VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

PODPORA PRO VÝUKOVÝ JAZYK MEZIKÓDU

SUPPORT FOR EDUCATIONAL INTERMEDIATE LANGUAGE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ONDŘEJ KREJČÍ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. ZBYNĚK KŘIVKA, Ph.D.**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Krejčí Ondřej**
Program: Informační technologie
Název: **Podpora pro výukový jazyk mezikódu**
Support for Educational Intermediate Language
Kategorie: Překladače

Zadání:

1. Seznamte se s překladači, vývojovými nástroji (IDE Eclipse, NetBeans, Visual Code apod.). Seznamte se s možnostmi integrace nového jazyka do vybraného IDE.
2. Nastudujte syntaxi a sémantiku intermediálních jazyků pro překladačnické projekty v předmětech IFJ a VYPa (tj. IFJcode a VYPcode).
3. Dle konzultace s vedoucím navrhněte podporu pro editaci zdrojového kódu IFJcode a VYPcode (včetně jejich každoročních drobných úprav), možnost spouštění a ladění zdrojových programů přímo v IDE.
4. Návrh implementujte a řádně otestujte a zhodnoťte.

Literatura:

- Lorenzo Bettini. Implementing Domain-Specific Languages with Xtext and Xtend (2nd Edition). Packt Publishing, 2016
- Microsoft. Language Extensions Overview | Visual Studio Code Extension API. 2020. Dostupné z: https://code.visualstudio.com/api/language-extensions/overview [cit. 2020-10-15]
- Zadání projektů do předmětu IFJ a VYPa

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Křivka Zbyněk, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 22. října 2020

Abstrakt

Tato bakalářská práce se zabývá vytvářením softwarové podpory pro výukové jazyky mezikódu IPPcode, IFJcode a VYPcode používané na FIT VUT v předmětech týkajících se překladačů. Cílem práce je vytvoření rozšíření pro existující editor zdrojových textů, které pro zmíněné mezikódy přidá zvýrazňování syntaxe, interpret a ladění. Práce obsahuje zhodnocení možných editorů, následně popisuje možnosti rozšíření ve zvoleném editoru Visual Studio Code a implementaci rozšíření.

Abstract

This bachelor's thesis deals with the creation of sotware support for educational intermediate codes IPPcode, IFJcode and VYPcode which are used at FIT BUT in courses covering compilers. The aim of this thesis is the creation of an extension for an existing source code editor which adds syntax highlighting, interpreter and debugger for the mentioned intermediate codes. This thesis contains an evaluation of possible editors. Subsequently, it describes options for extending the selected editor Visual Studio Code and the implementation of the extensions.

Klíčová slova

mezikód, Visual Studio Code, rozšíření, zvýrazňování syntaxe, protokol Debug Adapter, ladicí nástroje

Keywords

intermediate code, Visual Studio Code, extensions, syntax highlighting, Debug Adapter Protocol, debuggers

Citace

KREJČÍ, Ondřej. *Podpora pro výukový jazyk mezikódu.* Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zbyněk Křivka, Ph.D.

Podpora pro výukový jazyk mezikódu

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zbyňka Křivky, Ph.D. Další informace mi poskytl pan Ing. Radim Kocman. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Ondřej Krejčí
8. května 2021

Poděkování

Děkuji vedoucímu mé bakalářské práce Ing. Zbyňku Křivkovi, Ph.D. za odborné vedení a ochotu při konzultacích k této práci. Dále děkuji Ing. Radimovi Kocmanovi za poskytnutí zdrojových kódů interpretu jazyka VYPcode a konzultace k implementaci ladicího nástroje pro jazyk VYPcode. Také děkuji Janě Krejčí a Zdeňkovi a Anně Krejčím za jejich podporu po celou dobu studia.

Obsah

Kapitola 1

Úvod

Na fakultě informačních technologií Vysokého učení technického v Brně mají studenti v rámci předmětů zabývajících se tvorbou překladačů povinnost vypracovat projekt, ve kterém implementují překladač programovacího jazyka. Jeho výstupem je tzv. mezikód, což je výsledek překladu jiného programovacího jazyka, který ale sám o sobě není cílovým kódem spustitelným na konkrétní cílové platformě. Tyto mezikódy jsou specificky definované pro výukové účely v těchto předmětech, nejedná se o žádné v praxi používané mezikódy.

Vzhledem k omezenému použití těchto jazyků v rámci několika předmětů na jedné fakultě pro ně neexistuje žádná podpora v textových editorech ani integrovaných vývojových prostředích. Jediné programy, které s těmito mezikódy pracují, jsou tedy programy vytvářené studenty (překladače a interprety) a referenční interprety poskytnuté autory zadání projektů.

Rozšiřitelnost je v současné době běžnou vlastností editorů zdrojových textů. Může se jednat pouze o možnost přidání zvýrazňování syntaxe pro jazyk, který editor ve výchozí verzi nepodporuje. Možnosti rozšíření ale mohou být podstatně širší a mohou umožňovat doplnění zcela nové funkčnosti, případně dodání širší podpory pro libovolný jazyk. Pro tyto účely je nutné vytvořit program implementující novou funkcionalitu, který je definovaným způsobem začleněn do existujícího editoru. Klíčové je především, jaké možnosti rozšíření editory nabízí, a jak zdokumentovaný je jejich vývoj.

Cílem této práce je zvolit editor zdrojových textů a vytvořit pro něj rozšíření doplňující podporu pro zmíněné mezikódy. Rozšíření by mělo zajistit rozpoznání souborů obsahujících zvolený mezikód a zvýrazňovat jeho syntaxi. Dále by měla být přidána možnost mezikód v rámci editoru interpretovat a ladit. Pro implementaci rozšíření byl zvolen editor Visual Studio Code.

Následující kapitola obsahuje specifikaci mezikódů, kterými se tato práce zabývá. Další kapitola obsahuje přehled možností vytváření rozšíření pro některé současné editory zdrojových textů, stručný přehled významu ladění programů a podrobnější informace o možnostech rozšíření pro zvolený editor Visual Studio Code. Po těchto úvodních kapitolách je popsán návrh rozšíření a následně jejich implementace a testování. Závěr obsahuje shrnutí práce.

Kapitola 2

Jazyky mezikódu IFJcode a VYPcode

Jazyky IFJcode a VYPcode jsou intermediální jazyky využívané pro studijní a výukové účely. Vznikají jako výstup překladu různých vyšších programovacích jazyků v rámci bakalářského předmětu IFJ[1] (Formální jazyky a překladače) a magisterského předmětu VYPa[2] (Výstavba překladačů) na Fakultě informačních technologií Vysokého učení technického v Brně (FIT VUT). V rámci obou těchto předmětů studenti vypracovávají projekt, jehož cílem je vytvoření překladače. Vstupem překladačů je vyšší programovací jazyk, který se s každým akademickým rokem mění. Výstupem je odpovídající mezikód, jehož formát zůstává z větší části neměnný. Určité změny v něm ale existují, tyto rozdíly mezi jednotlivými verzemi jsou většinou dány vlastnostmi překládaného jazyka.

2.1 Jazyk IFJcode

Jazyk IFJcode je mezikód, který je obecně složen z jednořádkových tříadresných a zásobníkových instrukcí. Pro jeho definici je použito konkrétní zadání pro rok 2020 [12]. Je to dynamický, silně typovaný jazyk.

Tříadresné instrukce můžeme podle [1] definovat jako mezikód tvořený instrukcemi podobnými jazyku symbolických adres (angl. *assembler*, *assembly language*). Instrukce tříadresného kódu má maximálně 3 operandy, a pokud instrukce přiřazuje hodnotu, obsahuje maximálně jeden operátor na pravé straně přiřazení, jak můžeme vidět v následujícím zápisu 2.1, kde znak `=` značí přiřazení, a, b, c jsou operandy a **op** značí operátor.

$$a = b \,\mathbf{op}\, c \tag{2.1}$$

Každá instrukce tříadresného kódu provádí jednu operaci, nejčastěji matematickou nebo logickou, např. aplikování operátoru sčítání (uložení výsledku se chápe jako součást provedení této operace). Kód je tvořený posloupností instrukcí, nemá žádnou další strukturu kromě definic návěští. Složitější řízení je umožněno instrukcemi skoku, které přesunou vykonávání programu do jiné části kódu.

Tříadresné instrukce jazyka IFJcode odpovídají definici tříadresného kódu, možnosti jazyka jsou navíc rozšířeny o zásobníkové instrukce. Zásobníkové instrukce vykonávají identické operace jako tříadresné instrukce. Navíc existují instrukce pro práci se zásobníkem

[1] https://www.fit.vut.cz/study/course/13981/.cs
[2] https://www.fit.vut.cz/study/course/14192/.cs

sloužící k vložení a vyjmutí hodnot, případně vyprázdnění zásobníku. Zásobníkové instrukce své operandy získávají ze zásobníku a nejsou tedy obsažené v zápisu instrukce (až na výjimky, jako jsou např. instrukce skoku, které mají jako operand název návěští).

2.1.1 Specifikace a implementace

Zdrojový kód, jehož překladem vzniká jazyk IFJcode se každoročně mění. Toto má za důsledek, že ze všech v této práci zmíněných jazyků mezikódu se specifikace jazyka IFJcode mění nejvíce. V každoročních zadáních je snaha, aby chování vstupních a výstupních instrukcí jazyka IFJcode odpovídalo zdrojovému jazyku. Například v roce 2020 jazyk IFJcode v tomto smyslu odpovídal jazyku Python. S výjimkou těchto instrukcí je však implementace interpretu jazyka IFJcode poměrně neměnná. Interpret je implementován v jazyce C++ a pro studenty je dostupný pouze v podobě spustitelného binárního souboru, který může být použit k otestování přeloženého kódu.

Další vlastnost jazyka je dána implementačním jazykem překladače, kterým je jazyk C. Rozsah celočíselného typu proto odpovídá 64bitovému celočíselnému typu `long long int` jazyka C. Zápis literálu (přímý zápis hodnoty ve zdrojovém kódu v určitém definovaném formátu) desetinného čísla odpovídá tisku desetinného čísla standardní funkcí `printf` s formátovacím řetězcem `"%a"` pro tisk v hexadecimálním formátu.

2.1.2 Instrukce

Každá instrukce je složena z klíčového slova operačního kódu (názvu) instrukce následovaného nula až třemi operandy, jejichž počet je pevně daný názvem instrukce. U názvu instrukce nezáleží na velikosti písmen (angl. *case-insensitive*), u operandů na velikosti písmen záleží. Operand může být konstanta nebo proměnná, u instrukcí skoku je jeden z operandů název cílového návěští, u instrukce pro načtení vstupu je jako operand zadán datový typ. Pro oddělení názvu instrukce a operandů slouží libovolný neprázdný řetězec složený z mezer a tabulátorů.

Jednotlivé instrukce jsou odděleny znakem konce řádku a mezi instrukcemi může být libovolný počet prázdných řádků. Jazyk obsahuje pouze jednořádkové komentáře, které začínají znakem `#`. Řádky obsahující pouze tzv. bílé znaky (mezery a tabulátory) nebo komentář se také berou jako prázdné.

Jedna konkrétní instrukce je ukázána na příkladu 2.1, operační kód `ADD` značí instrukci sčítání, která sečte dva operandy `op1` a `op2` a uloží výsledek do proměnné `vysledek`.

```
ADD GF@vysledek GF@op1 GF@op2
```

Výpis 2.1: Ukázka instrukce sčítání jazyka IFJcode.

2.1.3 Proměnné a paměťový model

Jazyk umožňuje dva různé způsoby uložení hodnot. První je ukládání do proměnných, které jsou pojmenované. Na základě jména proměnné je možné přistoupit ke všem dostupným definovaným proměnným. Druhou možností je uložení hodnot na datový zásobník.

Pojmenované proměnné se definují instrukcí `DEFVAR`. Operandem instrukce je jméno proměnné, které má dvě části oddělené znakem `@`, jak lze vidět z ukázky instrukce 2.1. První část je název rámce, druhá část je samotný název proměnné. Název rámce určuje, do kterého rámce se nová proměnná uloží.

Rámce jsou datové struktury typu mapa (nebo asociativní pole) obsahující proměnné identifikované jejich názvem. Existují tři typy rámců, globální (`GF`), lokální (`LF`) a dočasný (`TF`). Dále existuje zásobník rámců, který je na začátku interpretace prázdný.

Globální rámec je pouze jeden a existuje po celou dobu interpretace kódu, na začátku je vytvořen automaticky.

Dočasný rámec na začátku interpretace neexistuje, vytváří se instrukcí `CREATEFRAME` a může být vytvořen opakovaně (čímž je přepsán případný existující dočasný rámec). Instrukcí `PUSHFRAME` se dočasný rámec přesune na zásobník rámců, dočasný rámec je poté opět nedefinován. Opačnou instrukcí je `POPFRAME`, která vyjme rámec z vrcholu zásobníku rámců a vytvoří z něj dočasný rámec.

Lokální rámec je rámec na vrcholu zásobníku rámců, rámce uložené v zásobníku rámců pod ním jsou nedostupné.

Alternativou k použití proměnných je datový zásobník, který je vždy dostupný a existuje od začátku interpretace. Je možné na něj ukládat hodnoty, odstraňovat je a ukládat do proměnných, nebo uložené hodnoty používat přímo zásobníkovými instrukcemi.

2.1.4 Datové typy a konstanty

Nově definovaná proměnná neobsahuje žádnou hodnotu a nemá definovaný typ. Takovou proměnnou není možné použít jinak než k uložení hodnoty. Pokus o získání hodnoty vyvolá chybu. Typ se dynamicky mění podle datového typu hodnoty uložené v proměnné. Existuje pět datových typů, speciální typ `nil` značící chybějící hodnotu (odlišné od žádné hodnoty u nově vytvořené proměnné) a čtyři standardní datové typy, znaménkové celé číslo – `int`, znaménkové desetinné číslo – `float`, logická hodnota pravda/nepravda (standardní `true` nebo `false`) – `bool` a řetězec – `string`.

Konstantní hodnoty se zapisují jako literály odpovídajícího typu. Formát libovolného literálu je ve tvaru `typ@hodnota`. V tomto zápise `typ` odpovídá jednomu z podporovaných typů popsaných v předchozím odstavci a `hodnota` nese hodnotu v daném formátu.

Pro typy `nil` a `bool` jsou možné hodnoty přímo řetězcovým zápisem povolených hodnot, tedy `nil` a `true` nebo `false`. Literály pro číselné typy odpovídají jazyku C, jak bylo popsáno v podkapitole o implementaci 2.1.1. Formát řetězcového typu `string` je sekvence `ASCII` znaků s výjimkou bílých znaků a znaků `#` a `\`. Navíc může obsahovat escape sekvence začínající zpětným lomítkem ve tvaru `\abc`, kde `abc` jsou číslice tvořící dohromady číslo v rozmezí 000 až 255 (počet číslic je pevný, např. escape sekvence `\10` není validní).

2.1.5 Jazyk IPPcode

Jazyk IPPcode vychází z jazyka IFJcode a je z velké části identický, protože předmět IPP navazuje na předmět IFJ a je snaha, aby vytvořené programy v IFJcode byly dále použitelné. Konkrétním příkladem zadání projektu obsahujícího jeho specifikaci je například [13]. Referenční interpret jak jazyka IFJcode, tak jazyka IPPcode je napsaný v jazyce C++. Cílem projektu v předmětu IPP je ale vytvořit interpret jazyka IPPcode v jazyce Python, podle čehož je psaná specifikace. Rozdíly jsou ve vstupních a výstupních instrukcích, kde jazyk IFJcode je popsaný tak, aby odpovídal jazyku C, zatímco jazyk IPPcode odpovídá jazyku Python. Rozdíl je například při načítání vstupu u instrukce `READ`, kde se používá funkce `input` a na načtení čísla typu `float` se používá funkce `float.fromhex`, jejíž vstup ovšem podporuje hodnoty vytištěné s použitím formátovacího řetězce `"%a"`.

Zadání projektu [13] obsahuje v základu specifikaci menšího množství instrukcí než specifikace jazyka IFJcode, jelikož implementace instrukcí pracujících se zásobníkem a typem

`float` je součástí rozšíření pro projekt. Spolu s těmito rozšířeními obsahuje jazyk identické instrukce jako jazyk IFJcode, další rozšíření zabývající se sbíráním statistik nejsou pro tuto práci relevantní.

2.2 Jazyk VYPcode

Jazyk VYPcode svou syntaxí připomíná jazyk symbolických instrukcí. Kód v jazyce VYPcode je rozdělen do několika sekcí a je složen z jednořádkových instrukcí a definic. Oproti běžným jazykům symbolických instrukcí má rozšířené možnosti pro práci se složitými datovými strukturami. Pro jeho definici je použito konkrétní zadání pro rok 2020 [11].

Jazyk pracuje s registry, jejichž počet je zvolen při zahájení interpretace. Počet obecných registrů je definovaný jako libovolné kladné číslo, existuje tedy vždy minimálně jeden obecný registr. Obecné registry jsou označené jako `$a`, kde `a` je index registru počítaný od nuly. Dále k ukládání hodnot slouží halda (angl. *heap*) a zásobník.

2.2.1 Instrukce

Instrukce jazyka je složena z klíčového slova operačního kódu instrukce následovaného operandy, jejichž počet je daný názvem instrukce. U názvu instrukce nezáleží na velikosti písmen. Název instrukce je od operandů oddělen libovolným nenulovým počtem bílých znaků, operandy jsou od sebe oddělené jedním znakem čárky nebo nenulovým počtem bílých znaků, před následujícím operandem může být vždy libovolný počet bílých znaků. Jazyk podporuje pouze jednořádkové komentáře, které začínají znakem `#` nebo `;`.

Hodnotu pro operandy je možné získat třemi různými způsoby adresování. První možností je adresování registrem (angl. *register addressing*), kdy se používají identifikátory registrů ve tvaru `$x` k určení registru, který se použije k uložení nebo načtení hodnoty. Další možností je tzv. přímé adresování (angl. *immediate addressing*), kdy se přímo používá hodnota, která je uložena specificky pro použití v této instrukci. Možné hodnoty jsou literály, konstanty nebo konstantní výrazy. Poslední možností je nepřímé zásobníkové adresování (angl. *indirect stack addressing*), kdy se hodnota operandu použije jako hodnota posunu na zásobníku pro přístup k hodnotě na něm. Operand je ve formátu `[a]`, kde `a` je registr, konstantní hodnota, pojmenovaná konstanta nebo výraz ve tvaru `$x [+-] konstanta`.

2.2.2 Paměťový model a speciální registry

Základní datová jednotka je slovo (angl. *word*) o rozsahu 64 bitů. Registry, položky na zásobníku a položky na haldě mají všechny rozsah jedno slovo.

Zásobník je souvislý blok paměti, která se používá po jednotlivých slovech a roste směrem nahoru. Rozsah je možné nastavit při zahájení interpretace, výchozí hodnota je 65535 slov.

Halda je další úsek paměti oddělený od zásobníku. Je složena z jednotlivých struktur nazvaných shluk (angl. *chunk*). Halda funguje jako asociativní paměť, kterou lze indexovat unikátními identifikátory existujících shluků. Každý shluk má svůj unikátní identifikátor (počítaný od čísla jedna) a počet položek uložených ve shluku, což je hodnota, která odpovídá velikosti shluku. Poté následuje obsah shluku, což je posloupnost uložených položek, kde každá položka má velikost jednoho slova. Velikost shluků je možné měnit. Identifikátor 0 je vyhrazen pro nedefinovaný shluk.

Speciální registr `$SP` ukazuje na vrchol zásobníku. Jeho hodnotu je možné přímo upravovat instrukcemi programu.

Druhý speciální registr `$PC` (angl. *program counter*, programový čítač) ukazuje na další instrukci, která má být vykonána. Po načtení instrukce z `$PC` se registr automaticky aktualizuje na novou hodnotu.

