

Implementacja protokołu LSP dla wybranego środowiska zintegrowanego

(Implementation of the LSP protocol
for selected 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 Language Server Protocol (LSP), który umożliwia napisanie logiki wspomagającej pisanie programu raz i użycie jej w wielu edytorach. Tematem tej pracy jest implementacja rozszerzenia do edytora kodu wykorzystującego ww. protokół.

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, Language Server Protocol (LSP) was created, to allow writing helper logic once and using it in many editors. The topic of this thesis is the implementation of code editor extension which uses said protocol.

Spis treści

1. Preliminaria	7
1.1. Visual Studio (Code)	7
1.2. Struktura rozszerzenia Visual Studio Code	7
1.3. Protokół Language Server Protocol	8
1.4. Biblioteka Luaparse i drzewa rozbioru	10
2. Część kliencka rozszerzenia	13
2.1. Komunikacja z serwerem	13
2.2. Dodatkowe funkcjonalności	14
3. Implementacja serwera LSP	15
3.1. Budowa serwera LSP	15
3.2. Opis protokołu LSP	16
3.3. Zapytanie <code>Initialize</code>	17
3.4. Komunikaty generujące drzewa	19
3.4.1. Komunikat <code>DidChangeTextDocument</code>	19
3.4.2. Komunikat <code>DidChangeConfiguration</code>	20
3.5. Zapytania przechodzące po drzewie	21
3.5.1. Wizytator <code>TraverseTreeDown</code>	21
3.5.2. Zapytanie <code>Hover</code>	21
3.5.3. Zapytanie <code>GotoDefinition</code>	21
3.5.4. Zapytanie <code>Completion</code>	22
4. Podsumowanie	25

Bibliografia	27
Dodatki	29
A Instrukcja uruchomienia rozszerzenia	29

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:

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

1.1. 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 edytorem kodu o otwartym kodzie opartym na silniku renderującym Electron od portalu Github, co sprawia, że jest on dosyć podobny do edytora Atom (również pod względem metodologii wtyczek). Dużą zaletą VS Code jest wbudowane wsparcie dla języków JavaScript i TypeScript, oraz środowiska Node.js, co przełożyło się na znaczny wzrost liczby użytkowników w ciągu 2 lat od powstania edytora (ponad 2,6 miliona aktywnych użytkowników [1])

1.2. 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"
  }
}

```

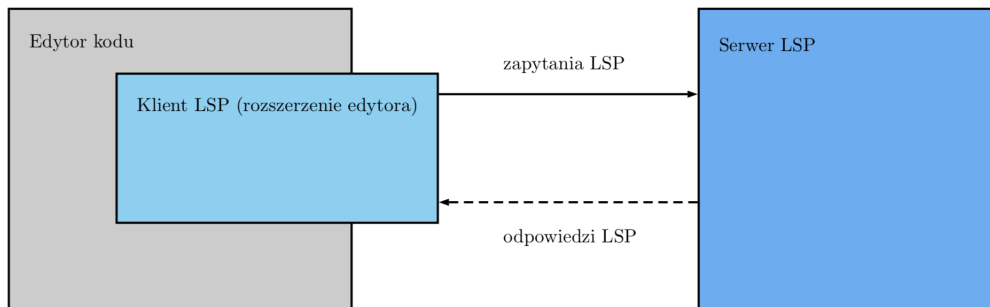
O ile serwer LSP może być napisany w dowolnym języku, część bezpośrednio łącząca się z edytorem (aktualna wtyczka) musi być w języku JavaScript dla środowiska uruchomieniowego Node.js.

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

1.3. Protokół Language Server Protocol

Protokół LSP [2] jest specjalizacją protokołu JSON-RPC, który przesyła dane między stronami komunikacji za pomocą obiektów JSON. Klient (rozszerzenie edytora 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 ułatwiającą korzystanie z niego w języku TypeScript. Poniżej przedstawiam przykładowy komunikat klienta informujący o otwarciu pliku w edytorze:

```
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "textDocument/didOpen",
  "params": {
    "uri": "/Users/wiktor/Desktop/test.lua",
    "languageId": "lua",
    "version": 1,
    "text": "w = 'hello world'\nprint(n)"
  }
}
```

W odpowiedzi serwer wyśle następujący komunikat z informacją o braku błędów w pliku:

