Implementacja protokołu LSP dla wybranego środowiska zintegrowanego

(Implementation of a LSP protocol
for chosen IDE)

Wiktor Adamski

Praca inżynierska

Promotor: dr Wiktor Zychla

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

30 stycznia 2018 r.

Wiktor Adamski

.....

.....
(adres zameldowania)

.....

.....
(adres korespondencyjny)

PESEL: .....

e-mail: .....

Wydział Matematyki i Informatyki
stacjonarne studia I stopnia
kierunek: informatyka
nr albumu: 272220

Oświadczenie
o autorskim wykonaniu pracy dyplomowej

Niniejszym oświadczam, że złożoną do oceny pracę zatytułowaną *Implementacja protokołu LSP dla wybranego środowiska zintegrowanego* wykonałem/am samodzielnie pod kierunkiem promotora, dr Wiktora Zychli. Oświadczam, że powyższe dane są zgodne ze stanem faktycznym i znane mi są przepisy ustawy z dn. 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (tekst jednolity: Dz. U. z 2006 r. nr 90, poz. 637, z późniejszymi zmianami) oraz że treść pracy dyplomowej przedstawionej do obrony, zawarta na przekazanym nośniku elektronicznym, jest identyczna z jej wersją drukowaną.

Wrocław, 30 stycznia 2018 r.

(czytelny podpis)

Streszczenie

Visual Studio Code to na pierwszy rzut oka prosty edytor kodu, jednakże system rozszerzeń pozwala rozbudować go do pełnoprawnego środowiska programistycznego. Ponieważ istnieje wiele edytorów, a każdy z nich posiadał swój interfejs programistyczny, powstał protokół LSP, który umożliwia napisanie logiki wspomagającej pisanie programu raz i użycie jej w wielu edytorach.

Visual Studio Code at first seems like a simple code editor, though its extension system allows to expand it into full-featured IDE. As there are many editors, and each of them has its own application interface, a LSP protocol was created, to allow writing helper logic once and use it in many editors.

Spis treści

Rozdział 1.

Preliminaria

Aby zrozumieć jak działa dostarczony serwer, należy wpierw zrozumieć architekturę rozszerzenia w systemie edytora Visual Studio Code, a także na czym polega omawiany protokół. W tym rozdziale poruszone zostaną:

- Struktura wtyczki rozszerzającej działanie edytora.
- Opis protokołu LSP.
- Visual Studio a Visual Studio Code.
- Parser Luaparse.

1.1. Struktura rozszerzenia Visual Studio Code

Sercem każdego rozszerzenia jest plik `package.json`, który przechowuje informacje na temat autora pakietu, warunki jego uruchomienia, a także wszystkie jego zależności:

```
{
    "name": "lua-lang",
    "description": "Lua language support",
    "author": "Wiktor Adamski",
    "license": "MIT",
    "version": "0.0.1",
    "engines": {
        "vscode": "^1.16.0"
    },
    "categories": [
        "Languages"
    ],
    "activationEvents": [
```

```
            "onLanguage:lua"
        ],
        "main": "./out/src/extension",
        "contributes": {
            "languages": [
                {
                    "id": "lua",
                    "aliases": [
                        "Lua",
                        "lua"
                    ],
                    "extensions": [
                        ".lua",
                        ".p8",
                        ".rockspec"
                    ],
                    "configuration":
                       "./language-configuration.json"
                }
            ],
        },
        "dependencies": {
            "vscode": "^1.1.5",
            "vscode-languageclient": "^3.4.2"
        }
    }
```

O ile serwer LSP może być napisany w dowolnym języku, część bezpośrednio łącząca się z edytorem (aktualna wtyczka) musi być w języku JavaScript dla środowiska uruchomieniowego node.js.

Duża część kodu który jest wspólny dla wszystkich rozszerzeń jest możliwa do automatycznego stworzenia przez generator kodu Yeoman. Proste polecenie `yo code` przeprowadzi nas przez kreator wtyczek i utworzy dodatkowe pliki konfiguracyjne, które pozwolą korzystać z edytora VS Code jako środowiska developerskiego.

1.2. Protokół Language Server Protocol

Protokół LSP [1] jest specjalizacją protokołu JSON-RPC, który przesyła dane między stronami komunikacji za pomocą obiektów JSON. Klient (edytor kodu) wysyła zapytania do serwera (program wspomagający) odpowiadające różnym akcjom podejmowanym przez programistę, np. zapytanie się o miejsce deklaracji danej zmiennej. Pierwszą wiadomością w trakcie połączenia jest wymiana możliwości

zarówno klienta (np. czy edytor wspiera przemianowanie zmiennej), jak i serwera (np. wskazanie definicji danego symbolu lub automatyczne uzupełnianie pisanego tekstu). Twórcy protokołu udostępnili bibliotekę bibliotekę korzystanie z niego w języku TypeScript. Poniżej przedstawiam przykładowe zapytanie klienta:

```
{
    "jsonrpc": "2.0",
    "id": 1,
    "method": "textDocument/didOpen",
    "params": {
        ...
    }
}
```

1.3. Visual Studio (Code)

Wiele osób nie rozróżnia od siebie dwóch produktów Microsoftu. Visual Studio to zintegrowane środowisko programistyczne, nastawione głównie na pisanie programów w języku C#. Visual Studio Code jest natomiast otwartoźródłowym edytorem kodu opartym na silniku renderującym Electron od firmy Github, co sprawia, że jest on dosyć podobny do edytora Atom (również pod względem metodologii wtyczek).