2.2.3 Datové typy, literály a konstanty

Jazyk VYPcode má tři datové typy, celočíselný `integer`, desetinný `float` a řetězec tisknutelných `UTF-8` znaků `string`.

Pro zápis hodnoty typu `integer` se používá buďto běžný decimální formát nebo hexadecimální formát s prefixem `0x`.

Datový typ `float` umožňuje tři zápisy. Běžný decimální formát, u kterého je povinná desetinná tečka. Dále vědecký formát (obsahující koncovku začínající `E` značící exponent konstanty 10, kterou se číslo násobí). Poslední formát je hexadecimální, který odpovídá výpisu ze standardní funkce `printf` při použití formátu `"%a"`.

Řetězcový literál je obklopen znaky dvojitých uvozovek `"` a je složen z tisknutelných znaků v kódování `UTF-8`. Tisknutelné znaky mají hodnotu větší než 31, výjimkou je znak s hodnotou 34, který odpovídá dvojitým uvozovkám obklopujícím literál. Kromě přímého zápisu je možné použít předdefinované escape sekvence `\n`, `\t`, `\\`, `\"` nebo obecnou escape sekvenci `\xhhhhhh`, kde `h` značí hexadecimální číslici. Tyto escape sekvence fungují identicky jako v jazyce C. Interně je každý řetězcový literál uložen ve shluku jako posloupnost položek.

Navíc k literálům reprezentujícím datové typy existuje literál pro název, který je používaný pro návěští a pojmenované konstanty. Literál názvu je složen z písmen, znaků `.`, `:`, `@`, `_` a s výjimkou prvního znaku literálu i číslic.

Každý literál je konstanta, kromě toho je možné definovat i pojmenované konstanty. Definice má tvar `CONSTANT název hodnota`, čímž je definován název konstanty, který se dále může používat místo hodnoty konstanty. Tento název musí být unikátní v množině názvů konstant a návěští. Hodnota musí být konstantní výraz, který je složen z literálů datových typů, již definovaných konstant, operací, které nad nimi lze podle jejich datového typu provádět, a závorek.

Podobně jako konstanty je možné definovat aliasy pro registry ve tvaru `ALIAS název $x`. K registru je poté možné přistupovat pomocí definovaného názvu aliasu, před který se doplní znak `$`.

2.2.4 Sekce

Kód zapsaný v jazyce VYPcode je členěný do tří sekcí. Teoreticky není povinná žádná sekce, ale očekává se, že instrukční sekce bude neprázdná, aby došlo k vykonání nějaké akce.

První je hlavičková sekce (angl. *header*), která obsahuje tři nepovinné komentářové řádky, první je tzv. *shebang*, což je řádek určující interpret pro zapsaný kód.

Poté následuje sekce konstant (angl. *constant section*), která obsahuje definice aliasů a pojmenovaných konstant.

Poslední a typicky nejrozsáhlejší sekce je instrukční sekce (angl. *instruction section*). Tato sekce je podobně jako u jazyka IFJcode (viz podkapitola 2.1.2) složena z jednořádkových instrukcí a případných prázdných nebo komentářových řádků. Další členění je poskytnuto pouze instrukcí/definicí `LABEL`, která definuje pojmenované návěští.

Kapitola 3

Možnosti vytváření rozšíření pro editory zdrojového textu

Tato kapitola se zabývá možnostmi a způsoby vytváření rozšíření pro editory zdrojových textů nebo integrovaná vývojová prostředí. Také se zmiňuje o ladění, které je typickou součástí integrovaných vývojových prostředí. Pro editor Visual Studio Code obsahuje podrobné informace o možnostech vytváření rozšíření a popis souvisejících technologií.

3.1 Editory zdrojového textu

Editory zdrojového kódu jsou speciální textové editory určené pro psaní zdrojového kódu, které typicky podporují větší množství programovacích jazyků. Jsou vlastně rozšířenými textovými editory, které jsou schopné rozpoznat podle typu souboru použitý jazyk a nabídnout pro něj základní pomocné funkce jako je např. zvýrazňování syntaxe (angl. *syntax highlighting*) podle definice daného jazyka.

Programátorská práce nicméně vyžaduje více nástrojů než jen jednodušší práci se soubory se zdrojovými texty a jejich lepší zobrazování. Například u překládaných jazyků je nutné zajistit jejich překlad. Výsledný program je potřeba spouštět nebo interpretovat a v případě nesprávného chování je nutné jej ladit. Editory, které pro specifické programovací jazyky doplňují některé z těchto funkcí a různé další, se nazývají integrovaná vývojová prostředí (angl. *Integrated Development Environment*, IDE). Jsou to programy, které obsahují více různých nástrojů v rámci jednoho programu.

Poskytování více funkcí pro programovací jazyk typicky vede k omezení rozšířených funkcí IDE pouze pro malé množství vybraných programovacích jazyků. IDE potom může být určeno specificky pro vývoj v jednom nebo v určité skupině programovacích jazyků. Pokud však IDE nabízí podporu pro větší množství jazyků, je typicky možné si vybrat, které mají být v programu obsaženy. Tento výběr může být proveden při instalaci, ale většina dnes používaných IDE následně umožňuje doplnit podporu pro další jazyky, pokud existuje. Podpora pro další jazyky může být vytvořena samotnými autory IDE, nebo může být její vytváření umožněno širší komunitě, což vyžaduje zveřejnění rozhraní, pomocí kterého je možné přistupovat k interní funkcionalitě editoru.

Následující podkapitoly se zabývají možností rozšíření tří známých integrovaných vývojových prostředí, které nejsou specializované pro určitý jazyk nebo skupinu jazyků. Cílem je především získání přehledu o způsobech vyvíjení rozšíření a dostupných nástrojích, zdroje často pocházejí z dokumentace nebo návodů pro dané IDE.

3.1.1 NetBeans

NetBeans je IDE s otevřeným zdrojovým kódem (angl. *open source*) napsané v jazyce Java. Původně sloužilo pouze pro vývoj v jazyce Java, v dnešní době ale podporuje více jazyků, nejznáměji zřejmě PHP. NetBeans původně vznikl jako studentský projekt v České republice, poté byl odkoupen společností Sun Microsystems, která byla odkoupena společností Oracle, která nakonec projekt NetBeans darovala neziskové organizaci Apache Software Foundation, která v současnosti odpovídá za vývoj IDE [2].

NetBeans je modulární IDE, jehož funkcionalitu je možné rozšířit pomocí zásuvných modulů (pluginů). Základní podporu dalšího programovacího jazyka ve formě syntaktického zvýrazňování (v kontextu NetBeans nazýváno i *syntax coloring*) lze přidat pomocí modulu definujícího lexikální a syntaktický analyzátor pro nový jazyk. Alternativu představoval dnes již zastaralý projekt Generic Language Framework [21], který umožňoval definovat lexikální a syntaktická pravidla jazyka pomocí jednoduchých příkazů a regulárních výrazů ve formátu NBS (NetBeans Schliemann) [20][1]. Současným přístupem je například generování lexikálního analyzátoru (angl. *scanner*) nástrojem JavaCC (*Java Compiler Compiler*) a použití vygenerovaného analyzátoru k implementaci požadovaných funkcí [3].

NetBeans umožňuje i vývoj komplikovanějších programovacích rozšíření v jazyce Java za použití dostupného NetBeans API (*application programming interface*, aplikační programovací rozhraní), které obsahuje i podporu pro ladění[2]. Příklad nezávislého rozšíření přidávající komplexní jazykovou podporu pro platformu NetBeans je projekt `nbPython` pro programovací jazyk Python, tato sada zásuvných modulů však byla vydána pro verzi NetBeans 8[3] a není podporována v dalších verzích.

3.1.2 Eclipse

Eclipse je další IDE s otevřeným zdrojovým kódem vytvořené v jazyce Java. Původně vzniklo jako produkt společnosti IBM, od roku 2004 je ale spravováno neziskovou organizací Eclipse Foundation. IDE Eclipse je opět postaveno na konceptu modularity. Samotný název Eclipse je často používán synonymně s IDE Eclipse, ale projekt Eclipse obsahuje několik částí, které dohromady vytváří IDE, mohou však mít i samostatné využití [9, 4].

Jádrem je platforma Eclipse, což je kolekce nástrojů, které dohromady tvoří obecné IDE. Další komponentou je JDT (*Java Development Tools*, nástroje pro vývoj v jazyce Java), které IDE rozšiřuje o možnosti pro práci s jazykem Java. Pro doplnění podpory pro další jazyky je nutné připojit další komponenty plnící stejný účel pro dané jazyky. Poslední významnou součástí je PDE (*Plug-in Development Enviroment*, vývojové prostředí zásuvných modulů), které poskytuje nástroje k vývoji dalších zásuvných modulů v jazyce Java.

Eclipse nepodporuje oddělenou definici pravidel pro zvýrazňování syntaxe pomocí gramatiky, je nutné ji implementovat jako standardní zásuvný modul [8, 6]. Syntaktický analyzátor (angl. *parser*) je nutné vytvořit odděleně nebo k jeho vygenerování použít existující generátor jako např. již zmíněný JavaCC nebo ANTLR [15]. Podporu pro ladění je poté možné vytvořit jako součást samotného rozšíření nebo s použitím externího ladicího nástroje

[1] Původní zdroje pochází z NetBeans Wiki společnosti Oracle, která již není dostupná, existuje ale nový návod od společnosti Apache https://netbeans.apache.org/tutorials/60/nbm-prolog.html

[2] https://bits.netbeans.org/12.2/javadoc/org-netbeans-api-debugger/overview-summary.html

[3] http://nbpython.org

(angl. *debugger*). Existuje Eclipse projekt poskytující podporu pro komunikaci s externími nástroji s použitím protokolu Language Server[4] (viz podkapitola 3.3.2).

Alternativním přístupem pro vytvoření podpory pro nový jazyk je použití aplikačního rámce (*application framework*) Xtext. Xtext slouží k vytváření celkové programové infrastruktury pro doménově specifické jazyky (*Domain-specific languages*, DSL), tedy jazyky s omezenou oblastí použití nebo speciálním určením. Umožňuje na základě definice jazyka generovat nejen syntaktický analyzátor, ale i další programové vybavení pro práci s jazykem, jako je například zvýrazňování syntaxe nebo překladač [5]. Tyto nástroje je možné vygenerovat přímo jako zásuvný modul pro IDE Eclipse.

3.1.3 Visual Studio Code

Visual Studio Code (VS Code) je editor zdrojových textů vytvořený společností Microsoft. Editor je implementován v jazycích TypeScript a JavaScript a v základu obsahuje širší podporu jen pro tyto dva jazyky. Pro další jazyky poskytuje asociace podle typu souborů a syntaktické zvýrazňování [16]. Základní vlastností editoru VS Code je však jeho jednoduchá rozšiřitelnost, která umožňuje doplnit do editoru další funkce [10]. Další podpora pro mnoho běžných populárních programovacích jazyků je obsažena v oficiálních rozšířeních vytvořených společností Microsoft, jedná se například o rozšíření pro programovací jazyky C/C++. Možnosti a základní informace o implementování rozšíření pro VS Code jsou popsány v oficiálním přehledu [17]. Na rozdíl od obou popsaných IDE je VS Code popsáno pouze jako editor zdrojových kódů, který ovšem podporuje další funkce včetně ladění. VS Code také nepoužívá koncept projektů vytvořených pro různé programovací jazyky, místo toho používá standardní souborový systém, kdy umožňuje otevřít libovolný adresář, pro který případně ukládá lokální nastavení. Pro účely této práce jsou však možnosti editoru VS Code ekvivalentní s integrovanými vývojovými prostředími.

Rozšíření doplňující syntaktické zvýrazňování se vytváří pomocí definice TextMate gramatiky pro daný jazyk. Gramatika jednotlivé úseky zdrojového textu popisuje regulárními výrazy a přiřazuje jim identifikátor. Na základě identifikátoru je jednotlivým úsekům podle vybraného stylu editoru VS Code (*theme*) přiřazován vzhled.

Rozšíření doplňující další funkce jsou vytvářeny v jazyce TypeScript nebo JavaScript. Samotná implementace funkcí pro programovací jazyk však může být napsána v libovolném programovacím jazyce a být spuštěna na tzv. jazykovém serveru (*language server*), se kterým rozšíření VS Code komunikuje pomocí protokolu jazykového serveru (Language Server Protocol, LSP) [18]. Identicky je možné mít externí implementaci ladicího nástroje (*debugger*), se kterou rozšíření komunikuje pomocí protokolu adaptéru ladění (Debug Adapter Protocol, DAP) [19].

3.1.4 Závěr

Všechna tři zkoumaná IDE teoreticky umožňují, aby pro ně bylo vytvořeno rozšíření, které by poskytovalo všechny požadované funkce, tedy syntaktické zvýrazňování, spouštění a ladění. NetBeans se však zdá být nejméně dobrou volbou, protože v současné době neexistuje žádné externí rozšíření, které by bylo stále vyvíjené a poskytovalo všechny požadované funkce. Kvůli přesunu vývoje ze společnosti Oracle do Apache Software Foundation je také horší přístup k dokumentaci a informacím o dostupnosti/funkčnosti nástrojů. Ze zbývajících dvou IDE bylo zvoleno Visual Studio Code, především kvůli dostupnosti velkého množství

[4] https://www.eclipse.org/community/eclipse_newsletter/2017/may/article3.php

aktuálních materiálů, návodů a dokumentace. Visual Studio Code se navíc v poslední době těší značné popularitě, jak vyplývá například z průzkumu Stack Overflow v roce 2019[5], ze kterého VS Code vyšel mezi dotázanými jako nejpopulárnější editor. Poslední výhodou je protokol Debug Adapter, který byl vytvořen společností Microsoft a v rámci editoru se používá k implementaci ladicích funkcí. Tento přístup poskytující standardizované rozhraní pro implementaci ladicích funkcí, které jsou oddělené od samotného editoru, se jeví jako vhodný pro účely implementace ladění interpretovaných jazyků mezikódu. Protože interprety jazyků mezikódu, které se budou rozšiřovat o ladicí nástroje, mají různé implementační jazyky odlišné od implementačního jazyka editoru.

3.2 Ladění

Jedním z hlavních cílů této práce je vytvořit podporu pro ladění mezikódů. Tato podkapitola se proto zabývá procesem ladění programů a pojmem ladicího nástroje (*debugger*). Podle [22] je možné ladicí nástroj definovat jako nástroj sloužící k izolování a odstraňování chyb v softwarů (*bugs*). Zároveň jsou to ale nástroje sloužící k analýze chování programu a usnadnění jeho pochopení pomocí interakce s ním.

Aby bylo možné provádět ladění programu, tak je nutné, aby nad ním převzal ladicí nástroj kontrolu. Existují dva scénáře, jak toto může nastat. První možností je spuštění laděného programu ladicím nástrojem, v tomto případě ladicí nástroj přímo zahájí provádění programu, který je tak od začátku ladění pod kontrolou ladicího nástroje. Druhou možnost představuje připojení (*attach*), kdy program, který má být laděn, již běží, a ladicí nástroj se k němu musí připojit a teprve poté získat kontrolu nad jeho řízením. Pro připojení k běžícímu programu je typicky nutné využít funkcí operačního systému.

Ladicí nástroj může být aplikace příkazové řádky, která umožňuje interakci s použitím jen textového vstupu. Současným standardem a dlouhodobým trendem je však implementace ladicích nástrojů v rámci aplikací s grafickým uživatelským rozhraním. Takové aplikace mohou teoreticky sloužit pouze k ladění, ale v praxi se jedná téměř vždy o integrované vývojové prostředí, které mimo jiné obsahuje podporu i pro editaci a případný překlad daného programovacího jazyka. Vývojové prostředí poté v ladicím módu umožňuje uživateli ovládat ladění programu pomocí interakce s grafickým rozhraním a zobrazuje data získaná z laděného programu v rámci různých pohledů. Ladicí nástroje je podle [22] možné mimo jiné dále dělit na nástroje pracující na strojové úrovni a na zdrojové (symbolické) úrovni. Relevantní pro tuto práci je druhá skupina, kdy ladicí nástroj mapuje prováděný kód na zdrojový text laděného programu, a to konkrétně ve formě ladicího nástroje obsaženého v rámci integrovaného vývojového prostředí.

Spuštěný ladicí nástroj ovládající vykonávání programu poskytuje uživateli různá data, která jsou standardně rozdělena do několika pohledů, jak popisuje [22]. Každý pohled poskytuje informace o určitém aspektu probíhajícího ladění programu.

Prvním je zdrojový pohled, který zobrazuje kód programu ve zdrojovém formátu. Prakticky se tedy jedná o okno editoru zdrojových textů. Pro účely ladění poskytuje interakci ve formě nastavování bodů přerušení (*breakpoint*) sloužících k pozastavení vykonávání kódu na dané pozici. Tento pohled také ukazuje pozici, ve které se právě nachází pozastavený laděný program, typicky zvýrazněním daného řádku.

Dalším pohledem je pohled zásobníku volání, který poskytuje informace o posloupnosti (resp. zanoření) volaných funkcí. Ideálně by tento pohled měl poskytovat informace o vo-

[5] https://insights.stackoverflow.com/survey/2019

laných funkcích, jako jsou jejich jména a parametry, tento pohled by měl také poskytovat propojení se zdrojovým pohledem, ve kterém by se měl odkazovat na danou funkci.

Pohled bodů přerušení poskytuje přehled všech bodů přerušení nastavených ve zdrojovém pohledu. Jak již bylo zmíněno u zdrojového pohledu, body přerušení značí místa, ve kterých má dojít k pozastavení vykonávání programu. Ne každý bod přerušení ale musí být aktivní a sloužit tomuto účelu. Neaktivní body přerušení pouze označují řádek a mohou být změněny na aktivní bod přerušení. Jinak s nimi není prováděna žádná další činnost. Důležitější je koncept neverifikovaného bodu přerušení, tento bod přerušení je běžným aktivním bodem přerušení, který by měl vést na pozastavení vykonávání programu, ale jeho funkce není v laděném programu k dispozici. Popsané chování může být způsobeno změnou ve zdrojovém kódu, která ještě nebyla v programu reflektována novým překladem zdrojového kódu, je-li toto nutné. Jinak to může znamenat, že na daném řádku není možné nastavit bod přerušení.

Pro sledování hodnot proměnných existují dva pohledy. První je pohled proměnných, který poskytuje přehled proměnných existujících ve vykonávaném programu ve chvíli, kdy je jeho vykonávání pozastaveno. Druhý související pohled je pohled sledovaných proměnných, který poskytuje identické informace, ale pouze o vybraných proměnných.

Posledním relevantním pohledem je vyhodnocovací pohled, který umožňuje zadávání zdrojového kódu, který je vyhodnocen v kontextu laděné aplikace. Tento pohled tedy poskytuje interaktivní vykonávání kódu, které však ovlivňuje laděnou aplikaci. Slouží především ke změně stavu programu v libovolném bodě ladění, aby bylo možné dále sledovat chování programu v modifikovaném stavu bez nutnosti zahajovat ladicí proces znovu.

Existují i další pohledy, například pohled zobrazující program na úrovni strojového kódu. Tento pohled a jiné specializované pohledy však pro tuto práci zabývající se laděním interpretovaných jazyků nemají význam.

3.3 Rozšíření pro editor Visual Studio Code

Tato podkapitola se zabývá vytvářením rozšíření pro editor zdrojových textů VS Code. Jak již bylo popsáno, princip rozšiřitelnosti je základní koncept editoru VS Code. Samotné jádro aplikace je textový editor odpovídající webovému editoru Monaco, kolem kterého jsou doplněny další funkce rozšiřující vlastnosti editoru a vytvářející program odpovídající integrovanému vývojovému prostředí. Podpora pro mnohé jazyky je implementována v podobě oficiálních rozšíření přímo společností Microsoft, ale možnosti vytváření rozšíření nejsou nijak omezeny.