```
{
  "jsonrpc": "2.0",
  "id": 2,
  "method": "textDocument/publishDiagnostics",
  "params": {
    "uri": "/Users/wiktor/Desktop/test.lua",
    "diagnostics": []
  }
}
```

Kontynuując, klient może wysłać zapytanie na temat definicji danego symbolu:

```
{
  "jsonrpc": "2.0",
  "id": 3,
  "method": "textDocument/definition",
  "params": {
    "uri": "/Users/wiktor/Desktop/test.lua",
    "position": {
```

```

        "line": 1,
        "character": 7
      }
    }
  }

```

Przy odpowiadaniu serwer musi stwierdzić co znajduje się na podanej w zapytaniu pozycji (w tym przykładzie zmienna `n`) i zwrócić miejsce w pliku gdzie została zadeklarowana:

```

{
  "jsonrpc": "2.0",
  "id": 3,
  "result": {
    "uri": "/Users/wiktor/Desktop/test.lua",
    "range": {
      "start": {
        "line": 0,
        "character": 0
      },
      "end": {
        "line": 0,
        "character": 0
      }
    }
  }
}

```

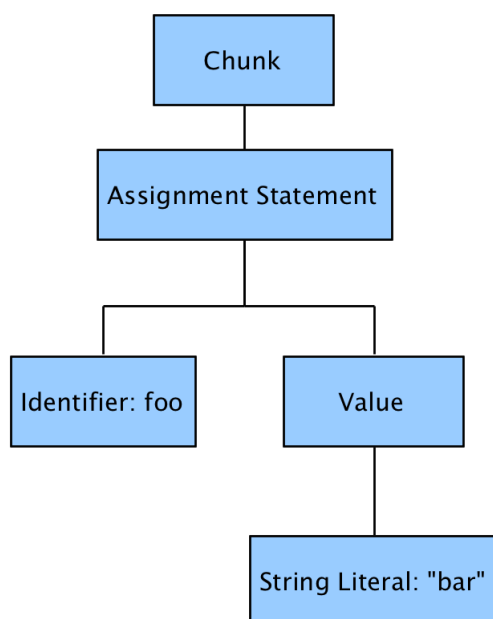
W powyższej odpowiedzi został zwrócony fragment długości 0, co daje znać edytorowi by zaznaczył całe słowo zawierające daną pozycję.

1.4. Biblioteka Luaparse i drzewa rozbioru

Aby dostarczać jakiegokolwiek sensowne informacje na temat kodu, potrzebne jest jego sparsowanie. Zajmuje się tym biblioteka Luaparse [3], 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 następujące drzewo:



Powstałe drzewo jest później wykorzystywane do wyliczania odpowiedzi na zapytania klienta LSP, np. posłuży do odnalezienia miejsca definicji zmiennej o którą pyta się użytkownik rozszerzenia.

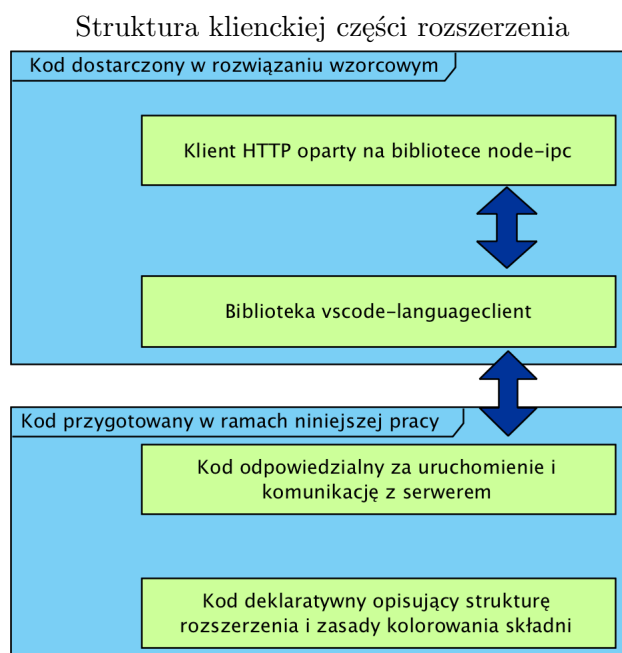
Rozdział 2.

Część kliencka rozszerzenia

W przypadku edytora Visual Studio Code, komunikacja między serwerem LSP a edytorem następuje poprzez dodatkowy adapter w postaci osobnego małego rozszerzenia. Dodatkowo, nie wszystkie możliwości edytora są wspierane przez protokół. Część kliencka rozszerzenia która powstała w ramach niniejszej pracy 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



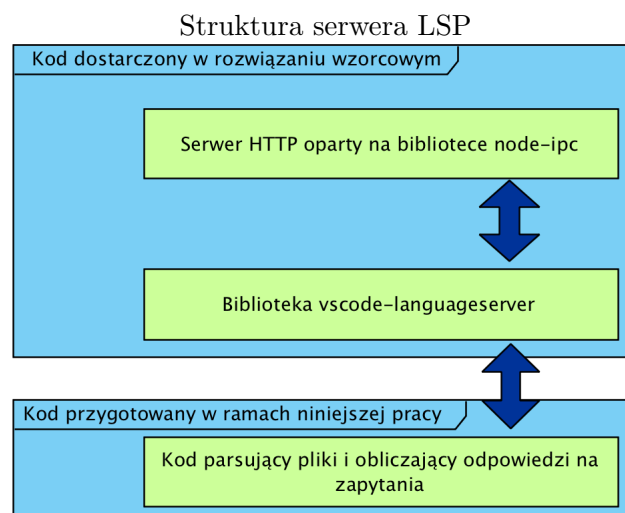
```
test_variable = 124
```

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 [4] opisujący utworzenie prostego serwera LSP.

3.1. Budowa serwera LSP



