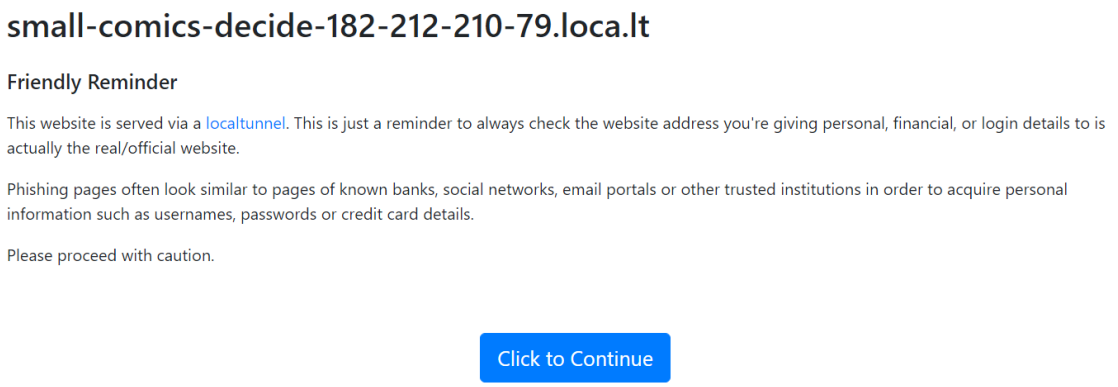
6. 새로운 터미널에서 **npx localtunnel -p 5000** 을 입력하여 로컬 서버를 https 서버에

[34] https://code.visualstudio.com/api/extension-guides/web-extensions#test-your-web-extension-in-on-vscode.dev

연결한다. 터미널 상에 출력된 주소로 이동한 후, 버튼을 누른다.

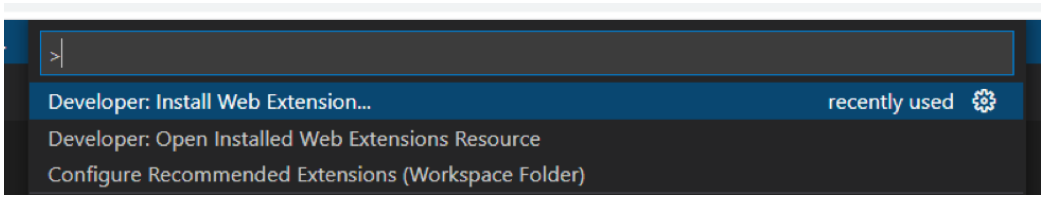
```
PS C:\Users\HJ-Lee\Coding\t\web-lint-test> npx localtunnel -p 5000
your url is: https://small-comics-decide-182-212-210-79.loca.lt
```

**12 제시된 코드를 실행하면 나오는 주소**

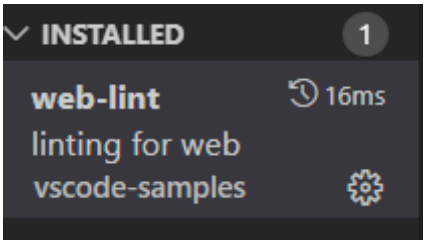



**13 입력된 주소로 이동했을 때 출력되는 메시지와 버튼. 버튼을 클릭해야만 https 로 연결된다.**

7. vscode.dev 으로 이동한다.
8. ctrl + shift + P 을 클릭하여 커맨드 팔레트를 연다.
9. Install Web Extension 을 클릭하여 위에서 이동한 주소를 입력한다. 해당 과정을 마치면 현재 플러그인이 vscode.dev 상에 설치된다.

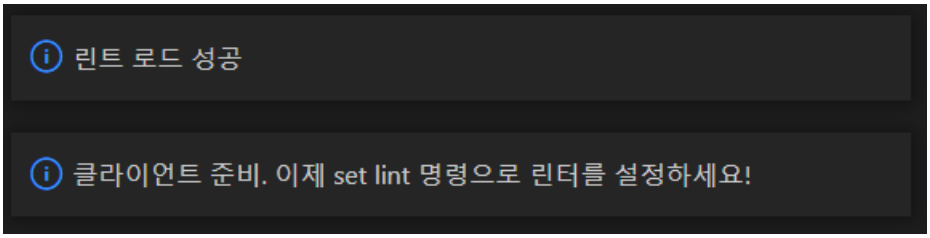



**14 커맨드 팔레트에서 Install Web Extension 을 선택하는 모습**



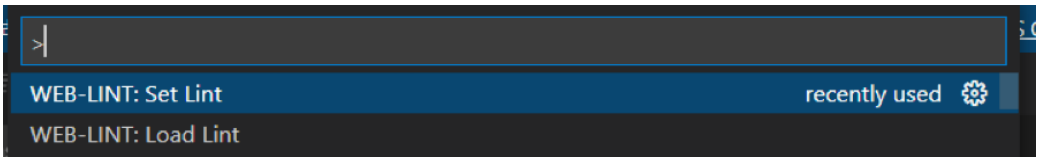

**15 현재 플러그인이 설치된 모습**

10. c, cpp, java, python 파일을 연다. 현재 플러그인은 해당 파일들이 열릴 때 활성화되는 것으로 구현되어 있다.
11. 그림 16 과 같은 메시지가 등장할 때까지 기다린다.