Implementačními jazyky editoru VS Code jsou JavaScript a TypeScript, přičemž prakticky se pro psaní zdrojového kódu používá převážně druhý jmenovaný. Jazyk TypeScript byl vytvořen společností Microsoft, která ho nadále vyvíjí. Jazyk TypeScript představuje nadmnožinu jazyka JavaScript, který rozšiřuje o striktní typový systém, čímž přidává možnost statické typové kontroly, také doplňuje další jazykové vlastnosti jako jsou například rozhraní (*interface*). Zdrojový kód napsaný v jazyce TypeScript je nutné přeložit do výsledného jazyka, kterým je JavaScript, který následně může být standardním způsobem interpretován. Pro překlad jazyka TypeScript existuje překladač TypeScript Compiler použitelný jako program příkazové řádky `tsc`.

Rozšíření pro Visual Studio Code je projektem v jazyce JavaScript mimo webové prostředí. Pro jeho vývoj je nutné běhové prostředí `Node.js`, což je aplikační rozhraní obsahující samostatnou implementaci interpretu jazyka JavaScript `V8` vytvořenou společností Google. Běhové prostředí `Node.js` umožňuje interpretovat kód v jazyce JavaScript mimo prostředí

webových prohlížečů. Pro správu projektů se používá správce balíčků, což je program sloužící k instalaci a další správě samostatných balíčků kódu – modulů. Výchozím správcem balíčků pro běhové prostředí `Node.js`, který je spolu s ním distribuovaný, je nástroj `npm`. Existují ale i další alternativy jako např. `yarn`. Pro vývoj rozšíření je nutné mít nainstalován správce balíčků, který zajistí stažení potřebných závislostí pro vytvářené rozšíření [17, 10], jedná se v základu přinejmenším o moduly umožňující přístup k API VS Code.

Všechny možné typy rozšíření jsou popsány na oficiální stránce [17]. Jednoduše je můžeme rozdělit do dvou kategorií na programová a neprogramová. Neprogramová rozšíření jsou definice gramatiky jazyka, které jsou použité k syntaktickému zvýrazňování, a definice stylu pro editor (*theme*). Programová rozšíření jsou napsána v jazyce TypeScript a umožňují přidat prakticky libovolnou funkcionalitu za použití API VS Code, příkladem programového rozšíření může být spouštění kódu v integrovaném terminálu.

Každé rozšíření musí v kořenovém adresáři obsahovat definiční soubor (*extension manifest*) `package.json`, který definuje základní informace o rozšíření jako je název, autor atd. a především v položce `contributes` definuje, jakou funkcionalitu rozšíření přidává (definována jednotlivými body rozšiřitelnosti, *contribution points*), pro programové rozšíření navíc definuje, jakou akcí se má rozšíření aktivovat.

Obsah definičního souboru rozšíření je nadmnožinou běžného souboru `package.json`, který je standardním definičním souborem pro správu projektu pomocí správce balíčků `npm`[6]. Balíčky, na kterých projekt závisí jsou tedy standardně popsány v polích pro deklaraci závislostí (`dependencies` a `devDependecies`) a správce balíčků zajišťuje jejich stažení.

Pro uložení dalších textových informací se používá značkovací jazyk `Markdown`. Text k zobrazení u publikovaného rozšíření je obsažen v souboru `README.md` a historie změn je v souboru `CHANGELOG.md`. Další soubory jsou dané určením vytvářeného rozšíření.

Soubor, který je vstupním bodem rozšíření, je standardně pojmenován `extension.ts`. Jméno je ale libovolné a je definováno pod klíčem `main` v souboru `package.json`. Tento soubor obsahuje funkce `activate` a `deactivate`, které provádějí aktivaci a případnou deaktivaci rozšíření.

Zahájení vývoje rozšíření je usnadněno pomocným nástrojem `yo code`, což je Yeoman[7] generátor pro rozšíření editoru VS Code. Nástroj umožňuje vygenerovat výchozí soubory a strukturu pro různé obecné typy rozšíření.

3.3.1 Definice syntaxe jazyka a syntaktické zvýrazňování

Syntaxe nového jazyka je popsána TextMate gramatikou ve formátu JSON nebo XML. TextMate gramatiky vznikly jako způsob popisu syntaxe jazyků pro editor TextMate pro operační systém Mac [7, 14]. Originální TextMate gramatiky pro editor TextMate jsou ve formátu Property List, který podporuje více reprezentací včetně XML. Formát používaný v dokumentaci ale vychází z formátu Property List vzniklého v operačním systému NeXTSTEP, tento formát se podobá formátu JSON, ale není s ním identický. TextMate gramatiky mají dané schéma, které definuje jaké klíče a jim příslušející hodnoty mohou existovat a jaká je struktura souboru.

Gramatiky pomocí regulárních výrazů označují části textu, kterým přiřazují identifikátory. TextMate gramatiky fungují řádkově, jedno pravidlo tedy nemůže samo o sobě označit víceřádkový blok textu. Je však možné provést selekci bloku pomocí označení jeho začátku

[6] https://docs.npmjs.com/cli/v7/configuring-npm/package-json
[7] https://yeoman.io/

a konce. Zpracování regulárních výrazů z TextMate gramatiky je prováděno knihovnou Oniguruma.

Pravidla pro označování textu jsou definována jako pole objektů v položce `patterns`. Každý objekt obsahuje položku `name`, která udává identifikátor označeného textu. Základní pravidlo pro označení části textu v řádku je položka `match`, která obsahuje regulární výraz. Pro označení většího bloku textu se používají dvě položky `begin` a `end`, které obsahují regulární výrazy konce a začátku označení. Pokud se použije toto blokové označení, je možné v rámci něj definovat další označování pomocí zanořené položky `patterns` obsahující další označovací pravidla.

Identifikátory (případně rámce, *scopes*) definované v položce `name` mají textový formát složený z několika částí oddělených tečkami, kde část nalevo představuje nejobecnější označení. Identifikátory mohou být teoreticky libovolné, v praxi ale existují standardní označení, která je doporučené používat. Identifikátory používané pro VS Code vycházejí z identifikátorů používaných v TextMate [17]. Příkladem identifikátoru může být `constant.numeric.java`, který ve směru zleva označuje konstantu, která má číselný typ a patří do jazyka Java.

Další složkou definice jazyka je konfigurační soubor ve formátu JSON, který definuje další vlastnosti jazyka, jako jsou např. symboly použité pro komentáře a závorky, které jsou použity k doplnění funkcí editoru, jako je skrytí nebo zakomentování části textu. Další část konfigurace je obsažena v souboru `package.json`, kde se definuje přípona souborů, pro které je jazyk definován, případně identifikace jazyka podle prvního řádku souboru.

Stylové předpisy VS Code (*themes*) pro jednotlivé identifikátory definují vzhled. Soubory stylových předpisů obsahují identifikátory a přiřazují jim styl. Pro části textu označené identifikátorem se poté hledá nejlépe odpovídající pravidlo v aktivním stylovém předpisu, které se použije k označení části textu. Nejlépe odpovídající je pravidlo, jehož identifikátor se ve směru zleva nejvíce shoduje s identifikátorem části textu. Pokud je tedy například část textu označena `constant.numeric.java` a existují styly pro `constant` a `constant.numeric`, použije se druhý z nich.

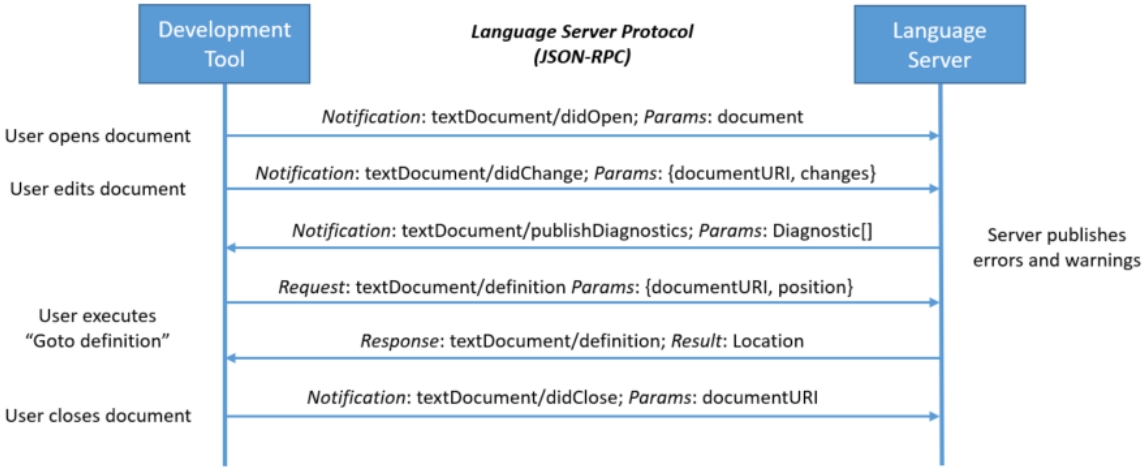
3.3.2 Protokol Language Server

Historicky byla rozšíření úzce vázána na editor zdrojového kódu nebo integrované vývojové prostředí, pro které byla vyvinuta. Toto řešení je nutné u částí rozšíření, které přímo modifikují daný editor, například přidáváním dalších oken, pohledů, nastavení a podobně. Velká část rozšíření doplňujících podporu pro specifický programovací jazyk však poskytuje poměrně standardní funkce, jako je např. analýza zdrojového kódu a detekce chyb, zobrazování definic nebo odkazů na symboly jazyka atp. Pro tyto rozšíření není úzké navázání na jeden editor nutné a přináší zbytečné komplikace při snaze o přidání identických funkcí pro další editory.

Společností Microsoft vyvinutý protokol Language Server (LSP) je snahou o vyřešení výše zmíněného problému [18]. Protokol definuje typy, obsah a formát zpráv, které si mezi sebou vyměňuje editor a server, který implementuje některé ze standardních funkcí. Při použití protokolu Language Server tedy pro zpřístupnění funkcí pro nový editor stačí, aby editor i server implementovaly komunikaci pomocí tohoto protokolu. Editor je poté odpovědný za případné spuštění serveru a na základě svého stavu nebo akcí uživatele odesílá zprávy serveru a zobrazuje přijaté odpovědí. Ukázka komunikace mezi editorem a jazykovým serverem je na obrázku 3.1, kde je možné vidět, jak editor informuje server o akcích uživatele, a následně požadavek na server, na který je vrácena odpověď.

Protokol Language Server je založený na bezstavovém protokolu JSON-RPC (*JSON Remote Procedure Call*, tedy vzdálené volání procedur používající JSON jako formát zpráv). Protokol JSON-RPC definuje formát obsahu zasílané zprávy, kterým je JSON objekt. Pro objekt zprávy dále definuje povinné atributy. Protokol Language Server používá JSON-RPC protokol verze 2 a rozšiřuje definice JSON objektu zpráv o další položky.

Samotná LSP zpráva je složena ze dvou sekcí, hlavičkové sekce a obsahu zprávy ve formě JSON objektu. Hlavičková sekce obsahuje dvě povinné hlavičky, první definující délku obsahu zprávy a druhou definující typ MIME a kódování obsahu. Hlavičková sekce je od samotného JSON obsahu zprávy oddělena dvěma znaky konce řádku identicky jako v protokolu HTTP, jedná se tedy o řetězec `"\r\n\r\n"`.



Obrázek 3.1: Ukázka komunikace při použití protokolu Language Server. Editor odesílá informace o proběhlých událostech a požadavek na načtení nějaké informace, kterou server poskytne. Převzato z [18].

3.3.3 Protokol Debug Adapter

Protokol Debug Adapter je další protokol vytvořený společností Microsoft pro komunikaci s externími nástroji. Motivace pro jeho vznik je podobná jako v případě protokolu Language Server, rozdíl je v tom, že protokol Debug Adapter slouží pro propojení editoru s ladicím nástrojem [19].

Protokol Debug Adapter je podobný protokolu Language Server, pro komunikaci opět používá zprávy složené z hlavičkové sekce a obsahu ve formátu JSON. Na rozdíl od protokolu Language Server však JSON zprávy nejsou založené na specifikaci protokolu JSON-RPC.

Hlavičková sekce zprávy protokolu Debug Adapter obsahuje jediné povinné pole. Jedná se o pole obsahující délku obsahu zprávy, které má identický formát jako u protokolu Language Server. Začíná řetězcem `Content-Length:`, který je následovaný mezerou a číselnou hodnotou, která značí délku JSON obsahu v bajtech. Dále je zprávou myšlen už jen JSON obsah zprávy bez hlavičky.

Formát JSON zpráv je definován ve specifikaci protokolu pomocí rozhraní v jazyce TypeScript. Základem pro všechny zprávy protokolu Debug Adapter je zpráva protokolu `ProtocolMessage`, která definuje povinná pole obsahující sekvenční číslo zprávy (`seq`) a její typ (`type`). Protokol definuje tři základní typy zpráv, které mají definované další povinná a volitelná pole.

Typy zpráv

Prvním typem zprávy je požadavek (`request`), tento typ zprávy posílá editor (klient) adaptéru ladění (ladicí nástroj komunikující protokolem Debug Adapter), když vyžaduje provedení nějaké operace související s laděním. Může se jednat o reakci na interakci od uživatele, například pokračování vykonávání programu, nebo se může jednat o akci iniciovanou editorem, například načtení hodnot proměnných. Jediným požadavkem, který může být odeslán v opačném směru je požadavek na spuštění v terminálu, kterým adaptér ladění žádá, aby editor spustil zadaný příkaz.

Dalším typem zprávy je odpověď (`response`), což je zpráva, která je vždy navázaná na přijatý požadavek a je odesílána v reakci na něj. Každý DAP požadavek (požadavek definovaný v protokolu Debug Adapter) má přiřazenou odpověď.

Posledním typem zpráv je událost (`event`), což jsou zprávy posílané adaptérem ladění editoru. Na rozdíl od zpráv typu odpověď, které jsou synchronní a reagují na přijatý požadavek, jsou zprávy typu událost asynchronní. Jejich smyslem je informovat o nějaké události, která nastala při procesu ladění. Události samy o sobě nemají přiřazenou odpověď, mohou ale vést k zaslání požadavku, na který poté přijde odpovídající odpověď. Podobně mohou být události vyvolány vykonáváním požadavku. Typickým příkladem situace, kdy dojde k odeslání události je například moment, kdy se při provádění kódu narazí na místo, kde má dojít k pozastavení vykonávání programu (*breakpoint*). Podobně se ale informuje například i o spuštění procesu laděného programu, nebo o spuštění a ukončení vláken.

Následující podkapitoly se zabývají konkrétními zprávami protokolu Debug Adapter, jejich funkcí a významem v rámci komunikace. V závorce je vždy obsažen původní název požadavku nebo události.

Inicializační sekvence

Komunikace protokolem Debug Adapter je dle specifikace [19] zahájena odesláním inicializačního požadavku (`initialize`), který obsahuje podporované vlastnosti editoru a jeho konfiguraci. Odpověď obsahuje podporované vlastnosti adaptéru, které definují, jaké další zprávy kromě základních může klient adaptéru zasílat. Po odeslání odpovědi odesílá adaptér událost ukončení inicializace (`initialized`), která signalizuje, že klient může zasílat konfigurační požadavky.

Konfigurační požadavky jsou požadavky na nastavení různých typů bodů přerušení. Hlavní je požadavek na nastavení obyčejných bodů přerušení (`breakpoints`). Pro každý soubor s nastavenými body přerušení ve složce právě otevřené v editoru je zaslán jeden požadavek. Pokud není v odpovědi na inicializační požadavek provedeno žádné nastavení, tak je konfigurace zakončena požadavkem na nastavení typů výjimek, při kterých má dojít k přerušení vykonávání programu (`setExceptionBreakpoints`). Pokud je podporován požadavek na ukončení konfigurace (`configurationDone`), tak je konfigurace ukončena tímto požadavkem. Zasílání požadavku na konfiguraci přerušení na výjimkách je poté možno potlačit explicitním nastavením prázdného pole podporovaných filtrů pro přerušení na výjimkách.

Zahájení ladění programu

Pro zadání programu, který má být laděn, existují dva požadavky. Pro účely této práce má význam požadavek na spuštění programu (`launch`), který adaptéru předá cestu k programu, jenž má adaptér následně pod svojí kontrolou spustit. Druhý požadavek je požadavek na

připojení k již běžícímu programu (`attach`), který pro jazyky mezikódu není podporován. Požadavky na zahájení ladění programu souvisí s inicializací ladění, jelikož je nutné mít přístup k laděnému programu, aby bylo možné provádět jeho konfiguraci. Klient VS Code zasílá požadavek na spuštění programu jako druhý po inicializačním požadavku. Skutečné spuštění programu je ale nutné synchronizovat až s dokončením inicializační sekvence, kdy jsou nastavené všechny body přerušení. Případné informace o spuštěném programu mohou být zaslány pomocí události o procesu (`process`) nebo vlákně (`threads`).

Vyhodnocovací sekvence

Následující požadavky nejsou ve specifikaci [19] přímo popsány jako sekvence, prakticky ale tvoří sekvenci používanou k vyhodnocení stavu laděného programu. Pokud dojde k zastavení vykonávání programu nebo provedení akce, kterou klient vyhodnotí jako měnící stav programu, tak klient pomocí tří požadavků načte informace o stavu programu. Ukázka komunikace, při které jsou vráceny dva rámce proměnných a načteny proměnné v nich, je na diagramu 3.2.

Jako první je odeslán požadavek na získání obsahu zásobníku volání (`stackTrace`), jeden pro každé laděné vlákno. Požadavkem klient získá informaci o volaných funkcích a řádku, na kterém došlo k pozastavení, který může zvýraznit v editoru. Zásobník volání je zobrazen v pohledu zásobníku volání (`CALL STACK`), jak je vidět na ukázce 3.3.

Poté následuje dotaz na rámce proměnných (`scopes`), který získá informace o existujících rámcích proměnných v programu (např. globální proměnné a lokální proměnné funkce). Každý rámec je definovaný číselným identifikátorem proměnných (`variablesReference`) o rozsahu 32 bitů. Získané rámce jsou následně zobrazeny v pohledu proměnných (`VARIABLES`), kde je uživatel může rozbalit pro zobrazení proměnných a jejich hodnot.

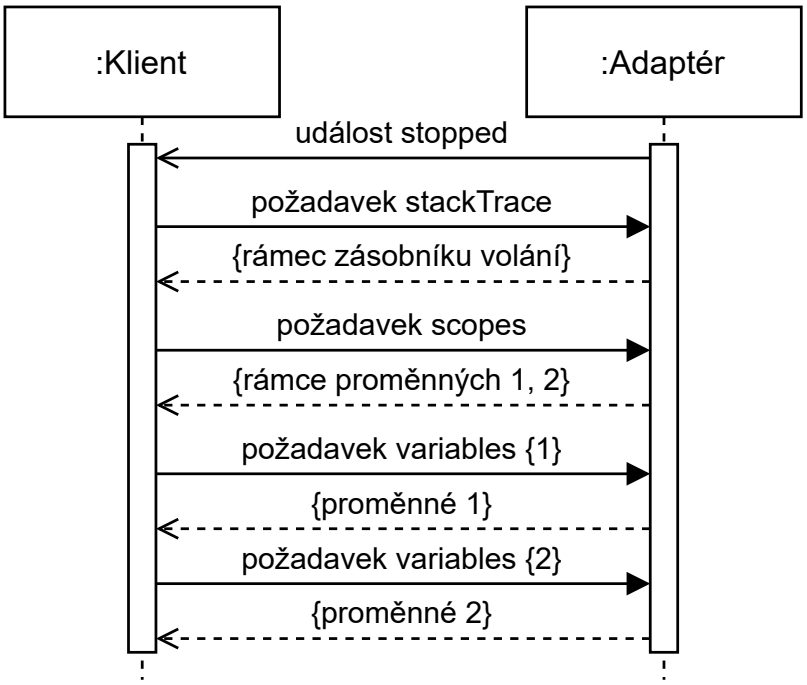
Pro každý viditelný rámec proměnných je nakonec zaslán požadavek na získání proměnných (`variables`). Odpovědí na něj je pole proměnných (objekty odpovídající rozhraní `Variable`, které pomocí řetězce reprezentují skutečné hodnoty v programu). Proměnné mohou obsahovat identifikátor proměnných (stejně jako rámce proměnných), který může být použit v dalším požadavku na získání proměnných. Tímto způsobem je možné reprezentovat složené proměnné jako např. pole nebo objekty, které je možné dále rozbalit pro načtení jejich vnitřních hodnot.