Jak widać na diagramie, kod odpowiedzialny za komunikację w ramach protokołu został dostarczony przez twórców odpowiednich bibliotek. W następnych podrozdziałach poruszana będzie implementacja tej części kodu która oblicza odpowiedzi na zapytania protokołu LSP.

3.2. Opis protokołu LSP

Protokół LSP, ze względu na szeroką gamę możliwości nowoczesnych edytorów kodu, ma niespełna 40 rodzajów zapytań i komunikatów. Poniżej znajduje się tabela z wypisanymi metodami tegoż interfejsu, a w następnych podrozdziałach nastąpi szczegółowy opis zaimplementowanych w ramach tej pracy rodzajów komunikatów. Metody oznaczone statusem “Nie będzie implementowana” są niepotrzebne do podstawowej pracy z edytorem, natomiast status “Planowana” oznacza, że metodę warto obsłużyć i będzie to kierunek dalszych prac.

Nazwa metody	Kierunek komunikacji Klient - Serwer	Status implementacji
Initialize	↻	Zaimplementowana
Initialized	→	Nie będzie implementowana
Shutdown	↻	Nie będzie implementowana
Exit	→	Nie będzie implementowana
ShowMessage	←	Nie będzie implementowana
ShowMessage	↻	Nie będzie implementowana
LogMessage	←	Nie będzie implementowana
Telemetry	←	Nie będzie implementowana
RegisterCapability	↻	Nie będzie implementowana
UnregisterCapability	↻	Nie będzie implementowana
DidChangeConfiguration	→	Zaimplementowana
DidChangeWatchedFiles	→	Planowana
Symbol	↻	Nie będzie implementowana
ExecuteCommand	↻	Nie będzie implementowana
ApplyEdit	↻	Nie będzie implementowana
DidOpen	→	Planowana
DidChange	→	Zaimplementowana
WillSave	→	Nie będzie implementowana
WillSaveWaitUntil	↻	Nie będzie implementowana
DidSave	→	Nie będzie implementowana
DidClose	→	Nie będzie implementowana
PublishDiagnostics	←	Obsłużona w ramach DidChange
Completion	↻	Zaimplementowana
Completion Resolve	↻	Nie będzie implementowana
Hover	↻	Zaimplementowana
SignatureHelp	↻	Nie będzie implementowana
Definition	↻	Zaimplementowana
References	↻	Nie będzie implementowana
DocumentHighlight	↻	Nie będzie implementowana
DocumentSymbol	↻	Nie będzie implementowana
CodeAction	↻	Nie będzie implementowana
CodeLens	↻	Planowana
CodeLens Resolve	↻	Planowana
DocumentLink	↻	Nie będzie implementowana
DocumentLink Resolve	↻	Nie będzie implementowana
Formatting	↻	Nie będzie implementowana
RangeFormatting	↻	Nie będzie implementowana
OnTypeFormatting	↻	Nie będzie implementowana
Rename	↻	Planowana

3.3. Zapytanie Initialize

Struktura argumentu zapytania