**16 클라이언트 준비 후 등장하는 메시지**

12. 확장 폴더 내에 제공된 test.json 의 내용을 복사한 후 커맨드 팔레트의 set lint 명령을 수행하여 작성한다. 만약 이전에 해당 설정을 로드한 적이 있다면 load lint 명령을 이용하여 브라우저 상에 저장된 설정을 이용할 수도 있다.



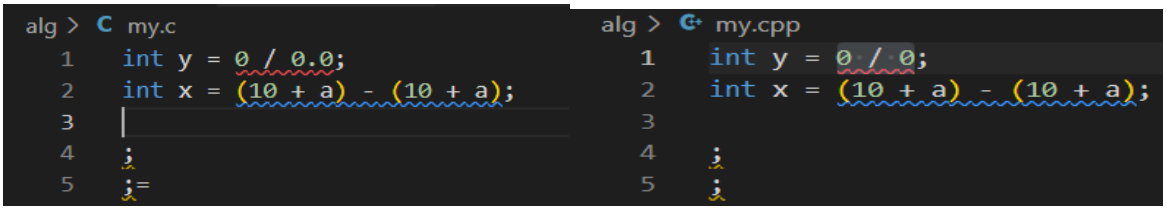

**17 Set Lint 및 Load Lint 명령이 커맨드 팔레트 상에 존재하는 모습**

## 4.2 실행 결과

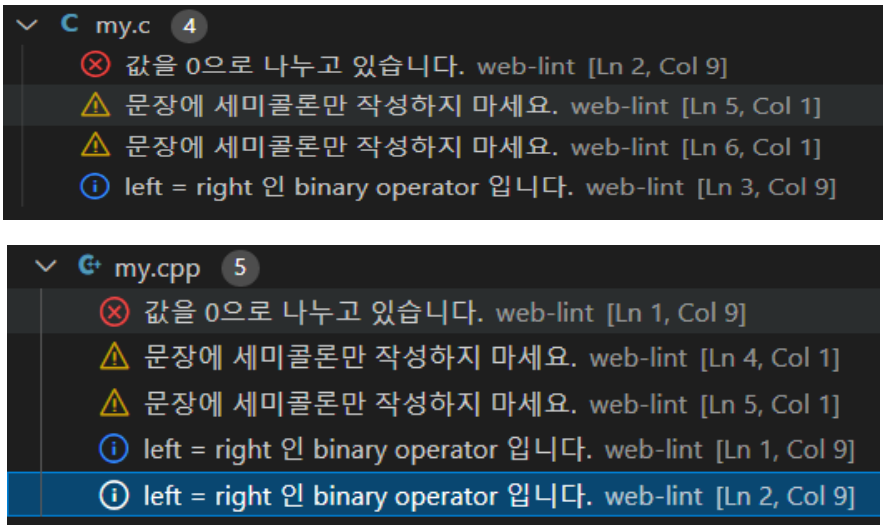
현재 프로젝트 폴더 내에 존재하는 test.json 파일은 언어 당 2 개 이상의 테스트용 린트 규칙을 구현하고 있다. 린트 규칙에 사용되는 S-expression 은 tree-sitter Playground[35]에서 테스트한 후 사용하였다. 만약 새로운 규칙을 추가하고 싶다면 tree-sitter 에서 제공하는 가이드라인을 따르면 된다.

### C / C++

c 언어 및 c++ 언어의 경우 파서 자체는 다르나, 서로 호환되는 성격을 지닌다. 현재 c / cpp 에 대해서는 0 으로 값을 나누는 경우, 세미콜론만 존재하는 경우 및 좌우가 동일한 binary operator 에 대한 린트 규칙이 설정되어 있다. 결과는 그림 18, 19 와 같다.