Vyhodnocování výrazů

Pro vyhodnocování výrazů existuje požadavek (`evaluate`), který zasílá řetězec, který má být vyhodnocen adaptérem v kontextu laděného programu. Požadavek obsahuje název kontextu, ze kterého požadavek na vyhodnocení vzešel. Odpověď na požadavek je DAP proměnná stejně jako při získávání proměnných obsažených v rámci.

První možností je nastavení sledování hodnoty proměnné v pohledu sledovaných proměnných (`WATCH`). V tomto kontextu se očekává vyhodnocování proměnných, které jsou zobrazeny paralelně s přehledem veškerých proměnných. Vyhodnocení proměnné může často skončit chybou, v takovém případě je zobrazen chybový text místo hodnoty proměnné.

Veškerý text zadaný do ladicí konzole (`DEBUG CONSOLE`) je odeslán jako vyhodnocovací požadavek s kontextem REPL. Možnosti vyhodnocování v tomto kontextu jsou libovolné a jsou dány implementací adaptéru. Typicky je možné vyhodnocovat další výrazy v jazyce laděného programu, které mění jeho vnitřní stav. Případný výstup provedených instrukcí může být odeslán pomocí výstupní události (`output`).



Obrázek 3.2: Ukázka části komunikace mezi editorem VS Code (klient) a ladicím adaptérem zahájené pozastavením laděného programu. Předpokládá se, že obsah obou rámců je v editoru rozbalen, tzn. jsou zobrazeny proměnné. Čísla značí identifikátory proměnných.

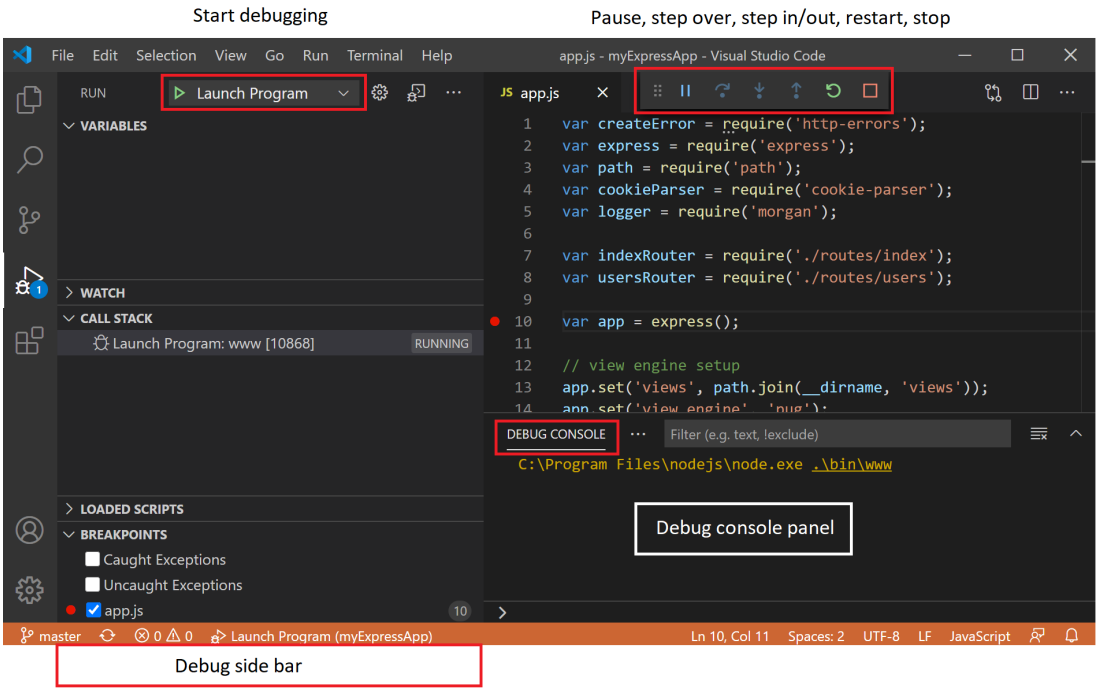
Požadavky řídicí provádění laděného programu

Následující požadavky implementují základní funkce typické pro každý ladicí nástroj zajišťující ovládání laděného programu. Jsou dostupné pomocí ikon v ovládacím panelu ladění, který je vidět na obrázku 3.3 vpravo nahoře. Ovládací panel umožňuje použití tlačítek na základě provedených akcí a stavu ladění, některá tlačítka tak mohou být zablokována a odpovídající požadavky neproveditelné.

První je požadavek na pokračování vykonávání programu (`continue`), který zajistí pokračování provádění laděného programu, dokud neskončí nebo nedojde k bodu přerušení. Pokud probíhá vykonávání programu, tak jsou blokována tlačítka na provádění kroku a místo tlačítka pro pokračování vykonávání je dostupné tlačítko pro požadavek na pozastavení vykonávání programu (`pause`). Pozastavení pouze nastaví požadavek pro pozastavení laděného programu, samotné pozastavení je následně reprezentováno odesláním události pozastavení. Pokud pozastavení programu není podporováno, tak je díky tomuto chování možné pouze odeslat odpověď a neprovést žádnou další operaci.

Existuje několik tlačítek na provedení kroku. Pozastavení po dokončení kroku je opět oznámeno zasláním události a odpověď je pouze potvrzením požadavku. Po dobu jeho provádění jsou také identicky blokována tlačítka. Prvním požadavkem je obyčejný krok (`next`), který provede minimální možnou jednotku jazyka (typicky řádek, který ale může obsahovat i více po sobě následujících výrazů), pro jazyky mezikódu se jedná o jediný řádek s instrukcí.

Následují požadavky na provedení kroku vstupujícího do případné funkce (`stepIn`) a na vystoupení z prováděné funkce (`stepOut`). Pokud tyto požadavky nejsou podporovány, tak je kvůli blokování ostatních tlačítek není možné implementovat jako prázdné operace. Podle specifikace [19] v takovém případě chování požadavku na provedení kroku do případné



Obrázek 3.3: Ukázka pohledu ladění v editoru Visual Studio Code. Převzato z [16].

funkce odpovídá provedení obyčejného kroku a požadavek na vystoupení z prováděné funkce odpovídá požadavku na pokračování vykonávání programu.

Předposlední tlačítko umožňuje restart ladění, požadavek na restart není výchozí a pokud není podporován, tak je restart implementován ukončením a novým spuštěním ladění. Poslední tlačítko odesílá požadavek na ukončení ladění (resp. na odpojení klienta a ukončení ladění, požadavek `disconnect`).

Další požadavky

Doposud popsané požadavky jsou základní požadavky implementující běžné ladicí funkce a komunikaci. Existují však i další požadavky, které klient může používat jen, pokud adaptér v odpovědi na inicializační požadavek nastaví, že odpovídající vlastnost podporuje. Příkladem je již zmíněný požadavek na ukončení konfigurace, který zjednodušuje konfigurační sekvenci. Další běžný požadavek je požadavek na nastavení hodnoty proměnné, který umožňuje proměnným v pohledu proměnných nastavit novou hodnotu.

3.3.4 Doplnění podpory pro ladění do rozšíření

Definiční soubor musí obsahovat definice funkcionality, kterou přidává, jak již bylo popsáno na začátku podkapitoly 3.3. Pro účely ladění musí definovat nový typ ladění (typicky odpovídající identifikátoru laděného jazyka) a případně další související konfiguraci, jako je doplnění dalších položek, které jsou podporovány v konfiguraci nového typu ladění a obsah výchozí konfigurace ladění.

Existují tři možné přístupy ke komunikaci s adaptérem ladění [17]. První je implementace ladění pomocí třídy implementující protokol DA (Debug Adapter) v jazyce Type-

Script, kde se implementují funkce pro vykonání jednotlivých požadavků. VS Code klient poté může přímo přistupovat k API adaptéru. Další možností je spuštění adaptéru jako externího procesu, což umožňuje spuštění adaptéru v libovolném programovacím jazyce. S externím adaptérem následně klient komunikuje pomocí standardního vstupu a výstupu. Poslední možností je spuštění adaptéru jako externího serveru, se kterým je komunikováno pomocí protokolu TCP, což je přístup, který byl v této práci zvolen, jak je popsáno v podkapitole 4.3.

Statická hodnota definující přístup k adaptéru může být definována přímo v definičním souboru (např. cesta k implementaci v jazyce TypeScript). Jinak je nutné definovat v kódu rozšíření tovární třídu, která vrací objekt definující přístup k adaptéru (např. port právě spuštěného serveru).

Výchozí chování editoru VS Code předpokládá použití ladicí konzole jako hlavního pohledu pro komunikaci s laděným programem. V tomto případě je program prováděn na pozadí a pomocí ladicí konzole je zobrazován veškerý jeho výstup a mohou být vyhodnocovány další příkazy. Text zadávaný do ladicí konzole je syntakticky zvýrazňován podle pravidel pro právě otevřený soubor. Ladicí konzole ovšem neumožňuje předání vstupu pro laděný program.

Další možností je spuštění laděného programu ve viditelném terminálu pomocí API pro práci s integrovanými terminály v editoru VS Code. Jako referenční chování je zde použito oficiální rozšíření pro jazyk Python[8], kde je toto chování výchozí. Spuštěný program poté běží primárně v terminálu, který je použitý pro standardní vstup a výstup. Ladicí konzole umožňuje vyhodnocování dalších instrukcí, jejichž výstup je přesměrován do konzole.

3.3.5 Existující rozšíření doplňující ladicí nástroj pro interpretované jazyky

Jako příklad implementace rozšíření, které doplňuje ladění, je možné použít již existující rozšíření doplňující ladicí nástroj pro některý interpretovaný jazyk.

Podpora pro ladění jazyka **CSCS**[9] je podobná podpoře pro jazyky mezikódu, jelikož se jedná o jazyk vykonávaný interpretem v jazyce C# [10]. Adaptér je implementován v jazyce TypeScript a v rozšíření je zadán jako externí implementace používající interpret `Node.js`. Adaptér používá implementaci protokolu DA pro jazyk TypeScript a v rámci metod pro jednotlivé požadavky komunikuje s ladicím serverem pro jazyk CSCS, který je nutné mít spuštěný na pozadí. Výstup laděného programu je zobrazován v ladicí konzoli.

Podobná implementace je použita pro jazyk **PHP**[10]. Na pozadí je spuštěn server Xdebug, což je existující ladicí nástroj pro jazyk PHP. Adaptér implementovaný v jazyce TypeScript provádí požadavky pomocí komunikace se serverem.

Ladění jazyka **Python** používá adaptér implementovaný jako server přímo v jazyce Python. Při výchozím nastavení je laděný program adaptérem spuštěn pomocí spouštěcího skriptu v jazyce Python v integrovaném terminálu editoru (zasláním požadavku na spuštění v terminálu). Server poté komunikuje se spuštěným procesem přes TCP pomocí protokolu DA a laděný program může využívat terminál pro vstup a výstup. Ladění je implementováno pomocí nástroje `debugpy`, který zajišťuje komunikaci pomocí protokolu DA a interně používá další ladicí nástroj pro jazyk Python `pydevd`.

[8] https://marketplace.visualstudio.com/items?itemName=ms-python.python
[9] https://marketplace.visualstudio.com/items?itemName=vassilik.cscs-debugger
[10] https://marketplace.visualstudio.com/items?itemName=felixfbecker.php-debug

Spouštění laděného programu v integrovaném terminálu je použito také u podpory pro jazyk **Julia**[11]. Samotný adaptér je implementován v jazyce TypeScript a při zahájení ladění spustí v integrovaném terminálu laděný program pomocí spouštěcího skriptu v jazyce Julia. Adaptér poté komunikuje se spuštěným programem pomocí protokolu JSON-RPC skrze pojmenované roury.

3.3.6 Publikování rozšíření

Pro správu a publikování rozšíření slouží nástroj příkazové řádky `vsce` (Visual Studio Code Extensions) dostupný jako `npm` modul. Nástroj slouží k vytváření archivu s rozšířením a jeho zveřejnění. Před vytvořením archivu kontroluje, jestli adresář s rozšířením obsahuje všechny povinné soubory a povinné položky v definici rozšíření (jako je např. identifikace autora).

Zveřejněná rozšíření jsou dostupná skrze stránky pro distribuci rozšíření pro editor VS Code (Visual Studio Code Marketplace). Rozšíření jsou spravována pomocí služby `Azure DevOps` [17]. Publikování rozšíření vyžaduje validní přístupový token (Personal Access Token). Vytvoření přístupového tokenu vyžaduje založení `Azure DevOps` účtu a vytvoření organizace v něm. Následně se vygeneruje přístupový token s právy pro distribuci rozšíření, který je použit pro vytvoření vydavatele (`publisher`).

Přístupový token je poté možné použít pomocí `vsce` k přihlášení jako vytvořený vydavatel a následně mohou být zveřejňována rozšíření, která mají definovaného tohoto vydavatele. Po zveřejnění je rozšíření dostupné ze stránek pro rozšíření nebo přímo z editoru. Nástroj `vsce` je dále možné používat k publikování nových verzí rozšíření a případně k odstranění zveřejněného rozšíření. Alternativní metodou distribuce rozšíření je použití vytvořeného archivu rozšíření, což může být vhodné použít např. pro testování před zveřejněním nové verze.

[11] https://marketplace.visualstudio.com/items?itemName=julialang.language-julia

Kapitola 4

Návrh rozšíření pro editor Visual Studio Code

Cílem práce je vytvořit podporu pro jazyky IFJcode a VYPcode, ke kterým je navíc připojen jazyk IPPcode. Vzhledem k podobnosti mezi jazyky IFJcode a IPPcode a návaznosti předmětů, které s nimi pracují, je vhodné spojit podporu pro ně. Pro jazyk VYPcode bude vytvořeno oddělené rozšíření. Implementace podpory pro jazyk VYPcode bude využívat a rozšiřovat implementaci existujícího interpretu pro jazyk VYPcode vytvořeného Ing. Radimem Kocmanem, který pro tyto účely dal k dispozici jeho zdrojový kód.

Pro všechny jazyky je nutné vytvořit definici pomocí TextMate gramatiky, interpret rozšířený o funkce nutné k ladění a adaptér ladění komunikující pomocí protokolu Debug Adapter. Všechny funkce budou nakonec propojeny s editorem VS Code v rámci dvou vytvořených rozšíření.

4.1 Definice gramatik

Prvním krokem k vytvoření podpory pro jazyky je vytvoření TextMate gramatiky, která umožní zvýrazňování syntaxe jazyků v editoru. Syntaxe jazyků IFJcode a IPPcode je identická, proto je možné začít s vytvořením gramatiky pro jeden jazyk a následně ji pouze upravit změnou definice hlavičky a změnou identifikátoru jazyka pro další verze jazyků. Vytváření gramatiky zároveň představuje možnost seznámit se s definicí rozšíření pro VS Code a jeho publikováním. Pokud bude jako první vytvořena definice pro jazyk IFJcode20, tak je navíc možné ji zveřejnit pro vyzkoušení studenty předmětu IFJ. Syntaxe jazyka VYPcode je odlišná a definice pro něj bude vytvořena odděleně, ale stále může převzít strukturu rozšíření a strukturu definice gramatiky. Pro definice gramatik všech jazyků bude možné použít část již definovaných regulárních výrazů pro jednotlivé tokeny jazyků, v případě IPPcode a IFJcode ze syntaktického analyzátoru pro jazyk IPPcode20 vytvořeného autorem v rámci projektu do předmětu IPP a v případě VYPcode z definic vytvořených v rámci projektu interpretu pro VYPcode Ing. Kocmanem.

4.2 Interprety jazyků mezikódu s podporou pro ladění

Pro účely spouštění programů napsaných v jazycích mezikódu je nutné vytvořit interpret, který je možné distribuovat spolu s rozšířením (pro samotnou interpretaci je také možné umožnit v rámci rozšíření specifikovat cestu k referenčnímu interpretu). Interpret musí

kromě běžného načtení a provedení kódu umožňovat i kontrolované vykonávání instrukcí, které je řízeno adaptérem ladění. Jednoznačnou volbou je zde objektový přístup, který umožní adaptéru ladění spravovat objekt interpretu a používat ho k vykonání požadovaných operací podle zpráv protokolu DA. Interpret navíc musí být rozšířen o metody umožňující přístup k paměti, modifikování jeho vnitřního stavu i mimo jeho metodu pro interpretování zadaného vstupu atd. Dále je nutné, aby interpret umožňoval kontrolovaně provádět instrukce na základě řízení adaptérem.

Referenční interpret pro jazyk IFJcode je implementován v jazyce C++ a je dostupný jako spustitelný binární soubor. Referenční interpret pro jazyk IPPcode je neveřejný a je pouze modifikací existujícího interpretu pro IFJcode. Oproti tomu interpret jazyka IPPcode vytvářený v rámci projektu v předmětu IPP je implementován v jazyce Python. Implementace interpretu v rámci této práce bude používat jazyk Python stejně jako projekt do předmětu IPP. Může tak být převzata část implementace z autorova řešení tohoto projektu. Použití jazyka Python je také vhodné, jelikož je to na rozdíl od jazyka C++ jazyk interpretovaný, což umožňuje, aby byl výsledný interpret a ladicí nástroj spustitelný na téměř libovolné platformě s jedinou závislostí v podobě interpretu jazyka Python (konkrétně ve verzi 3). Vzhledem k tomu, že jazyk Python je zadaný implementační jazyk pro část projektu v předmětu IPP, která se zabývá vytvářením interpretu, tak je navíc téměř zaručené, že ho studenti budou mít dopředu k dispozici.

Volba jazyka Python pro implementaci interpretu jazyka IFJcode může vést k drobným rozdílům v chování vstupních a výstupních instrukcí, kvůli rozdílům popsaným v podkapitole 2.1 zabývající se jazykem IFJcode. Chování programu zapsaného v jazyce IFJcode je však identické s jazykem IPPcode a hlavní motivací pro vytvoření podpory pro ladění je možnost sledovat chování programu v IFJcode, což případnými drobnými rozdíly v hodnotách proměnných nebude ovlivněno. Pro kontrolu správnosti chování vstupních a výstupních instrukcí navíc studenti stále mohou použít zveřejněný referenční interpret IFJcode.

Rozšíření pro jazyk VYPcode se vytvoří rozšířením existujícího referenčního interpretu jazyka VYPcode, který je implementován v jazyce Java a distribuován v podobě přeloženého kódu ve spustitelném archivu formátu `jar`.