```
interface InitializeParams {
  processId: number | null
  rootPath?: string | null
  rootUri: DocumentUri | null
  initializationOptions?: any
  capabilities: ClientCapabilities
  trace?: 'off' | 'messages' | 'verbose'
}

interface ClientCapabilities {
  workspace?: WorkspaceClientCapabilities
  textDocument?: TextDocumentClientCapabilities
  experimental?: any
}

interface WorkspaceClientCapabilities {
  applyEdit?: boolean
  workspaceEdit?: { documentChanges?: boolean }
  didChangeConfiguration?: { dynamicRegistration?: boolean }
  didChangeWatchedFiles?: { dynamicRegistration?: boolean }
  symbol?: {
    dynamicRegistration?: boolean
    symbolKind?: { valueSet?: SymbolKind[] }
  }
  executeCommand?: { dynamicRegistration?: boolean }
}

interface TextDocumentClientCapabilities {
  synchronization?: {
    dynamicRegistration?: boolean
    willSave?: boolean
    willSaveWaitUntil?: boolean
    didSave?: boolean
  }
  completion?: {
    dynamicRegistration?: boolean
    completionItem?: {
      snippetSupport?: boolean
      commitCharactersSupport?: boolean
      documentationFormat?: MarkupKind[]
    }
    completionItemKind?: { valueSet?: CompletionItemKind[] }
    contextSupport?: boolean
  }
  hover?: {
    dynamicRegistration?: boolean
    contentFormat?: MarkupKind[]
  }
  signatureHelp?: {
    dynamicRegistration?: boolean
    signatureInformation?: { documentationFormat?: MarkupKind[] }
  }
  references?: { dynamicRegistration?: boolean }
  documentHighlight?: { dynamicRegistration?: boolean }
```

```

    documentSymbol?: {
        dynamicRegistration?: boolean
        symbolKind?: { valueSet?: SymbolKind[] }
    }
    formatting?: { dynamicRegistration?: boolean }
    rangeFormatting?: { dynamicRegistration?: boolean }
    onTypeFormatting?: { dynamicRegistration?: boolean }
    definition?: { dynamicRegistration?: boolean }
    codeAction?: { dynamicRegistration?: boolean }
    codeLens?: { dynamicRegistration?: boolean }
    documentLink?: { dynamicRegistration?: boolean }
    rename?: { dynamicRegistration?: boolean }
}

```

Struktura odpowiedzi

```

interface InitializeResult {
    capabilities: ServerCapabilities
}

interface ServerCapabilities {
    textDocumentSync?: TextDocumentSyncOptions | number
    hoverProvider?: boolean
    completionProvider?: CompletionOptions
    signatureHelpProvider?: SignatureHelpOptions
    definitionProvider?: boolean
    referencesProvider?: boolean
    documentHighlightProvider?: boolean
    documentSymbolProvider?: boolean
    workspaceSymbolProvider?: boolean
    codeActionProvider?: boolean
    codeLensProvider?: CodeLensOptions
    documentFormattingProvider?: boolean
    documentRangeFormattingProvider?: boolean
    documentOnTypeFormattingProvider?: DocumentOnTypeFormattingOptions
    renameProvider?: boolean
    documentLinkProvider?: DocumentLinkOptions
    executeCommandProvider?: ExecuteCommandOptions
    experimental?: any
}

namespace TextDocumentSyncKind {
    const None = 0
    const Full = 1
    const Incremental = 2
}

interface CompletionOptions {
    resolveProvider?: boolean
    triggerCharacters?: string[]
}

interface SignatureHelpOptions {
    triggerCharacters?: string[]
}

interface CodeLensOptions {
    resolveProvider?: boolean
}

```

```
interface DocumentOnTypeFormattingOptions {
    firstTriggerCharacter: string
    moreTriggerCharacter?: string[]
}

interface DocumentLinkOptions {
    resolveProvider?: boolean
}

interface ExecuteCommandOptions {
    commands: string[]
}

interface SaveOptions {
    includeText?: boolean
}

interface TextDocumentSyncOptions {
    openClose?: boolean
    change?: number
    willSave?: boolean
    willSaveWaitUntil?: boolean
    save?: SaveOptions
}
```

Zapytanie `Initialize` posiada masywny interfejs zarówno argumentu zapytania klienta, jak i odpowiedzi serwera. Jest tak ze względu na możliwość implementacji dowolnego fragmentu interfejsu LSP. Przy odpowiadaniu na to zapytanie serwer zwraca przygotowany wcześniej obiekt JSON określający jakie funkcjonalności implementuje. W przypadku tej pracy są to:

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

3.4. Komunikaty generujące drzewa

3.4.1. Komunikat `DidChangeTextDocument`

Struktura argumentu komunikatu