1.4. Biblioteka Luaparse i drzewa rozbioru

Aby dostarczać jakiekolwiek sensowne informacje na temat kodu, potrzebne jest jego sparsowanie. Zajmuje się tym biblioteka Luaparse [2], która produkuje abstrakcyjne drzewa rozbioru programów napisanych w języku Lua. Drzewa reprezentowane za pomocą obiektów JavaScript są inspirowane na specyfikacji Mozilla Parser API. Przykładowo wyrażenie:

```
foo = "bar"
```

zostanie przełożone na drzewo:

```
{
    "type":"Chunk",
    "body":[{
        "type":"AssignmentStatement",
        "variables":[{
            "type":"Identifier",
            "name":"foo",
            "loc":{
                "start":{ "line":1, "column":0 },
```

```
                    "end":{ "line":1, "column":3 }
                }
            }],
            "init":[{
                "type":"StringLiteral",
                "value":"bar",
                "raw":"\"bar\"",
                "loc":{
                    "start":{ "line":1, "column":6 },
                    "end":{ "line":1, "column":11 }
                }
            }],
            "loc":{
                "start":{ "line":1, "column":0 },
                "end":{ "line":1, "column":11 }
            }
        }],
        "loc":{
            "start":{ "line":1, "column":0 },
            "end":{ "line":1, "column":11 }
        },
        "comments":[]
}
```

Rozdział 2.

Część kliencka rozszerzenia

W przypadku edytora Visual Studio Code, serwer LSP nie komunikuje się bezpośrednio z edytorem, potrzebny jest dodatkowy adapter w postaci osobnego małego rozszerzenia. W dodatku nie wszystkie możliwości edytora są wspierane przez protokół. Część kliencka rozszerzenia zatem ma 2 funkcjonalności:

1. Uruchomienie serwera i komunikacja z nim.
2. Udostępnienie funkcjonalności które nie są wspierane przez protokół LSP.

2.1. Komunikacja z serwerem

Ponieważ zarówno klient jak i serwer LSP są napisane w środowisku Node.js, możliwe było wykorzystanie bibliotek udostępnionych przez twórców edytora, przez co nawiązanie połączenia i jego obsługa ogranicza się do wskazania pliku serwera. Dodatkowo moduł serwera zostaje przy utworzeniu dodany do listy obiektów usuwanych przy zamknięciu rozszerzenia (np. zamknięte zostały wszystkie pliki Lua lub został wyłączony edytor).

2.2. Dodatkowe funkcjonalności

Klient poza przekazywaniem pracy do serwera, może implementować wiele osobnych funkcjonalności. Są to między innymi rejestracja danego języka programowania w słowniku edytora, co pozwala wielu wtyczkom dotyczących jednego języka na korzystanie ze swoich funkcji nawzajem. Inną ważną funkcjonalnością jest dodanie opisu kolorowania składni. W przypadku VS Code robi się to przez załączenie pliku gramatyki dla edytora TextMate (podejście identyczne jak w przypadku rozszerzeń do edytora Atom).

Rozdział 3.

Implementacja serwera LSP

Dysponując parserem kodu Lua, wystarczy odpowiednio przechodzić generowane przez niego drzewa, w celu odpowiedzi na poszczególne zapytania klienta. Punktem wejścia dla projektu będącego częścią niniejszej pracy jest artykuł [3] opisujący utworzenie prostego serwera LSP.

3.1. Zapytanie `Initialize`

W odpowiedzi na to zapytanie serwer zwraca informacje na temat jego możliwości. W przypadku tej pracy są to:

- Znalezienie definicji danej zmiennej lub funkcji.
- Automatyczne sugestie pisanego kodu.
- Wyświetlenie informacji na temat danego symbolu.

3.2. Komunikaty generujące drzewa

3.2.1. Komunikat `DidChangeTextDocument`

Komunikat `DidChangeTextDocument` informuje serwer, że użytkownik dokonał zmian w danym pliku. Uruchomiony zostaje wtedy parser, który czyta treść rzeczonego pliku (możliwością jest pracowanie na samej treści zmiany, jednakże wiązałoby się to z koniecznością napisania własnego parsera wspierającego inkrementalne zmiany w parsowanym tekście). Jeżeli parser napotkał jakiś błąd, jest on zwracany z powrotem do klienta i wyświetlany użytkownikowi pod postacią czerwonego podkreślenia problematycznego fragmentu kodu. Przy udanym parsowaniu otrzymane drzewo jest zapisywane w pamięci, aby można było je odwiedzić przy odpowiadaniu na inne zapytania.