4.3 Ladicí nástroj komunikující protokolem Debug Adapter

Vytvořené interprety mají jiný implementační jazyk než jazyk TypeScript používaný pro vytváření rozšíření pro VS Code. Vytvářený ladicí nástroj (dále také nazývaný ladicí adaptér nebo jen adaptér) proto musí být externí a komunikovat s klientem pomocí jednoho z definovaných způsobů zasílání zpráv protokolu DA. Jako nejvhodnější se jeví použití lokální komunikace přes protokol TCP, což umožní adaptéru řídit komunikaci na základě zpráv přijatých přes TCP socket, a ponechá standardní vstup a výstup nevyužitý. Je tedy možné, aby laděný program kontrolovaný adaptérem standardní vstup a výstup používal bez nutnosti dalších zásahů do interpretu. Použití protokolu TCP navíc umožňuje sledování komunikace mezi klientem pomocí nástrojů jako je např. Wireshark, což zjednodušuje analýzu komunikace a případné ladění.

Všechny tři jazyky mezikódu obsahují instrukce pro načítání vstupu, které jsou základní a používanou složkou jazyka. Není proto možné spouštět laděný program mimo terminál a komunikovat pouze zasíláním výstupních událostí do ladicí konzole.

Vzhledem k tomu, že standardní vstup a výstup adaptéru bude používán laděným programem, tak bude svázán pouze s jediným laděným programem. Adaptér tedy funguje jako nadstavba nad interpretem pro účely ladění jednoho programu, je spuštěn před zahájením

ladění a je ukončen spolu s jeho skončením. Po spuštění adaptér počká na první spojení a dále již komunikuje jen s tímto klientem, od kterého očekává zprávy protokolu DA.

4.3.1 Řízení laděného programu

Adaptér ladění je implementován ve stejném jazyce jako interpret a obsahuje objekt interpretu, který slouží k vykonávání laděného programu. Adaptér proto může používat veškeré metody interpretu a případně přistupovat k jeho interním proměnným, čímž je umožněno načtení instrukcí laděného programu a konfigurace bodů přerušení. Adaptér ale musí umožňovat především kontrolované vykonávání řízeného programu. Pro kompilované jazyky je implementace těchto funkcí zajištěna operačním systémem, který umožňuje pomocí systémových volání ovlivňovat externí laděný proces. Jazyky mezikódu jsou však interpretované a pro účely konfigurace a dalšího vyhodnocování je interpret přímo obsažen v adaptéru ladění, existuje tedy pouze jeden proces vykonávající kód adaptéru.

Naivní řešení představuje blokující provádění instrukcí na základě přijatých požadavků protokolu DA. V tomto případě se po přijetí požadavku na pokračování provádění programu spustí odpovídající metoda interpretu a až po jejím provedení je řízení předáno adaptéru, který zpracuje další přijatou zprávu. Problémem tohoto přístupu je existence blokujících instrukcí, které slouží k načítání vstupu (instrukce `READ` pro IFJcode a IPPcode a varianty této instrukce pro různé datové typy pro VYPcode). Tyto instrukce blokují, dokud nenačtou data z nastaveného vstupu, což v případě výchozího standardního vstupu znamená čekání, dokud uživatel nezadá vstup do terminálu, ve kterém běží interpret (nebo v tomto případě adaptér). Podobný problém nastane, pokud je v programu obsažen nekonečný cyklus tvořený instrukcemi skoku a je proveden požadavek na pokračování vykonávání laděného programu. Požadavek na pozastavení vykonávání programu by mohl v tomto případě přerušit vykonávání a odhalit tyto cykly, což může být vhodné především v předmětu IFJ pro odhalení chybných cyklů ve vygenerovaném kódu. Při tomto naivním přístupu to však není možné, protože adaptér nikdy nezíská zpět kontrolu a neprovede další zaslané požadavky. V tomto případě také není možné korektně ukončit ladění a je nutné ukončit proces adaptéru zasláním signálu v terminálu.

Řešením může být rozšíření interpretu o možnosti ladění, jako je tomu například u virtuálního stroje pro jazyk Java, který umožňuje přistoupit k jinému procesu pomocí ladicího rozhraní (zde se nejedná přímo o interpretovaný jazyk, ale koncept je podobný). Bližším příkladem je jazyk Python, pro který existují různé implementace ladění s rozdílnými funkcemi (některé např. nepodporují požadavek na pozastavení), které obecně obsahují vlákno interpretující zdrojový kód a v dalším vlákně přijímají zprávy, které řídí vykonávání programu.

Přesunutí vykonávání instrukcí interpretem do jiného vlákna řeší problém blokujících instrukcí a nekonečných cyklů. Žádný existující interpret však momentálně neobsahuje podporu pro řízení jakýmikoli zprávami, prakticky by proto bylo nutné přidat do implementace interpretu vlákno, které by přijímalo zprávy a na základě nich vykonávalo program. Adaptér by poté obsahoval další vlákno, které by komunikovalo s řídícím vláknem tohoto interpretu. Interpret by tak bylo nutné rozšířit o komunikační rozhraní a problém by se zduplikoval, jelikož oba existující procesy adaptéru i interpretu by stále vyžadovaly řídicí vlákno komunikující přes protokol TCP (pravděpodobně s použitím protokolu DA i v interpretu, kterému by adaptér přeposílal dotazy) a vlákno zodpovědné za vykonávání programu (v případě adaptéru vlákno komunikující s procesem interpretu a čekající na odpověď).

Nebude proto dále rozšiřován interpret, ale adaptér bude obsahovat vlákno odpovědné za interpretaci kódu. Objekt interpretu bude sdílen mezi hlavním vláknem adaptéru přijímajícím požadavky přes TCP a vláknem odpovědným za interpretaci (dále jako ladicí vlákno). Toto řešení zachovává možnost konfigurovat interpret v kontextu hlavního vlákna adaptéru a ponechává mu možnost zasahovat do paměti interpretu a používat jeho metody k provádění vyhodnocovacích požadavků. Žádná z těchto operací není blokující a je tedy možné je provést po přijetí zprávy hlavním vláknem. Požadavky na vykonávání kódu vedou na spuštění interpretu v ladicím vlákně a blokující instrukce tak neovlivní fungování adaptéru.

4.3.2 Ladicí vlákno

Ladicí vlákno by mělo existovat od začátku ladění až do jeho ukončení. Funkce předaná vláknu k vykonání proto musí obsahovat cyklus, který v každém průchodu vykoná jedinou instrukci až do skončení laděného programu. Pozastavení vlákna je možné implementovat funkcí implementačního jazyka k čekání na nějaké podmínce. Pro pozastavení vlákna na základě požadavku hlavního vlákna je nutné, aby vlákno kooperovalo pomocí explicitní kontroly nastavené podmínky, na níž má čekat. Tato kontrola je vždy prováděna na začátku cyklu, stále je tedy možné, že dojde k zablokování vlákna na instrukci načítající vstup. Při tomto řešení to však není problém, jelikož hlavní vlákno pouze zadá vláknu případný požadavek na pokračování nebo pozastavení, na který ladicí vlákno zareaguje po dokončení blokující instrukce. Kvůli blokování na vstupních operacích je však nutné, aby vlákno běželo na pozadí (*daemon thread*) a došlo tak k jeho ukončení automaticky při skončení hlavního vlákna.

Veškeré vykonávání instrukcí je přesunuto do ladicího vlákna, které proto musí i odesílat události související s vykonáváním programu. Primárně se jedná o událost pozastavení (kvůli bodu přerušení, po dokončení kroku, na základě požadavku na pozastavení atd.). Dále musí ladicí vlákno posílat událost o skočení laděného programu a následně o ukončení ladění. Ladicí vlákno proto musí mít přístup k metodě nebo objektu hlavního vlákna sloužícího k odesílání DAP zpráv. Příslušná metoda musí být doplněna o synchronizaci, jelikož může být použita oběma vlákny, zde stačí použít jediný semafor pro omezení přístupu k socketu pro odesílání.

Vlákna nemohou na rozdíl od procesů vracet chybové kódy, ladicí vlákno by navíc nemělo nekontrolovaně skončit, proto je nutné, aby interpret chyby reprezentoval vyvoláním výjimky a ne pouhým ukončením programu s danou chybou. Ladicí vlákno může výjimky zachytávat, nastavit do proměnné odpovídající chybový kód a následně kontrolovaně skončit. Ukončení adaptéru proběhne po přijetí zprávy o odpojení klienta (požadavek `disconnect`), na který adaptér odpoví a následně se ukončí s nastaveným chybovým kódem, což opět odpovídá přístupu, že adaptér funguje pouze jako nadstavba nad interpretem a z hlediska vstupu a výstupu v terminálu chováním odpovídá obyčejnému interpretu.

4.4 Struktura rozšíření

Rozšíření jsou složena z definice gramatiky jazyka a kódu rozšíření v jazyce TypeScript sloužícího k propojení vytvořené podpory s editorem Visual Studio Code. Definice rozšíření musí obsahovat definice jazyka, přidané příkazy na spouštění interpretu a nový typ ladění. Kód rozšíření implementuje spouštění interpretu a externí implementace adaptéru ladění.

Ke spouštění interpretu i adaptéru je možné použít API VS Code pro práci s integrovanými terminály. Spouštění kódu lze doplnit jako příkaz editoru, který spustí interpretaci otevřeného souboru v integrovaném terminálu. Pro ladění rozšíření registruje továrnu pro ladicí adaptér typu server, která v aktivační metodě v integrovaném terminálu spustí adaptér a vrátí port, na kterém byl spuštěn. Tento adaptér následně obsluhuje jediného klienta, kterým je právě spuštěné ladění a skončí při jeho skončení. Integrovaný terminál je výchozí a jediný podporovaný způsob spuštění laděného programu. Podobně se chová ve výchozím nastavení ladění poskytované standardním rozšířením pro jazyk Python[1], které je obecně používáno jako vzor pro implementaci funkcí ladění.

[1] https://marketplace.visualstudio.com/items?itemName=ms-python.python

Kapitola 5

Implementace a testování

Tato kapitola se zabývá implementací rozšíření pro Visual Studio Code poskytujících podporu pro jazyky IPPcode, IFJcode a VYPcode. Zabývá se vytvářením vlastního rozšíření, které je přímo závislé na editoru VS Code, a ladicího nástroje pro zmíněné jazyky, který komunikuje protokolem Debug Adapter. Implementace je rozdělena do dvou hlavních částí. První se zabývá podporou pro IPPcode a z ní odvozenou podporou pro IFJcode. Druhá část implementace se zabývá podporou pro VYPcode. Implementace rozšíření pro VS Code je zde vytvořena úpravou již vytvořeného rozšíření pro IPPcode a IFJcode, implementace ladění používá interpret pro VYPcode vytvořený Ing. Kocmanem.

5.1 Definice jazyka IFJcode20 a jeho gramatiky pro syntaktické zvýrazňování

Definice jazyka a jeho gramatiky neobsahuje žádnou programovou část, doplní však syntaktické zvýrazňování, což je jedna ze tří komponent podpory pro jazyky mezikódu, které v rámci této práce mají být vytvořeny. Přidání syntaktického zvýrazňování samo o sobě přidává dostatečnou funkcionalitu, která může být smysluplná i jako jediný obsah zveřejněného rozšíření, a umožňuje otestovat proces vytváření a zveřejnění rozšíření.

Jako první byla vytvořena definice jazyka pro IFJcode ve verzi 20, která byla vydána jako samostatné rozšíření **IFJcode20 syntax**[1]. Tento jazyk byl zvolen, protože rozšíření mohlo být zveřejněno v průběhu zimního semestru, kdy probíhal předmět IFJ, a studenti tak měli možnost rozšíření vyzkoušet. Díky syntaktické podobnosti mezi jazyky mohly být pro definici gramatiky jazyka použity existující regulární výrazy z autorova projektu z předmětu IPP z roku 2020.

5.1.1 Definice rozšíření

Nový jazyk je definován pod klíčem `languages`, který obsahuje pole objektů jazykových konfigurací [17]. Jazyková konfigurace obsahuje identifikátor `id`, který je dále používán v definičním souboru rozšíření a v kódu rozšíření pro práci s definovaným jazykem, jako identifikátor je použito `ifjc20`. Položka `aliases` definuje pole názvů jazyka, přičemž první z nich je použit jako jméno jazyka v rámci editoru, je použit název `IFJcode20`. Položka `extensions` slouží k přiřazení identifikátoru jazyka k souborům podle jejich přípony. Definuje se pole přípon, konkrétně byly zvoleny hodnoty `.ifjc20`, `.ifjcode20`, `.ifjc`, `.ifjcode`. Definice

[1] https://marketplace.visualstudio.com/items?itemName=okrejci.ifjc20-syntax

obsahuje i obecné přípony bez čísla verze, protože se předpokládá, že rozšíření by mělo podporovat vždy poslední verzi jazyka a verze z předchozích let nemají dále význam.

Další možnost identifikace jazyka představuje položka `firstLine`, která obsahuje regulární výraz, který editor použije k pokusu o identifikaci typu souboru na základě porovnání s prvním řádkem souboru. Tento druh identifikace má však nižší prioritu než výběr na základě přípony souboru. Důsledkem tohoto chování je, že pokud je možné identifikovat jazyk na základě přípony, tak se kontrola prvního řádku neprovede. Vzhledem k výše popsanému předpokladu, že se jako výchozí jazyk bere vždy nejnovější verze, to však nutně nepředstavuje problém. Existuje návrh na doplnění větší míry kontroly nad výběrem identifikátoru jazyka pomocí konfigurace[2], zatím však v této oblasti nedošlo k žádnému posunu. Větší problém představuje fakt, že pro identifikaci souboru pomocí prvního řádku je první řádek striktně chápán jako první řádek souboru, i když je řádek prázdný, což je v rozporu se specifikací jazyka IPPcode21 a obecně s chováním interpretů pro IFJcode a IPPcode, které umožňují nejen prázdné řádky, ale i komentáře před samotným úvodním řádkem. Stále se ale nejedná o zásadní problém za předpokladu, že pro soubory s jazyky mezikódu bude používána jedna z definovaných přípon. Navíc je možné v nastavení VS Code doplnit asociaci mezi dalšími příponami a identifikátorem jazyka.

Poslední položkou definice jazyka v sekci `languages` je položka `configuration`, která obsahuje cestu ke konfiguračnímu souboru jazyka. Konfigurační soubor je také ve formátu JSON a umožňuje nastavit složky jazyka, pro které editor přidává další akce. Jedná se například o automatické odsazování, označování závorek, doplňování párových znaků a podobně. Vzhledem k tomu, že jazyky IPPcode a IFJcode mají velmi jednoduchý syntax, který neobsahuje žádné bloky, tak je v konfiguračním souboru nastavena pouze jedna hodnota, a to definice řádkových komentářů, které může editor použít k zakomentování části zdrojového kódu. Kompletní obsah konfiguračního souboru proto vypadá takto 5.1.

```
{
    "comments": {
        "lineComment": "#"
    }
}
```

Výpis 5.1: Obsah konfiguračního souboru jazyka `language-configuration.json` pro jazyky IFJcode a IPPcode.

5.1.2 Definice gramatiky jazyka

Gramatiky jazyků jsou definovány pod klíčem `grammars`. Definice gramatiky v položce `language` obsahuje identifikátor jazyka definovaný v sekci `languages`, v tomto případě `ifjc20`. Položka `scopeName` obsahuje kořenový identifikátor rámce pro definovanou gramatiku, který je přiřazen celému souboru. Jedná se o definice zdrojového kódu, proto je podle [7] zvolen identifikátor `source.ifjc20`. Poslední položka `path` obsahuje cestu k souboru se samotnou TextMate gramatikou. Pro definici gramatiky je opět zvolen soubor ve formátu JSON, který je umístěn standardně ve složce `syntaxes`.

Soubor s TextMate gramatikou obsahuje definice tokenů jazyka, kterým přiřazuje identifikátory. Na ukázce 5.2 je vidět zjednodušený příklad obsahu souboru definující pouze ně-

[2] https://github.com/microsoft/vscode/issues/10915

kolik operačních kódů a návěští. Položka `$schema` obsahuje definici schématu pro TextMate gramatiky ve formátu JSON sloužící k validaci. Položka `scopeName` opakuje hodnotu již zadefinovanou v definičním souboru rozšíření a položka `name` opakuje název jazyka, pro který je definována gramatika.

```
{
  "$schema": "https://raw.githubusercontent.com/martinring/tmlanguage/
  ↪  master/tmlanguage.json",
  "name": "IFJcode20",
  "scopeName": "source.ifjc20",
  "patterns": [
    {"include": "#keywords"},
    {"include": "#labels"}
  ],
  "repository": {
    "keywords": {
      "patterns": [{
        "name": "keyword.control.ifjc20",
        "match": "(?:^|^\\s*)(?i:MOVE|NOT|INT2CHAR|STRLEN|TYPE)\\b"
      }]
    },
    "labels": {
      "patterns": [{
        "name": "variable.other.label.ifjc20",
        "match": "(?<!^|^\\s*)(?<=\\s)([a-zA-Z]|_|-|\\$|&|%|\\*|!|\\?)
        ↪  ([a-zA-Z]|_|-|\\$|&|%|\\*|!|\\?|[0-9])*(?=\\s)"
      }]
    }
  }
}
```

Výpis 5.2: Zjednodušená ukázka souboru s TextMate gramatikou pro jazyk IFJcode20 (soubor `ifjc20.tmLanguage.json`).

Pro definici tokenů se používá pouze repozitář, tzn. všechny definice jsou obsaženy v položce `repository` a kořenová položka `patterns` obsahuje pouze odkazy na hodnoty v repozitáři. Tento přístup není nutný, jelikož jazyk IFJcode neobsahuje žádné bloky, a pravidla definovaná v repozitáři proto nejsou používána rekurzivně. Nabízí ale jednoduchý přehled všech definovaných kategorií tokenů a jejich pořadí. Také umožňuje přestat používat definice jen pomocí odstranění vložení definice pomocí položky `include` bez zásahu do samotných definic, což je vhodné pro testování.

Definice tokenů jsou napsané podle specifikace jazyka [12], díky identické syntaxi jazyků IFJcode a IPPcode mohly být použity regulární výrazy pro řetězce a názvy ze syntaktického analyzátoru pro IPPcode. Regulární výraz pro desetinná čísla v hexadecimálním formátu byl napsán podle syntaktického diagramu pro literál hexadecimálního desetinného čísla

dostupného na stránkách společnosti IBM[3], který bez přípony určující datový typ odpovídá požadovanému formátu.

S výjimkou komentáře jsou definice pomocí regulárních výrazů složeny ze dvou částí. První část je definice samotného tokenu, která definuje znaky, ze kterých se může daný token skládat. Druhá část je definice hranic tokenu, které definují jaké znaky token mohou obklopovat. Tokeny reprezentující jednotlivé části instrukcí, totiž od sebe musí být odděleny bílými znaky.

Pro definici konce tokenů existuje běžně používaná escape sekvence `\b` označující hranici slova (*word boundary*). Tato hranice slova ale předpokládá, že slovo je složené pouze z písmen a číslic. Jazyk IFJcode však obsahuje v názvech proměnných a návěští další znaky a navíc používá znak `@` jako oddělovač názvu rámce nebo typu proměnné od části s názvem/hodnotou. Kontrola pomocí hranice slova `\b` je proto použita pouze na detekci konce operačního kódu. Kontroly dalších hranic slov jsou řešeny pomocí dopředných a zpětných kontrol (*lookahead* a *lookbehind assertions*). Toto je možné vidět na příkladu definice názvu návěští (sekce `labels` v repozitáři). Konec slova je určen detekcí libovolného bílého znaku pomocí regulárního výrazu `(?=\s)` a jeho začátek je určen identicky pomocí `(?<=\s)`.

Vzhledem k tomu, že definice jazykových tokenů neobsahují žádnou další strukturu, tak by v zatím popsaném stavu nebyly schopny rozlišovat mezi názvy návěští a názvy operačních kódů, jelikož libovolný operační kód je zároveň i validním názvem návěští. Z tohoto důvodu je pro definici úvodní hranice operačního kódu použit regulární výraz `(?:^|^\s*)`, který vyžaduje, aby tokenu operačního kódu předcházel začátek řádku. Tokeny označující argumenty instrukcí před již popsanou definicí začátku tokenu obsahují navíc regulární výraz `(?<!^|^\s*)` kontrolující, že se token nenachází na začátku řádku. Tokeny argumentů tedy nesmí být na začátku řádku a jsou odděleny bílými znaky, čímž je do definic tokenů jazyka doplněna určitá kontrola syntaktické správnosti zapsaných instrukcí.