```
interface DidChangeTextDocumentParams {
    textDocument: VersionedTextDocumentIdentifier
    contentChanges: TextDocumentContentChangeEvent[]
}

interface VersionedTextDocumentIdentifier {
    uri: string
    version: number
}
```

```

interface TextDocumentContentChangeEvent {
  range?: Range
  rangeLength?: number
  text: string
}

interface Range {
  start: Position
  end: Position
}

interface Position {
  line: number
  character: number
}

```

Komunikat `DidChangeTextDocument` zostaje przesłany do serwera gdy użytkownik zmodyfikował treść pliku (niekoniecznie plik został po tych zmianach zapisany). Należy tutaj nadmienić, że protokół LSP wspiera dwie metody synchronizacji treści pliku między klientem a serwerem, pełną w której każdy komunikat o zmianie pliku zawiera całkowitą treść tego pliku, lub inkrementalną, w którym przesyłane są jedynie pozycje i treść zmienionych fragmentów. Tryb synchronizacji jest określany w odpowiedzi na zapytanie `Initialize`. Na potrzeby tej pracy zaimplementowano tryb pełny. Po otrzymaniu komunikatu serwer uruchamia parser języka Lua na treści pliku przesłanej w komunikacie. Jeżeli parser napotkał jakiś błąd, jest on zwracany z powrotem do klienta w komunikacie `PublishDiagnostics` i wyświetlany użytkownikowi pod postacią czerwonego podkreślenia problematycznego fragmentu kodu. Przy udanym parsowaniu otrzymane drzewo jest zapisywane słownikowi działającemu w roli cache'a, aby można było je odwiedzić przy odpowiadaniu na inne zapytania. Jest to opcjonalna optymalizacja, która pozwala uniknąć ponownego parsowania pliku przy niezmienionej treści. Drzewo dla danego pliku jest ważne tak długo jak nie zostanie dostarczony nowy komunikat świadczący o zmianie treści pliku. Treść pliku również jest zapisywana po stronie serwera za pomocą menadżera otwartych plików dostarczanego przez bibliotekę `vscode-languageserver`.

3.4.2. Komunikat `DidChangeConfiguration`

Struktura argumentu komunikatu

```

interface DidChangeConfigurationParams {
  settings: any
}

```

Komunikat `DidChangeConfiguration` informuje nas, że użytkownik zmienił ustawienia edytora, co mogło w różny sposób wpłynąć na proces parsowania otwartych plików. W takim przypadku następuje ponowne parsowanie wszystkich dokumentów znajdujących się w cache serwera, każdy z nich traktując analogicznie do komunikatu

DidChangeTextDocument.

3.5. Zapytania przechodzące po drzewie

3.5.1. Wizytator 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, wizytator schodzi po drzewie tak długo jak szukana pozycja znajduje się w zakresie przeszukiwanego wierzchołka. Funkcja zwraca wierzchołek na którym zatrzymało się przeszukiwanie.

3.5.2. Zapytanie Hover

Struktura argumentu zapytania

```
interface TextDocumentPositionParams {  
    textDocument: string  
    position: Position  
}
```

Struktura odpowiedzi

```
interface Hover {  
    contents: MarkedString | MarkedString[]  
    range?: Range  
}
```

Zapytanie `Hover` zostaje wysłane przez edytor, gdy użytkownik zatrzyma na chwilę kursor myszy nad fragmentem tekstu. Edytor ma wtedy możliwość pokazania dodatkowego okienka zawierającego informacje na temat danego miejsca w kodzie. Serwer w odpowiedzi ma zwrócić treść tegoż okienka, która może być oznakowana za pomocą składni Markdown. Po znalezieniu najlepiej pasującego wierzchołka za pomocą funkcji `TraverseTreeDown`, zwracany jest tekst którego dokładna treść jest określana na podstawie typu wierzchołka (np. dla funkcji będzie to jej nazwa i lista argumentów, a dla zmiennej jej typ).

3.5.3. Zapytanie GotoDefinition

Struktura argumentu zapytania