**18 각 린트 규칙에 대해 밑줄로 표기된 모습**

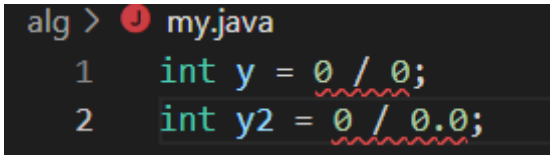



**19 캡처된 린트 규칙에 대한 PROBLEMS 패널의 메시지**

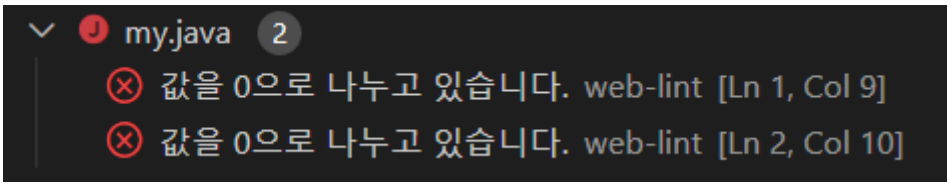
[35] https://tree-sitter.github.io/tree-sitter/playground

### Java

java 언어는 현재 값을 0 으로 나누는 divide 0 에 대한 린트 규칙이 설정되어 있다. java 에 대한 파서는 integer 과 floating_point 을 구분하도록 구현되어 있어 린트 규칙 자체는 2 개로 구성되어 있으나, 동일한 divide 0 상황을 의미한다. 결과는 그림 20, 21 과 같다.



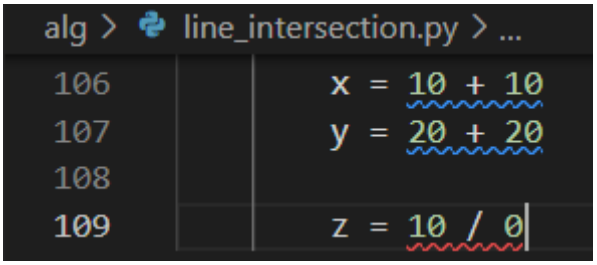

**20 divide 0 에 대한 에러 밑줄**



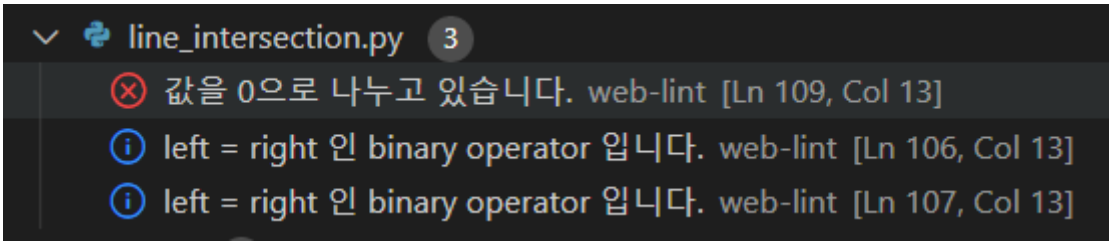

**21 divide 0 에 대한 에러 메시지**

### Python

python 언어는 divide 0 에러 및 좌우가 동일한 operator 에 대한 린트 규칙이 설정되어 있다.




**22 캡처된 린트 규칙에 대한 밑줄**




**23 린트 규칙에 대한 메시지**

## 5. 결론 및 향후 연구

위와 같은 결과를 통해 현재 프로그램의 린트 설정 및 반영이 정상적으로 이루어지고 있음을 알 수 있다. 현재 설정된 린트 규칙들은 tree-sitter 에서 제공하는 S-expression 설명에 기반하고 있으므로, 해당 규칙들이 실제로 동작한다는 것은 tree-sitter 이 제공하는 규칙 조건 하에서 구현 가능한 모든 패턴을 감지할 수 있음을 의미한다. 따라서 현재 플러그인이 꽤 성공적으로 동작한다고 판단할 수 있다.

현재 프로젝트에서 각 언어에 대한 쿼리는 웹 상에 저장된다. 따라서 웹 캐시를 제거하는 경우 쿼리 내역이 함께 제거될 수 있다는 문제점이 존재한다. eslint 와 같은 시중 lint 플러그인의 경우 플러그인이 설치된 폴더에 lint 규칙들이 js 혹은 json 포맷 형태로 미리 저장되어 사용되는 경우가 많아 해당 동작을 그대로 프로젝트에 반영하기에는 어려움이 있었다. 따라서 vscode.dev 환경에서의 파일 시스템 혹은 http 통신 방식을 조금 더 파악하여 lint 규칙을 유저가 직접 지정하지 않고도 사용할 수 있는 방법을 고려하는 것도 좋을 것 같다.

현재 사용된 모든 코드는 본인의 github 주소인 https://github.com/blaxsior/web-lint-test 에 있다.