Definice gramatiky obsahuje oddělené definice pro proměnné v různých rámcích, sdílejí ale identický kořenový identifikátor, takže na jejich zobrazování toto nemá vliv. Jako poslední pravidlo gramatika obsahuje i definici hlavičky, které je přiřazen identifikátor `storage.type.ifjc20`, který ve většině stylových předpisů nemá přiřazen žádný styl (pravidla kontrolují pouze formát, takže řádek s hlavičkou identifikuje kdekoli ve zdrojovém kódu).

Pro identifikátory jazykových tokenů (položka `name`) byly použity existující identifikátory jako `variable` pro proměnné, `keyword.control` pro operační kódy a podobně pro konstanty. Hlavním cílem bylo, aby pro tokeny byl přiřazen styl v běžných stylových předpisech pro VS Code. Proto je například pro název návěští také použit kořenový identifikátor `variable`.

5.2 Implementace interpretu pro jazyk IPPcode

Pro vytvoření podpory pro spouštění a ladění programů v jazycích IPPcode a IFJcode je nutné vytvoření interpretu, který může být dále rozšiřován a modifikován pro potřeby různých verzí jazyka. Výsledný interpret zároveň musí být možné ovládat pro účely běžných ladicích funkcí, především jde o kontrolované interpretování kódu, tj. provádění kroků, pozastavování interpretace a s tím spojená možnost nastavování bodů přerušení a pozastavování na nich. Dále by měl být interpret a jeho syntaktický analyzátor snadno použitelný pro účely vyhodnocování proměnných a dalších instrukcí. Popisovaná implementace odpo-

[3] https://www.ibm.com/docs/en/zos/2.3.0?topic=literals-floating-point

vídá konkrétně interpretu pro jazyk IPPcode20, což je primární implementace, ze které se následně úpravami vytvoří podpora pro další verze obou jazyků.

5.2.1 Analýza a zpracování vstupního kódu

V prvním kroku je nutné zpracovat zdrojový kód a převést ho do formátu, který může být interpretován. Běžným přístupem je vytvoření lexikálního analyzátoru (angl. *lexer*), který vstupní text rozdělí na jednotlivé tokeny programovacího jazyka a kontroluje pouze, jestli vstup reprezentuje sekvenci validních tokenů zpracovávaného jazyka. Tokeny následně zpracovává syntaktický analyzátor, který kontroluje, jestli posloupnost tokenů odpovídá validním syntaktickým konstrukcím jazyka, a vytváří z nich reprezentaci, která může být vyhodnocována interpretem.

Jazyky IFJcode a IPPcode mají velmi jednoduchou syntaktickou strukturu, která je tvořena pouze hlavičkou a poté jedinou validní syntaktickou strukturou instrukce skládající se z operačního kódu následovaného argumenty, jejichž počet (nula až tři) a typ závisí na operačním kódu. Mají také dvě pravidla, která závisí na kontextu, konkrétně se jedná o úvodní hlavičku a operační kódy na počátku řádku. Pro analýzu vstupu proto byla vytvořena jediná třída syntaktického analyzátoru, která je zodpovědná jak za lexikální, tak syntaktickou analýzu.

Cílem syntaktického analyzátoru je zpracovat vstupní zdrojový kód a převést ho na instrukce. Instrukce musí být adresovatelné pomocí hodnoty `order` reprezentující jejich pořadí, která teoreticky může být libovolná, k reprezentaci instrukcí je proto použita mapa s číselnými klíči. Při načítání zdrojového kódu se hodnota `order` počítá od jedničky. Syntaktický analyzátor je také zodpovědný za zpracování vstupních hodnot a jejich převedení do interní reprezentace, kterou může interpret použít bez dalších kontrol nebo vyhodnocování. Syntaktický analyzátor také vyhodnocuje definice návěští a ukládá je do mapy, kde klíčem je název návěští a hodnotou je číslo `order` instrukce `LABEL`.

Případné chyby při analýze vedou k vyvolání výjimky třídy `ParserException`, která obsahuje kód chyby a chybovou zprávu. Toto chování umožňuje případné zachycení chyb a jejich zpracování při využívání funkcí syntaktického analyzátoru. Chybové kódy odpovídají chybovým kódům z definice jazyka IPPcode20 [13].

Syntaktický analyzátor stejně jako skript `parse.php` z projektu z předmětu IPP umožňuje načítání ze standardního vstupu, hlavní je ale načítání vstupu ze zadaného souboru, které je programově přístupné pro použití v interpretu a adaptéru ladění. Syntaktický analyzátor také umožňuje méně striktní chování, než jaké je ve specifikaci jazyků, kdy nevyžaduje hlavičku na začátku souboru se zdrojovým kódem.

Definice instrukcí

Pro zpracování instrukcí obsahuje syntaktický analyzátor mapu všech známých operačních kódů a jejich atributy. Jak je vidět na příkladu 5.3, každý operační kód má přiřazené pole typů atributů, které určuje jejich počet, pořadí a typy. Názvy atributů odpovídají definici instrukcí ve specifikaci jazyka. Možné hodnoty jsou `var` odpovídající tokenu identifikátoru proměnné, `symb` odpovídající konstantám a proměnným, `label` značící návěští a `type` značící název typu (s výjimkou typu `nil`).

```
'ADD': ['var', 'symb', 'symb']
```

Výpis 5.3: Definice instrukce `ADD` v syntaktickém analyzátoru.

Reprezentace načtených dat

Pro reprezentaci instrukcí existuje třída `Instruction`, která obsahuje operační kód a pole argumentů. Také obsahuje položku `next_order`, která se odkazuje na následující instrukci, a číslo řádku, ze kterého byla instrukce načtena. Argumenty jsou reprezentované třídou `Argument`, která obsahuje hodnotu a typ obsažené hodnoty. Typ může obsahovat všechny datové typy a typy argumentů kromě `symb`, který je nahrazen konkrétním datovým typem nebo typem proměnné `var`. Třída `Argument` je použita pro všechny argumenty kromě proměnných. Jména proměnných jsou reprezentována zděděnou třídou `VariableRef`, která navíc obsahuje oddělený název rámce a proměnné (jako hodnota je uloženo celé jméno ve zdrojovém formátu).

Zpracování vstupu

Syntaktický analyzátor načítá vstup po řádcích a odstraňuje bílé znaky a komentáře, znak mřížky `#` uvozující komentář se v definici jazyka jinde nevyskytuje a nemůže být obsažen v řetězcovém literálu, proto stačí pouze zahodit část řádku od mřížky dále. Pokud takto očištěné řádky zůstanou prázdné, tak se přeskakují. Pro následnou analýzu neprázdných řádků existují dva stavy. První je hledání hlavičky, které se provádí na začátku analýzy. Pokud první řádek neodpovídá hlavičce, která je vyžadována, tak se vyvolá výjimka. Pokud byla nalezena hlavička, pokračuje se na další řádek vstupu.

Následuje hlavní stav analýzy, kdy se zpracovávají instrukce. Načtený řádek je rozdělen na části oddělené bílými znaky. Zkontroluje se, jestli první část odpovídá známému operačnímu kódu, a podle jeho očekávaných typů a počtu argumentů se zpracují zbývající části.

Pro zpracování jednotlivých typů argumentů existují funkce vracející objekt `Argument`. Samotné zpracování očištěného vstupního řádku je také implementováno jako samostatná funkce vracející objekt `Instruction`. Cílem je, aby funkce mohly být samostatně znovupoužitelné pro vyhodnocování výrazů.

Doplnění další logiky pro zpracování určitých instrukcí je možné přiřazením funkce k určitému operačnímu kódu, tato funkce se při zpracovávání instrukce provede po vytvoření objektu `Instruction` a vyhodnocení všech argumentů. Tímto způsobem je řešeno definování návěští.

5.2.2 Zpracování XML reprezentace jazyka IPPcode

V projektu v předmětu IPP je výstupem skriptu `parse.php` XML reprezentace načtených instrukcí s nevyhodnocenými argumenty. Interpret je poté zodpovědný za zpracování této reprezentace, další kontrolu a převedení do interní reprezentace. Interpret vytvořený v rámci této práce by ale měl pouze načíst již zpracovaný vstup ze syntaktického analyzátoru a interpretovat podporované instrukce. Možnost interpretace vstupu v XML formátu pro IPPcode by ale měla zůstat zachována, proto je nutné rozšířit syntaktický analyzátor o práci s XML reprezentací.

Zpracování vstupu s XML reprezentací je doplněno jako samostatná metoda, která pomocí modulu `ElementTree` načte vstupní XML. Kořenový element XML musí obsahovat verzi IPPcode, tato konstanta je doplněna do třídy syntaktického analyzátoru. Jednotlivé argumenty se poté zpracovávají již existujícími metodami syntaktického analyzátoru a vytváří se objekty `Instruction`. Rozdílem je, že instrukce nemají přiřazené číslo řádku (obsahují hodnotu `None`) a hodnota `order` odpovídá hodnotě definované ve vstupním XML. Chyby při analýze XML také vedou na vyvolání výjimky `ParserException`, ale obsahují chybové kódy definované v zadání projektu z předmětu IPP v sekci o zpracování XML vstupu pro interpret.

Také je doplněna funkce k výpisu XML reprezentace zdrojového kódu na standardní výstup, čímž je zajištěna kompatibilita se skriptem `parse.php`. Funkce využívá vytvořenou mapu zpracovaných instrukcí a převádí je na XML elementy, které nakonec vytiskne na standardní výstup. Výstup skriptu `parse.php` ve formátu XML ale obsahuje nezpracované hodnoty konstant, proto je nutné, aby vytvořené objekty `Argument` obsahovaly i původní tvar obsažené hodnoty.

5.2.3 Interpret

Hlavní třída sloužící k vykonávání programu v jazyce IPPcode nebo IFJcode je třída interpretu. Tato třída je odpovědná za načtení zdrojového kódu a jeho provádění. Obsahuje paměť programu, jak byla popsána v podkapitole o paměťovém modelu jazyka IFJcode 2.1.3, a implementuje chování instrukcí. Třída obsahuje objekt syntaktického analyzátoru, který používá ke zpracování vstupu. Interpret zadaný vstup pomocí syntaktického analyzátoru zpracuje a získá z něj instrukce s vyhodnocenými argumenty a definovaná návěští.

Chování interpretu vychází ze zadání projektu z předmětu IPP, jsou proto podporovány stejné argumenty. Argument `--source` definuje soubor se zdrojovým kódem a argument `--input` obsahuje soubor se vstupem pro prováděný program. Musí být nastaven alespoň jeden ze vstupních souborů a místo druhého je případně použit standardní vstup. Ve výchozím stavu je nastaven vstup ze standardního vstupu a očekává se, že bude zadán soubor se zdrojovým kódem.

Chyby při interpretaci vyvolávají výjimku třídy `InterpreterException`, která identicky s chybami syntaktického analyzátoru obsahuje chybové kódy odpovídající zadání projektu z předmětu IPP. Chyby při načítání instrukcí pomocí syntaktického analyzátoru vedou k vyvolání odpovídajících výjimek syntaktického analyzátoru.

Provádění instrukcí

Řízení prováděného programu je dané interní proměnnou s hodnotou `order` další instrukce k provedení. Tato hodnota je nastavena při načtení instrukcí ze syntaktického analyzátoru. Samotné vykonávání instrukcí je řízeno metodou `run`, která na základě hodnoty `order` vybírá instrukci k vyhodnocení. Objekt instrukce je poté vyhodnocen metodou `execute_instruction`, která nastaví interní hodnotu `order` na hodnotu `next_order` prováděné instrukce a na základě operačního kódu provede odpovídající chování instrukce. Metoda `run` provádí instrukce dokud hodnota `order` není `None`, což značí vykonání poslední instrukce a skončení prováděného programu. Pro účely ladění je možné pomocí metody `run` provést vždy jen jednu instrukci, což slouží ke kontrolovanému provádění laděného programu. Také je možné, aby metoda `execute_instruction` neprováděla úpravu hodnoty `order`, což slouží k provádění dalších instrukcí nepocházejících z načteného programu.

Každá podporovaná instrukce je implementována jako samostatná metoda, aby bylo možné chování redefinovat pro jednotlivé verze jazyků. Instrukce k načtení vstupu `READ` používá metody syntaktického analyzátoru ke zpracování předaných hodnot. Interpret navíc obsahuje metodu, která převádí konstanty do řetězcové reprezentace pro účely zobrazování hodnot v paměti. Tato metoda je využita v implementaci instrukce tisku `WRITE`, ale není zcela ekvivalentní. Například hodnota `nil` se tiskne jako prázdný řetězec ale pro její textovou reprezentaci se používá řetězec `"nil"`.

Rozšíření o interpretaci XML vstupu

Výchozí chování interpretu načítá instrukce pomocí syntaktického analyzátoru jen ve zdrojovém formátu, což je vhodné pro jazyky IFJcode i IPPcode. Pro jazyk IPPcode je však třeba doplnit i načtení instrukcí v XML formátu, které bylo doplněno do syntaktického analyzátoru, a simulovat tak chování skriptu `interpret.py` z projektu z předmětu IPP. Pro tento účel byl přidán další argument `--xml`, který nastaví používání syntaktického analyzátoru k načítání vstupu v XML místo zdrojového formátu.

5.2.4 Výsledná struktura vytvořených tříd

Popsaná implementace obecně popisovala části syntaktického analyzátoru a interpretu, které byly vytvořeny pro účel interpretování zdrojového kódu jazyka IPPcode20. Tato implementace slouží jako základ pro vytvoření implementace všech potřebných verzí jazyků IPPcode a IFJcode. Bylo by sice možné, aby všechny další třídy dědily z implementace pro jazyk IPPcode20, takový přístup by ale mohl být matoucí a zbytečně by přidával do interpretu pro jazyk IFJcode práci s XML, která pro něj není definovaná.

Byla proto vytvořena obecná třída `IPPcIFJcParser`, která obsahuje metody ke zpracování zdrojového kódu v jazyce IFJcode nebo IPPcode, tato třída ale nemá definovaný konkrétní formát hlavičky souboru se zdrojovým kódem. Z této třídy dědí třída `IPPcParserXML`, což je implementace syntaktického analyzátoru pro IPPcode doplňující zpracování vstupu v XML formátu.

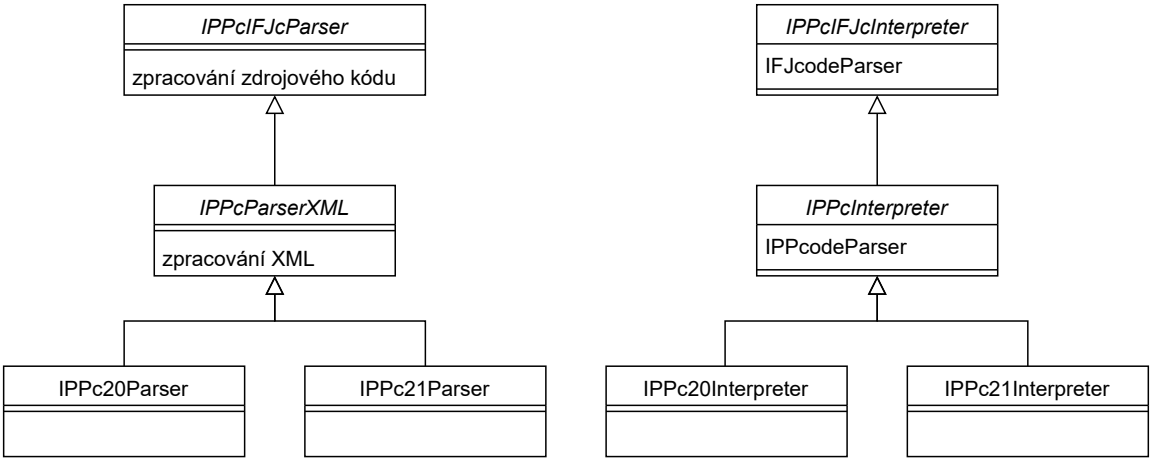
Konkrétní implementace syntaktického analyzátoru pro jazyk IPPcode následně dědí ze třídy `IPPcParserXML` a definují kontrolu hlavičky a název jazyka. Případně mohou předefinovat chování syntaktického analyzátoru, což ale není pro současné verze nutné. První konkrétní implementací je verze pro jazyk IPPcode20, na níž byla práce založena. Nicméně nejaktuálnější verzí v době odevzdání práce je již jazyk IPPcode21. Tyto verze jazyka se od sebe liší jen formátem hlavičky a názvem jazyka.

Existují tedy dva možné přístupy k implementaci syntaktického analyzátoru pro tyto jazyky. Prvním je vytvoření jediného konkrétního syntaktického analyzátoru, který podporuje jazyky IPPcode20 i IPPcode21. Toto řešení bylo obsaženo v první zveřejněné verzi rozšíření pro jazyk IPPcode a IFJcode. Řešení je sice plně funkční a umožňovalo by podporovat libovolné množství jazyků, které se liší jen kontrolami těchto názvů, ale pro uživatele může být matoucí používání jednoho nástroje pro IPPcode20 podporujícího i pozdější verze jazyka. Proto byl následně zvolen druhý přístup, vytvoření syntaktického analyzátoru pro každou verzi jazyka zvlášť, i když obsahují minimum rozdílů. Do budoucna se stejně očekává, že rozšíření bude podporovat pouze nejaktuálnější verzi jazyka a bude tedy existovat jen jedna konkrétní implementace syntaktického analyzátoru pro IPPcode.

Struktura tříd interpretu je podobná, základem je třída `IPPcIFJcInterpreter`, která obsahuje veškeré popsané chování interpretu s výjimkou načítání XML vstupu. Tato třída navíc obsahuje metodu odpovědnou za instanciaci syntaktického analyzátoru, aby bylo

možné pro konkrétní verze interpretu jednoduše nastavit správný syntaktický analyzátor. Implementací interpretu pro jazyk IPPcode je zděděná třída `IPPcInterpreter`, která doplňuje popsanou práci s XML. Konkrétní třídy interpretu pro IPPcode pouze definují správnou verzi syntaktického analyzátoru, mohou ale případně předefinovat další metody interpretu.

Dědičnost popsaných tříd je zobrazena na diagramu 5.1. Další konkrétní implementace tříd pro jazyk IPPcode by byly vytvořeny identicky. Třídy pro jazyk IFJcode mohou být vytvořeny přímo ze tříd `IPPcIFJcParser` a `IPPcIFJcInterpreter`.



Obrázek 5.1: Ukázka dědičnosti tříd syntaktického analyzátoru a interpretu s konkrétními implementacemi pro jazyky IPPcode20 a IPPcode21.

5.2.5 Doplnění podpory pro jazyk IFJcode

Jak již bylo popsáno, největší rozdíl mezi jazyky IPPcode a IFJcode je v chování vstupních a výstupních instrukcí. Instrukce `READ` v obou případech načte řádek ze vstupu. Pro číselné hodnoty je obsah načteného řádku v jazyce IPPcode přímo zkonvertován na odpovídající datový typ. Referenční interpret jazyka IFJcode požadovanou hodnotu získá pomocí funkce `sscanf` s odpovídajícím formátovacím řetězcem.

Načítání čísel pomocí formátovacího řetězce je simulováno pomocí odstranění bílých znaků z načteného řetězce a poté nalezení hodnoty pomocí regulárního výrazu, který reprezentuje podporované formáty odpovídajícího formátovacího řetězce.