3.2.2. Komunikat `DicChangeConfiguration`

Komunikat `DidChangeConfiguration` informuje nas, że użytkownik zmienił ustawienia edytora, co mogło wpłynąć na pliki w niekontrolowany przez nas sposób. W takim przypadku następuje ponowne parsowanie wszystkich otwartych dokumentów analogicznie do komunikatu `DidChangeTextDocument`.

3.3. Zapytania przechodzące po drzewie

3.3.1. Funkcja `TraverseTreeDown`

Każde z zapytań które mają w efekcie przejść się po drzewie rozbioru dostarcza nam informacje na temat pozycji w pliku na której się znajduje kursor. W takim razie wydzielona została funkcjonalność tłumaczenia pozycji na węzeł drzewa. Ponieważ każdy z węzłów drzewa rozbioru zawiera informację na temat zakresu tegoż węzła, wystarczy przejść się wgłąb drzewa tak długo jak szukana pozycja znajduje się wewnątrz zakresu przeszukiwanego wierzchołka. Funkcja zwraca odnaleziony wierzchołek.

3.3.2. Zapytanie `Hover`

Zapytanie `Hover` pyta się o możliwe informacje do wyświetlenia gdy użytkownik najedzie kursorem myszy na daną pozycję w pliku. Po znalezieniu najlepiej pasującego wierzchołka za pomocą funkcji `TraverseTreeDown`, zwracany jest komunikat którego dokładna treść jest określana na podstawie typu wierzchołka (np. dla funkcji będzie to jej nazwa i lista argumentów, a dla zmiennej jej typ).

3.3.3. Zapytanie `GotoDefinition`

Zapytanie `GotoDefinition` odpowiada akcji polegającej na szukaniu miejsca zdefiniowania danego symbolu w kodzie. Po znalezieniu szukanego symbolu następuje ponowne przejście po drzewie w celu odnalezienia najpóźniejszej definicji tegoż symbolu (w języku Lua symbole mogą być definiowane na nowo w trakcie działania programu, a także przesłaniane za pomocą słowa kluczowego `local`). Zwracana jest pozycja odnalezionej definicji.

3.3.4. Zapytanie `Completion`

Zapytanie `Completion` jest wysyłane przez klienta w celu odpytania serwera na temat możliwego dokończenia aktualnie pisanego tekstu. Również i w tym przypadku dostarczana jest pozycja kursora, jednakże serwer nie szuka aktualnie edytowanego

wierzchołka, tylko listę symboli które zostały zdefiniowane i są dostępne w danym kontekście. Lista odnalezionych symboli jest później rozszerzana o funkcje i zmienne zdefiniowane w bibliotece standardowej Lua (informacje na ich temat znajdują się w osobnym pliku JSON, który został utworzony na podstawie dokumentacji języka [4]).

Rozdział 4.

Podsumowanie

W ramach niniejszej pracy powstało rozszerzenie programu Visual Studio Code, które wspomaga programistę przy pisaniu kodu. Główna część programu, mianowicie serwer LSP, może zostać użyta przy implementacji analogicznego rozszerzenia dla innych edytorów, bez potrzeby wprowadzania zmian w kodzie. Powstałe rozszerzenie implementuje znaczną część funkcjonalności protokołu LSP, dodanie do niego pozostałych funkcji nie będzie zadaniem trudnym, jedynie czasochłonnym. Zadanie było rozwijające zarówno pod względem pracy z parserem kodu i interpretowaniem jego wyników, ale również pozwoliło prześledzić cały proces rozszerzania funkcjonalności istniejącego programu za pomocą udostępnionego interfejsu.

Bibliografia

[1] Microsoft Language Server Protocol documentation, 2018.

[2] Oskar Schöldström Luaparse, 2013 - 2017

[3] Microsoft Creating Language Servers for Visual Studio Code, 2018.

[4] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes Lua 5.3 Reference Manual, 2015 - 2017

Dodatek A

Instrukcja uruchomienia rozszerzenia

Niniejszy dodatek opisuje instrukcję uruchomienia rozszerzenia w programie Visual Studio Code. Wymagana jest dodatkowo instalacja środowiska Node.js, które musi być dostępne z wiersza poleceń. Kroki do wykonania:

1. Otworzyć katalog **lua** w edytorze VS Code.
2. Nacisnąć kombinację klawiszy `Ctrl+‘`, otwierając tym samym wbudowane okno terminala.
3. Wykonać polecenie `npm install`, które zainstaluje brakujące pakiety od których zależy rozszerzenie.
4. W sekcji **Debug** wybrać konfigurację **Launch Client** i nacisnąć przycisk zielonej strzałki (ewentualnie nacisnąć klawisz `F5`).
5. Zostanie otwarta druga instancja edytora, w której rozszerzenie jest aktywne.