```
interface TextDocumentPositionParams {  
    textDocument: string  
    position: Position  
}
```

Struktura odpowiedzi

```
type GotoDefinitionReturn = Location | Location[]
```

Zapytanie `GotoDefinition` odpowiada wywołaniu przez użytkownika akcji **Go to Definition**, polegającej na szukaniu miejsca zdefiniowania w kodzie symbolu znajdującego się pod kursorem myszy. 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`). W odpowiedzi zwracana jest pozycja w tekście odnalezionej definicji. Następnie edytor ustawia kursor na otrzymaną pozycję, odkrywając w ten sposób szukaną definicję dla użytkownika.

3.5.4. Zapytanie Completion

Struktura argumentu zapytania

```
interface CompletionParams {
  textDocument: string
  position: Position
  context?: CompletionContext
}

namespace CompletionTriggerKind {
  const Invoked: 1 = 1
  const TriggerCharacter: 2 = 2
}
type CompletionTriggerKind = 1 | 2

interface CompletionContext {
  triggerKind: CompletionTriggerKind
  triggerCharacter?: string
}
```

Struktura odpowiedzi

```
interface CompletionList {
  isIncomplete: boolean
  items: CompletionItem[]
}

namespace InsertTextFormat {
  const PlainText = 1
  const Snippet = 2
}
type InsertTextFormat = 1 | 2

interface CompletionItem {
  label: string
  kind?: number
  detail?: string
  documentation?: string | MarkupContent
  sortText?: string
  filterText?: string
  insertText?: string
  insertTextFormat?: InsertTextFormat
  textEdit?: TextEdit
  additionalTextEdits?: TextEdit[]
  commitCharacters?: string[]
}
```

```

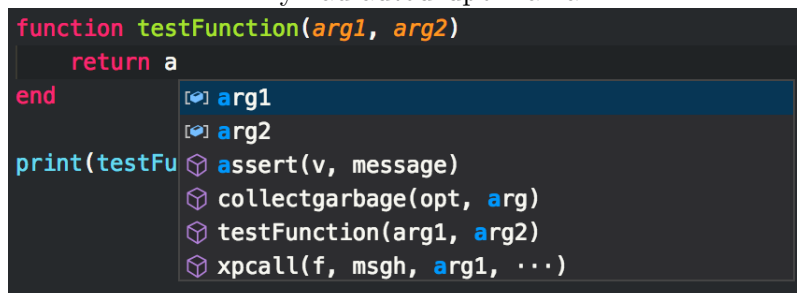
    command?: Command
    data?: any
}

namespace CompletionItemKind {
    const Text = 1
    const Method = 2
    const Function = 3
    const Constructor = 4
    const Field = 5
    const Variable = 6
    const Class = 7
    const Interface = 8
    const Module = 9
    const Property = 10
    const Unit = 11
    const Value = 12
    const Enum = 13
    const Keyword = 14
    const Snippet = 15
    const Color = 16
    const File = 17
    const Reference = 18
    const Folder = 19
    const EnumMember = 20
    const Constant = 21
    const Struct = 22
    const Event = 23
    const Operator = 24
    const TypeParameter = 25
}

```

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 [5]). Obliczona lista zostaje odesłana do klienta, a edytor przed pokazaniem jej użytkownikowi sortuje wpisy pod względem prawdopodobieństwa poprawnego dokończenia wpisywanej frazy.

Przykład autouzupełniania



```

function testFunction(arg1, arg2)
    return a
end
print(testFu

```

The screenshot shows a code editor with a dark theme. The code being edited is a Lua function `testFunction` that takes two arguments, `arg1` and `arg2`, and returns a value `a`. The function is called in a `print` statement. The text `print(testFu` is currently selected, and a dropdown menu of suggestions is visible. The suggestions include `arg1`, `arg2`, `assert(v, message)`, `collectgarbage(opt, arg)`, `testFunction(arg1, arg2)`, and `xpcall(f, msg, arg1, ...)`. The suggestions are displayed with small icons to the left of the text.

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] Sean McBreen Visual Studio Code at **Connect(); 2017**, 2017
- [2] Microsoft Language Server Protocol documentation, 2018
- [3] Oskar Schöldström Luaparse, 2013 - 2017
- [4] Microsoft Creating Language Servers for Visual Studio Code, 2018
- [5] 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.