Tisk hodnot v interpretu IPPcode odpovídá interpretu pro jazyk IFJcode až na datový typ `float`, jelikož funkce jazyka Python převádějící desetinné číslo na řetězec s jeho hexadecimální reprezentací ponechává v desetinné části nadbytečné nuly, proto je nutné jeho formát dále upravit pro zajištění kompatibility.

Dále bylo upraveno zpracování argumentů interpretu pro větší kompatibilitu s referenčním interpretem, s výjimkou argumentu pro tisk informací pro ladění, který není podporován. Také se nemodifikuje chování instrukcí pro ladicí výpisy. Navíc byl ponechán argument pro zadání vstupního souboru, aby jej bylo možné použít v konfiguraci ladění.

Chování referenčního interpretu jazyka IFJcode se dále liší ve zpracování řetězců, které jsou v jazyce C++ reprezentovány jako sekvence 8bitových hodnot zatímco v jazyce Python se jedná o sekvenci znaků s daným kódováním, které mohou mít interně různou délku. Práce

s řetězci obsahujícími `Unicode` symboly ale není v předmětu IFJ testována, tudíž je možné tento rozdíl zanedbat.

Pro účely interpretace je ve výchozím stavu používán vytvořený interpret pro jazyk IFJcode, do nastavení rozšíření je však přidána možnost zadání cesty k referenčnímu interpretu, který je používán, pokud je zadán. Referenční interpret je studentům vždy k dispozici a mohou ho tak jednoduše použít pro srovnání a ověření skutečného chování programu.

5.2.6 Problém zveřejnění zdrojového kódu

Zvolený implementační jazyk Python je interpretovaný, vytvořené programy jsou proto běžně distribuovány v podobě souborů se zdrojovým kódem, které poté mohou být spouštěny pomocí interpretu. Ve vytvořeném zdrojovém kódu je obsažena implementace interpretu jazyka IPPcode, což je jedna z částí projektu do předmětu IPP. Proto byla zkoumána možnost skrytí zdrojových kódů v jazyce Python.

Interpret jazyka Python při interpretaci zdrojového kódu vytváří přeložený bajtkód v podobě `pyc` souborů, které následně interpretuje. Naivním řešením skrytí kódu může být distribuce pouze těchto přeložených souborů. Přeložené soubory jsou však navázány na konkrétní verzi interpretu jazyka Python, kterým byly vytvořeny. Na rozdíl od přeloženého bajtkódu jazyka Java navíc mohou být `pyc` soubory jednoduše dekompilovány zpět do poměrně srozumitelného zdrojového formátu.

Další možností je použití instalátorů, jako je např. `PyInstaller`, které umožňují z projektu v jazyce Python vytvořit spustitelný soubor. Spustitelný soubor poté skrývá zdrojový kód v jazyce Python, stále však obsahuje přeložené `pyc` soubory. Vytvořený spustitelný soubor je navíc platformě závislý.

Jiným přístupem je vytvoření binárního spustitelného souboru ze zdrojového kódu v jazyce Python, čímž se odstraní nutnost distribuovat `pyc` soubory. Existují projekty překladačů jazyka Python do kompilovaných jazyků, jako je např. `Nuitka`, což je překladač do jazyka C.

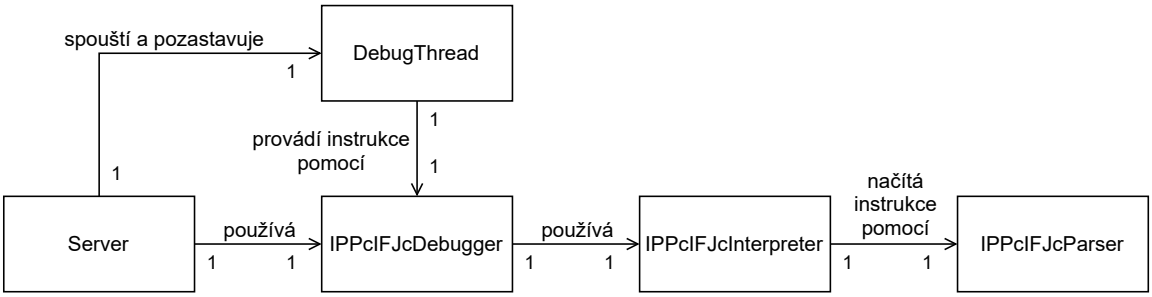
Podobným příkladem je `Cython`, což je jazyk definovaný jako nadmnožina jazyka Python, pro který existuje překladač do jazyka C. Kód v jazyce Python je až na drobné rozdíly (např. obsah proměnné `__class__` ve statických metodách) kompatibilní s jazykem `Cython`. Použití překladače `Cython` je proto doporučeno jako možnost skrytí části implementace při distribuci spustitelného souboru pomocí `PyInstaller`.

Poslední možnost představuje použití různých obfuskátorů, které mají za cíl modifikovat zdrojový kód tak, aby bylo ztížené jeho pochopení, ale aby stále vykonával identickou činnost. Existují moduly v jazyce Python (`IntensioObfuscator` nebo `pyminifier`), které umožňují obfuskovat kód v jazyce Python. Ani jeden však není aktivně vyvíjen a mají poměrně rozsáhlé limitace pro obfuskovaný kód, jako je nemožnost opakování stejných jmen identifikátorů a nutnost deklarace všech jmen ze standardních modulů, které nemají být měněny. Stále vyvíjený obfuskátor je komerční projekt `pyarmor`, který generuje obfuskovaný kód, k jehož interpretaci je následně vyžadován modul `pyarmor`.

Žádné skrytí zdrojových kódů však nakonec provedeno nebylo, jelikož by vyžadovalo další zásahy do projektu a přidávalo by nutnost řešit překlad vytvořeného kódu a jeho distribuci pro různé platformy. Vytvořené programy jsou tedy distribuovány ve zdrojovém formátu pouze s odstraněním komentářů a typových nápověd v relevantních třídách.

5.3 Implementace ladicího nástroje komunikujícího přes protokol Debug Adapter pro jazyky IPPcode a IFJcode

Adaptér funguje jako server přijímající DAP požadavky. Podporuje parametry pro nastavení portu, na kterém má být spuštěn TCP server a verzi jazyka IFJcode/IPPcode, se kterou má pracovat. Obsahuje interpret a vlákno odpovědné za vykonávání laděného programu. Pro interakci s interpretem pro účely vyhodnocování DAP požadavků byla vytvořena pomocná třída `IPPcIFJcDebugger`, která obaluje interpret jazyka IPPcode nebo IFJcode (instanci třídy `IPPcIFJcInterpreter`, viz obrázek 5.1). Tato třída doplňuje metody pro načítání a modifikování hodnot v paměti a další interakce s interpretem a jeho syntaktickým analyzátorem pro účely vykonání DAP požadavků. Jedná se v podstatě o další rozšíření interpretu, které doplňuje metody pro vyhodnocení požadavků protokolu Debug Adapter. Ukázka tříd obsažených v adaptéru je zobrazena na obrázku 5.2. Implementované metody v pomocné třídě neberou ohled na obsažený interpret, jelikož paměťový model interpretu a potřebné metody jsou ve všech verzích stejné. Dále se interpretem myslí jak tato třída, tak samotný v ní obsažený interpret.



Obrázek 5.2: Struktura tříd ladicího adaptéru pro jazyky IPPcode a IFJcode. Adaptér (třída `Server`) obsahuje objekt ladicího vlákna, které spolu s adaptérem má odkaz na pomocnou třídu `IPPcIFJcDebugger` pro přístup k interpretu a jeho řízení.

Ladicí vlákno je implementované jako podtřída vlákna, která obsahuje instanci rozšířeného interpretu a metodu k odesílání DAP zpráv. Pro účely pozastavování vlákno obsahuje synchronizační objekt `Event`, který umožňuje čekání, dokud není nastavena jeho hodnota. Vlákno obsahuje metody sloužící k nastavování hodnoty synchronizačního objektu a nastavování případného důvodu čekání. Pokud dojde k pozastavení vlákna a vlákno má nastaven důvod čekání, tak před zahájením čekání odešle klientovi událost zastavení.

Hlavní funkce vlákna na začátku zkontroluje, jestli má dojít k čekání před zahájením interpretace. Pokud ano, tak odešle odpovídající událost pozastavení a zahájí čekání. Následuje hlavní cyklus vykonávající celý program. Na začátku cyklu se vždy provádí kontrola podmínky a případné čekání, následně se provádí jedna instrukce. Po provedení instrukce se kontroluje existence bodu přerušení na následující instrukci a vlákno se případně samo pozastaví. Pokud je vláknu ukončeno čekání, ale je nastaven důvod přerušení na provedení kroku, tak si vlákno samo nastaví nové pozastavení. Po skončení cyklu (provedením poslední instrukce nebo vyvoláním výjimky) vlákno nastaví informační proměnnou o skončení interpretace, chybový kód, odešle událost skončení laděného programu a skončení ladění, čímž činnost vlákna skončí.

5.3.1 Přijímání a zpracování zpráv

Pro jazyk Python neexistuje žádná implementace protokolu Debug Adapter, která by umožňovala vytvořit generický ladicí adaptér v jazyce Python. Ladicí nástroj obsažený v rámci oficiálního rozšíření pro jazyk Python obsahuje svoji vlastní interní implementaci protokolu Debug Adapter. Existují implementace protokolu Language Server a samostatného protokolu JSON-RPC. Bylo by teoreticky možné použít některou z existujících implementací protokolu JSON-RPC a modifikovat ji pro zpracování protokolu Debug Adapter, toto řešení by však přidalo do projektu závislost na zvoleném modulu implementujícím protokol JSON-RPC, do kterého by následně musely být provedeny zásahy modifikující obsah zpracovávaných zpráv. Jazyk Python obsahuje modul `json`, který dokáže převádět řetězce obsahující data ve formátu JSON přímo do objektů jazyka Python (struktura map, polí a primitivních typů). Bez ohledu na implementaci posílání zpráv protokolu bude nutné implementovat strukturu a zpracování argumentů protokolu Debug Adapter. Byla proto zvolena možnost implementovat celou komunikaci přes protokol Debug Adapter bez využití externích modulů.

Po spuštění adaptéru je vytvořen TCP socket, který čeká na připojení prvního klienta. Po navázání spojení se původní socket uzavře a ponechá se pouze socket pro komunikaci s klientem, přes který se v nekonečné smyčce přijímají data. Přijatá data v kódování `UTF-8` se ukládají do interního pole bajtů. Po každém přijetí dat se adaptér pokusí z přijatých dat získat DAP zprávu. Vyhledá se oddělovač hlavičkové sekce od těla zprávy a data se rozdělí na dvě části. Hlavičková část dle specifikace obsahuje jedinou hlavičku definující délku zprávy v bajtech. Načte se tato hodnota, a pokud druhá část dat obsahuje zadané množství bajtů, tak se z ní odeberou. Jinak zpracování končí a čeká se na další data. Pokud byla z dat odebrána část s obsahem zprávy, tak se překonvertuje na řetězec, který je následně pomocí modulu `json` převeden na interní JSON reprezentaci a dále se zpracovává jako DAP požadavek. Pokud jakákoli část analýzy selže, tak se zahazuje zpracovávaná část přijatých dat a pokračuje se v čekání na další data.

Prvním krokem zpracování přijatých JSON dat je kontrola povinných argumentů pro DAP požadavek. Zprávy které neodpovídají požadavku jsou ignorovány. Pokud je zpráva požadavek, tak se na základě jejího typu (položka `command`) zavolá metoda adaptéru implementující daný požadavek. Pokud přijatý požadavek není podporován, tak se odešle chybová odpověď.

5.3.2 Reprezentace, zpracování a odesílání zpráv

Kromě základních kontrol správnosti požadavku se s přijatou zprávou neprovádějí další operace a požadavek je odpovídající metodě předán jako JSON objekt. Metody implementující požadavky jsou odpovědné za kontrolu vyžadovaných atributů přijatého požadavku a případné odeslání chybové odpovědi.

Jsou definovány třídy reprezentující DAP odpověď a událost. Modul `json` umožňuje konvertovat interní reprezentaci JSON formátu na řetězec. Obě třídy DAP zpráv proto fungují pouze jako obal nad mapou představující JSON objekt. Pomocí konstruktoru a metod objektů je ale zjednodušeno vytváření validních DAP zpráv. Navíc je doplněna metoda, která na základě DAP požadavku vytvoří základ odpovídající odpovědi (vyplní typ a sekvenční číslo požadavku, na který je odpovídáno).

Pro odeslání DAP zprávy má adaptér metodu `dap_send`, která odešle zprávu přes socket pro komunikaci s klientem. Metoda z objektu DAP zprávy vyjme vnitřní JSON objekt. Do objektu zprávy se doplní sekvenční číslo a inkrementuje se interní čítač sekvenčních čísel.

Následně se zpráva překonvertuje na řetězec a zakóduje do posloupnosti bajtů s kódováním `UTF-8`. Poté se spočítá délka těla zprávy, doplní se hlavička a celá zpráva se odešle klientovi. Vytvoření a odeslání odpovědi je chráněno semaforem.

5.3.3 Provádění požadavků

Po zpracování přijaté zprávy se volají metody implementující chování požadavků, které byly popsané v podkapitole 3.3.3 o protokolu DA. Každá metoda je odpovědná za zpracování požadavku a odeslání odpovědi. Pokud nastane při vykonávání metody výjimka, odesílá se výchozí chybová odpověď.

Adaptér po odeslání odpovědi na inicializační požadavek odesílá okamžitě událost ukončení inicializace. Následně očekává přijetí požadavku na spuštění laděného programu a teprve poté požadavky na nastavení bodů přerušení, což odpovídá chování klienta VS Code. Požadavek na spuštění programu načte instrukce ze zadaného souboru. Pokud soubor neexistuje nebo nastane chyba syntaktického analyzátoru, tak je vrácena chyba, na kterou klient reaguje ukončením ladění. Pokud požadavek obsahuje požadavek na zastavení programu po zahájení ladění, tak je tato vlastnost nastavena ladicímu vláknu. Metoda implementující požadavek na spuštění programu nespustí ladicí vlákno, jelikož požadavek je následován konfiguračními požadavky. Jediné podporované konfigurační požadavky jsou požadavky na nastavení obyčejných bodů přerušení. Pokud se nastavují body přerušení pro soubor s laděným programem, tak se kontroluje, jestli existuje instrukce s řádkem odpovídajícím bodu přerušení. Pokud existuje, tak se pro instrukci nastaví bod přerušení a v odpovědi se bod přerušení nastaví jako potvrzený, jinak se nastaví jako nepotvrzený. Body přerušení ze všech ostatních souborů se potvrdí. Je podporován požadavek na ukončení konfigurace, který zajistí spuštění ladicího vlákna.

Programy v jazycích IFJcode a IPPcode neobsahují žádné funkce, do zásobníku volání je proto vždy vrácen jediný záznam reprezentující instrukci, která má být po skončení pozastavení provedena jako další a její odpovídající číslo řádku. Rámce proměnných odpovídají specifikaci paměti těchto jazyků mezikódu. Vždy je vrácen globální rámec, zásobník a zásobník rámců. Lokální a dočasný rámec je vrácen jen, pokud existuje. Identifikátory rámců a zásobníků jsou konstanty, rámce na zásobníku jsou reprezentovány jako složené proměnné a jejich identifikátory odpovídají jejich indexu na zásobníku přičteného k definované konstantě. Hodnoty proměnných je možné získat přímo z interpretu. Proměnné i hodnoty na zásobníku mají svůj typ, stačí tedy vyhodnotit řetězcovou reprezentaci jejich hodnoty a odeslat ji.

Pro požadavek na sledování hodnot proměnných se používá zpracování identifikátoru proměnné pomocí syntaktického analyzátoru a poté přístup k odpovídající proměnné v paměti interpretu. Je tedy možné sledovat hodnoty proměnných ve třech existujících rámcích.

Identifikátor proměnné je také možné zadat do ladicí konzole jako jediný vstup, což vede na identické vyhodnocení. Jinak je vstup zadaný do konzole zpracován syntaktickým analyzátorem jako instrukce, které jsou následně vykonány interpretem. V konzoli nejsou povolené instrukce definující návěští, ukončující program a načítající vstup. Také se ignoruje hodnota `order` načtených instrukcí a při vyhodnocování se nepřenastavuje hodnota `order` interpretu na hodnotu pro následující instrukci. Po dobu vyhodnocování instrukcí z konzole jsou výstupy interpretu přesměrovány do vyrovnávacích pamětí a následně odeslány do konzole. Instrukce jsou ale přímo prováděné interpretem a mohou běžným způsobem měnit jeho interní stav, např. instrukce skoku tak stále mění hodnotu `order`. Při vyhodnocování v konzoli jsou zachytávané veškeré chyby interpretu, které jsou případně odeslané

jako obsah chybové odpovědi. Vyhodnocování tedy nemůže ukončit ladění, ale může měnit stav interpretu, což může způsobit chybu a ukončení při pokračování vykonávání laděného programu.

Řídicí požadavky používají metody na pozastavení a spuštění ladicího vlákna, při nastavení pozastavení předávají důvod pozastavení, který poté vlákno odešle před tím, než začne čekat.

Adaptér podporuje nastavování hodnot proměnných, vyhodnocení proměnné probíhá identicky jako u dotazu na získání hodnoty, navíc se ale pomocí syntaktického analyzátoru zpracuje předaná hodnota jako literál. Pokud je argument správný, tak se nastaví do proměnné a odešle se nová hodnota. Podpora nastavování hodnot proměnných umožní editovat všechny proměnné v pohledu proměnných, včetně složených proměnných, rozhraní DAP proměnné neumožňuje nastavit proměnnou jako needitovatelnou. Pokus o editování rámce na zásobníku proměnných proto automaticky vrací chybovou odpověď. Lokální rámec je podle specifikace jazyka vždy rámec na vrcholu zásobníku rámců, jeho obsah je proto v pohledu proměnných zobrazen dvakrát. Klient po provedení požadavku na změnu hodnoty proměnné neprovádí automaticky aktualizaci všech hodnot. Pokud je tedy modifikována hodnota v jednom z těchto rámců, odesílá se událost o znehodnocení načtených hodnot proměnných (`invalidated`), která zajistí aktualizaci hodnoty na druhém místě.

5.4 Rozšíření pro jazyky IPPcode a IFJcode

Jak již bylo popsáno v podkapitole 3.3 o rozšiřitelnosti editoru VS Code, základem rozšíření je definiční soubor rozšíření, který obsahuje informace o rozšíření a deklaruje funkce, které rozšíření doplňuje. Dále rozšíření obsahuje definici gramatik všech podporovaných verzí jazyků mezikódu. V rozšíření jsou obsaženy i zdrojové kódy v jazyce Python obsahující implementaci syntaktického analyzátoru, interpretu a ladicího adaptéru. Programová část rozšíření registruje příkazy pro interpretování otevřených souborů a implementuje spouštění adaptéru ladění.

V definičním souboru rozšíření je v položce `configuration` definováno nastavení pro rozšíření. Nastavení definuje své jméno (`IPPcode/IFJcode`) a poté jednotlivé položky nastavení. Nastavení je pod svým jménem dostupné v grafickém rozhraní nastavení VS Code. Každá položka obsahuje svůj typ a identifikátor, který je používán pro nastavení hodnoty v JSON souboru s nastavením. Pro účely zobrazení v GUI je tento identifikátor automaticky transformován editorem (např. identifikátor `IPPcode/IFJcode.pythonPath` na `IPPcode/IFJcode Python Path`). Je možné nastavit cestu k interpretu jazyka Python 3, který se má používat pro spouštění interpretu a adaptéru. Dále je možné nastavit výchozí port pro spouštění adaptéru a výchozí verzi jazyka IFJcode/IPPcode k ladění. Pro jazyk IFJcode je navíc možné specifikovat cestu k referenčnímu interpretu.

5.4.1 Definice jazyků a gramatik

Definice jazyků vychází z původního rozšíření definujícího jazyk IFJcode20 (viz podkapitola 5.1). V definičním souboru je v položce `languages` definováno několik jazyků (IPPcode21, IPPcode20 a IFJcode20). Všechny jazyky používají stejný konfigurační soubor jazyka. Asociace jazyků podle obecné přípony je nastavena pouze u nejnovější verze jazyka, jinak se definice jazyků nemění.

Pro každý jazyk je definována jeho vlastní gramatika. Každá verze gramatiky se liší používaným identifikátorem jazyka (včetně jeho použití v kořenovém rámci a v identifi-

kátoru tokenů). Oproti gramatice definované v prvním rozšíření je hlavička identifikována jako komentář, protože pro komentáře je běžně definovaný styl, navíc se sjednotí zobrazení s jazykem VYPcode, kde hlavičková sekce má formát komentáře. Také je doplněn nový typ komentářů začínající trojicí speciálních znaků na začátku řádku, což jsou speciální komentáře používané v testech interpretu. Z definice tokenů proměnných je navíc odstraněna kontrola, že se nenachází na začátku řádku, aby zvýrazňování mohlo fungovat pro proměnné vyhodnocované v konzoli.

Navíc byla přidána definice pro jazyk IPPeCode. Jedná se o jazyk vycházející z jazyka IPPcode, který je oproti němu zjednodušený. Neobsahuje rámce, podporuje méně typů a má odlišnou jednodušší instrukční sadu.

5.4.2 Ladění

Definice nového typu ladění je obsažena v položce `debuggers`. Definuje se jediný obecný typ ladění pro oba jazyky s identifikátorem `ippc-ifjc`. Definice typu ladění dále obsahuje definici dalších položek konfigurace ladění, které jsou definovány jen pro typ ladění spouštějící program – `launch`. Konfigurace ladění obsahuje stejné položky jako nastavení rozšíření a tři další. Pouze položka `program` je povinná.

1. `program`: definuje cestu k programu, který má být interpretován
2. `args`: pole argumentů, které mají být předány laděnému programu
3. `stopOnEntry`: zastavení programu před zahájením vykonávání instrukcí

Dále je v definici typu ladění definována výchozí konfigurace ladění (položka `initialConfigurations`), která je použita při spuštění ladění tohoto typu bez existující konfigurace, a úryvek kódu s nastavením (v položce `configurationSnippets`) pro poskytnutí možnosti vytvoření předdefinované konfigurace. Navíc je nutné doplnit jazyky, pro které je možné přidávat body přerušení. Toto nastavení je obsaženo mimo položku `debuggers` v položce `breakpoints`, kam se přidají všechny identifikátory jazyků IPPcode a IFJcode.

Kód rozšíření zaregistruje třídu odpovědnou za zpracování konfigurace ladění a třídu odpovědnou za generování adaptéru ladění. První třída implementuje rozhraní `DebugConfigurationProvider` a obsahuje metody, které kontrolují předanou konfiguraci. Do konfigurace jsou doplněny případné výchozí hodnoty a pokud konfigurace neobsahuje povinnou položku, tak se ukončí spouštění ladění. Druhá třída implementuje rozhraní `DebugAdapterDescriptorFactory`. Tato třída obsahuje metodu vracející deskriptor adaptéru ladění (konkrétně číslo portu), která je použita pro zahájení ladění (dostává již zpracovanou konfiguraci).

Metoda se pokusí získat cestu k interpretu jazyka Python z konfigurace ladění, jinak ji získá z nastavení rozšíření a pokud není nastavena ani tam, tak použije výchozí hodnotu `python`. Identicky získá i hodnotu verze jazyka a číslo portu. Pokud není nastaveno číslo portu, tak používá modul `portfinder` k nalezení prvního volného neprivilegovaného portu. Následně vytvoří nový integrovaný terminál a odešle do něj text příkazu na spuštění adaptéru interpretem jazyka Python. Poté počká na spuštění serveru a vrátí zvolené číslo portu. Čekání je prováděno pomocí modulu `wait-port`, který se na daný port pokouší připojit, adaptér proto ve skutečnosti přijme dvě spojení, první slouží k ověření spojení a až druhé je skutečný DAP klient editoru.

5.4.3 Příkazy pro interpretaci a ladění

Možnost spouštění programu v jazycích IFJcode a IPPcode je implementována pomocí příkazů, které jsou definované v sekci `commands`. Spouštění programu je řešeno podobně jako spouštění adaptéru, akorát pracuje pouze s nastavením rozšíření a spouští odpovídající interpret. Pro každou verzi jazyka je definován odpovídající příkaz na spuštění interpretace, pro jazyk IPPcode je navíc doplněna verze pro interpretaci z XML. Navíc jsou doplněny příkazy pro spuštění ladění pro každou verzi jazyka (používají API VS Code pro spuštění ladění a výchozí konfiguraci ladění). Veškeré příkazy jsou deaktivované, pokud je aktivní ladění. V položce `menus` jsou definované příkazy umístěny do výběru příkazů (`Command Palette`) a do záhlaví editoru, kde jsou zobrazené pomocí definovaných ikon, pokud je otevřený a identifikovaný typ souboru, pro který je poskytována podpora.

5.5 Rozšíření pro jazyk VYPcode

Rozšíření pro jazyk VYPcode vychází z popsaného rozšíření pro jazyky IPPcode a IFJcode. Definice a kód rozšíření jsou velmi podobné, pouze neřeší práci s více jazyky, nastavení obsahuje jiné položky, jsou použity jiné identifikátory atp. Propojení s ladicím adaptérem je identické a adaptér podporuje stejné požadavky.

Rozšíření je distribuováno spolu s implementací ladicího adaptéru ve formátu `jar`. Pro účely interpretace kódu se ve výchozím nastavení používá ladění spuštěné s parametrem `noDebug`, tedy bez ladění, které je implementováno pouze blokujícím spuštěním interpretu v rámci adaptéru po přijetí požadavku na spuštění[4], v nastavení rozšíření je ale možné specifikovat cestu k interpretu jazyka VYPcode, který je poté používán k provádění interpretace.

5.5.1 Definice jazyka VYPcode a jeho gramatiky

Rozšíření obsahuje pouze definici jazyka VYPcode, která obsahuje stejné položky jako definice jednotlivých verzí jazyka IPPcode nebo IFJcode kromě rozpoznání na základě prvního řádku, jelikož jazyk VYPcode nemá předepsaný formát hlavičky. Vzhledem k tomu, že jazyk VYPcode nemá roční verze, tak není nutné definovat více verzí ani přípony s číslem verze.

Definice gramatiky jazyka VYPcode má stejnou strukturu a princip přiřazování identifikátorů jako již popsané gramatiky. Operační kódy, komentáře a speciální komentáře pro test jsou definovány identicky, pouze je doplněn druhý typ komentáře začínající znakem středníku `;`. Další tokeny jazyka VYPcode vycházejí z definic literálů ve specifikaci a pro jejich definici byly převzaté regulární výrazy použité v referenčním interpretu. Definován je token registru, obecného jména (identický literál pro konstanty, návěští a alias registru) a konstantních hodnot. Konstanty a jména mohou být v jazyce VYPcode použity jako argument, ale dále i v konstantních výrazech při definici konstanty a spolu s registrem při zásobníkovém adresování. Jsou proto rozšířeny definice hranic tokenů o další možné znaky operátorů a závorek.

5.5.2 Rozšíření projektu interpretu pro jazyk VYPcode

Práce používá pro zpracování zdrojového kódu a interpretaci existující interpret pro jazyk VYPcode (projekt `vypint`), jehož autorem je Ing. Kocman. Projekt obsahuje lexikální

[4] adaptér pro IPPcode/IFJcode se chová identicky, ale parametr se v rozšíření nepoužívá

analyzátor sloužící k načítání jednotlivých tokenů ze vstupu. Práce lexikálního analyzátoru je řízena syntaktickým analyzátorem, který jej používá na načtení specifických typů tokenů podle kontextu. Syntaktický analyzátor vyhodnocuje vstupní tokeny a ukládá načtené hodnoty do paměti programu (instrukce do instrukční paměti, řetězcové konstanty na haldu). Interpretace je řízena třídou `Engine`, která provádí zpracování argumentů, zpracování vstupu a jeho následnou interpretaci.

Pro účely ladění bylo nutné provést několik modifikací původního projektu. Především byla doplněna možnost nastavovat pro instrukce body přerušení a do interpretu možnost provádět program po jednotlivých krocích (tzn. provést jen jednu instrukci), kontrolovat body přerušení a přístup k právě vykonávaným instrukcím. Do tříd implementujících paměť programu v jazyce VYPcode byla doplněna možnost přímého přístupu k této paměti pro účely zobrazování hodnot a přístup k jednotlivým hodnotám bez počítání statistik (běžné metody pro přístup k paměťovým místům slouží jen pro účely vykonávání instrukcí). Také byly doplněny nové tokeny pro identifikátory vyhodnotitelných paměťových míst.

5.5.3 Implementace ladicího nástroje komunikujícího přes protokol Debug Adapter

Implementace ladicího adaptéru a samotných ladicích funkcí je obsažena v nových třídách, které jsou obsaženy v odděleném projektu `vypadapter`, který závisí na projektu `vypint`. Dochází tak k minimálním zásahům do původního projektu a každý z projektů může být přeložen do spustitelného souboru obsahujícího pouze nutné třídy a závislosti pro daný projekt.

Jako první je nutné zajistit přijímání a zpracování DAP zpráv. Zpracování JSON obsahu zpráv je v jazyce Java netriviální proces, který vyžaduje použití externích knihoven jako je např. `Jackson` nebo `Gson`, které umožňují převést přijatá JSON data na objekt třídy, která definuje strukturu přijatých dat. Musejí tedy existovat definice všech použitých typů zpráv, které následně mohou být použity pro zpracování obsahu přijatých zpráv. Pro jazyk Java existuje knihovna implementující protokol Debug Adapter, která je pro implementaci adaptéru použita.

Knihovna implementující protokol Debug Adapter je součást projektu `lsp4j` (Language Server Protocol for Java) od společnosti Eclipse, jehož jádrem je implementace protokolu JSON-RPC pro jazyk Java, nad kterým je postavena podpora pro protokoly Language Server a Debug Adapter. Projekt je používán primárně pro propojení existujících nástrojů s IDE Eclipse. Oficiální rozšíření pro jazyk Java ho nepoužívá, ale obsahuje vlastní interní implementaci, která však funguje podobně a shodně zpracovává přijaté zprávy knihovnou `Gson` od společnosti Google. Knihovna je použita v rozšíření pro jazyk Ballerina nebo například v projektu Kotlin Debug Adapter.

Knihovna definuje třídy pro práci s DAP zprávami, zpracování přijatých zpráv a zajišťuje veškerou komunikaci. Při inicializaci adaptéru je nutné poskytnout objekt implementující DAP požadavky a objekty pro vstup a výstup sloužící ke komunikaci s klientem, pro což je použit socket TCP spojení. Pro implementaci DAP požadavků existuje rozhraní `IDebugProtocolServer`, které obsahuje metody pro jednotlivé DAP požadavky. Metody požadavků dostávají jako argument objekt již zpracovaných argumentů a vracejí objekt s argumenty specifickými pro odpověď.

Po dokončení inicializace a zahájení komunikace je řízení řešeno knihovnou. Na základě přijatých zpráv jsou volány odpovídající metody serveru[5] a následně odesílány odpovědi podle vrácených hodnot.

Metody rozhraní `IDebugProtocolServer` nemají přístup k samotným odpovědím, ale pouze k objektu argumentů specifických pro danou odpověď. Nemohou proto přímo vrátit chybovou odpověď. Pokud vykonání metody proběhne úspěšně, tak je odeslána úspěšná odpověď s vrácenými argumenty, při výjimce je odeslána výchozí zpráva o neúspěchu. Pro vytvoření chybové odpovědi slouží výjimka `ResponseErrorException`, pro jejíž použití je nutné při inicializaci předefinovat výchozí zpracovávání výjimek. Pro odesílání událostí existuje objekt rozhraní `IDebugProtocolClient`, které obsahuje metody pro zasílání událostí klientovi.

Požadavky jsou implementovány v třídě `VYPcServer`. Implementace jednotlivých požadavků a jejich chování je podobné jako v případě adaptéru pro IPPcode a IFJcode. Pro práci s interpretem slouží třída `DebugEngine`, která rozšiřuje třídu `Engine` o interaktivní práci s interpretem, zasahování do jeho paměti a další pomocné funkce. Samotné provádění instrukcí je opět implementováno v dalším vlákně, kterému mohou být zadávány požadavky na pozastavení.

V pohledu proměnných jsou vždy zobrazeny všechny existující registry, použitá paměťová místa na zásobníku a všechny existující shluky na haldě. Shluky na haldě jsou reprezentovány jako složená proměnná obsahující jednotlivé položky. Veškerá paměť v jazyce VYPcode je složena z 64bitových částí reprezentovaných typem `Long`. Paměťová místa nenesou žádnou informaci o typu uložené hodnoty, práce s typem je dána pouze použitou instrukcí. Při zobrazování v pohledu proměnných jsou proto všechny hodnoty zobrazovány v základním celočíselném formátu, hodnota shluku zobrazuje počet jeho položek a řetězcovou reprezentaci jeho položek.

Pro sledování hodnot konkrétních proměnných se jako identifikátory používají tokeny registru a adresování na zásobníku, navíc je doplněno adresování na haldě, které umožňuje identifikovat shluk pomocí čísla jeho identifikátoru případně položku shluku pomocí dalšího indexu. Všechny tři identifikátory paměťových míst navíc mohou končit formátovací značkou sloužící k zobrazení hodnoty v celočíselném, desetinném nebo hexadecimálním desetinném formátu.

Každému paměťovému místu v pohledu proměnných je možné nastavit novou celočíselnou nebo desetinnou hodnotu v identickém formátu jako ve zdrojovém kódu. Modifikace celého shluku přímým nastavením hodnoty není povolená. Pro ukládání řetězců je možné použít konzoli, která umožňuje vyhodnocování dalších instrukcí. Vyhodnocování instrukcí v konzoli umožňuje vykonat jakoukoli instrukci kromě instrukcí definice návěští, načítání vstupu a instrukcí ladicích výpisů.

Pro zpracování identifikátorů je použita třída `EvalParser`, která rozšiřuje syntaktický analyzátor interpretu a využívá jeho metody na zpracování instrukcí. Dále umožňuje zpracování hodnot pro nastavování proměnných a zpracování nových tokenů identifikátorů paměťových míst přidaných do lexikálního analyzátoru.

[5] pozn. výchozí implementace metod vyvolá výjimku, která ukončí server, proto je nutné předefinovat všechny metody pro základní požadavky

5.6 Testování

Integrační testování správného fungování rozšíření bylo prováděno v průběhu implementace manuálním používáním vytvořeného rozšíření. Editor VS Code umožňuje spouštět vytvářené rozšíření v nové instanci editoru, kde je rozšíření aplikováno. V tomto prostředí je následně možné sledovat zobrazování zdrojového kódu a případně ho dále analyzovat pomocí zobrazení identifikovaných TextMate tokenů. Stejně je možné používat doplněné příkazy editoru a nový typ ladění, který byl testován především. Ladění bylo testováno na různých souborech se zdrojovým kódem pro všechny podporované jazyky při změnách nebo doplnění dalších DAP požadavků. Testování probíhalo primárně na operačním systému Linux Mint, na kterém bylo rozšíření vyvíjeno, jelikož operační systém Linux je primární cílová platforma. Zveřejněné verze rozšíření byly navíc testovány na operačním systému Windows 10. Pro testování jednotlivých částí práce navíc slouží automatizované testy.

Pro ověření správného fungování syntaktického analyzátoru a interpretu pro IPPcode byly používány studentské testy z předmětu IPP[6]. Testy bylo možné použít díky implementaci podpory pro vstup a výstup ve formátu XML, zajišťující kompatibilitu se skripty ze zadání projektu z předmětu IPP.

Podpora pro jazyk IFJcode je odvozena ze stejných tříd jako testované třídy syntaktického analyzátoru a interpretu pro IPPcode, je tedy ověřena i základní správná funkčnost interpretu pro IFJcode. Pro testování upravených instrukcí je doplněn test používající testovací nástroj `pytest`. Testy jsou prováděny spuštěním programu v jazyce IFJcode20 nebo voláním odpovídajících metod a následným výpisem pomocí instrukce `WRITE` ve vytvořeném interpretu a porovnání výstupu s výstupem ekvivalentního programu spuštěného pomocí referenčního interpretu.

Projekt interpretu pro jazyk VYPcode obsahuje testy používající aplikační rámec pro testování `JUnit`, tyto testy byly používány pro ověření správnosti fungování interpretu po provedených úpravách. Navíc byly doplněny `JUnit` testy pro odvozené třídy `EvalParser` a `DebugEngine` v projektu adaptéru.

5.6.1 Testování adaptéru

Vytváření automatizovaných testů pro ladicí adaptér je usnadněno existujícím `npm` modulem `vscode-debugadapter-testsupport` sloužícím pro tyto účely. Modul umožňuje vytvořit DAP klienta, který se připojí k adaptéru buďto zadaného pomocí příkazu ke spuštění procesu adaptéru, nebo pomocí portu. Prostřednictvím metod klienta je možné zasílat DAP požadavky, na které je vrácena odpověď pomocí asynchronních objektů `Promise`. Dále umožňuje čekání na přijetí událostí od adaptéru a několik doplňujících funkcí pro kontrolu typických situací jako je např. pozastavení na bodu přerušení.

Pro vytváření testů je použit testovací rámec `mocha` v kombinaci s knihovnou `chai`. Testy byly primárně vytvořeny pro jazyk IPPcode21, jelikož adaptér je pro všechny verze jazyků IPPcode a IFJcode identický. Pro další verze jazyků existuje pouze základní test funkčnosti ladění. Testování je prováděno zasíláním požadavků a porovnáváním odpovědí s očekávaným chováním. Existují testy pro všechny podporované požadavky. Vytvořené testy adaptéru pro jednotlivé požadavky byly následně upraveny pro jazyk VYPcode a použity i k testování adaptéru pro VYPcode.

[6] https://github.com/jk8/ipp-2020-tests

5.6.2 Uživatelské testování

V průběhu práce byly pro studenty zveřejněny dvě rozšíření, první doplňující syntaktické zvýrazňování a jako druhé již plnohodnotné rozšíření pro jazyky IPPcode a IFJcode. Obě rozšíření mají přes 50 stažení a studenti měli možnost informovat o případných chybách, žádná chyba ale oznámena nebyla.

Pro zjištění informací o jejich používání rozšíření byl vypracován dotazník, který vyplnilo celkem 13 studentů. Vzorek je poměrně malý, ale je možné z něj zjistit základní informace. Žádný z respondentů neměl problém se samotnou instalací a spuštěním rozšíření a pouze jeden respondent narazil na problém při používání rozšíření, o kterém bohužel nenapsal žádné další informace. Většina studentů používala rozšíření na operačním systému Linux, což byla očekávaná primární platforma, několik jich ale rozšíření vyzkoušelo na operačním systému Windows a jeden respondent i na operačním systému Mac, čímž je ověřena i správná multiplatformní funkčnost rozšíření.

Většina respondentů nicméně odpověděla, že rozšíření používali spíše minimálně nebo jen na zkoušku a jen sedm vyzkoušelo ladění. Jako největší přínos většina uvedla syntaktické zvýrazňování. Na otázku, zda si myslí, že by rozšíření bylo přínosné pro seznámení s jazykem IFJcode v předmětu IFJ, odpověděli všichni kladně.

Kapitola 6

Závěr

Cílem práce bylo doplnit do existujícího editoru zdrojových textů podporu pro editování, spouštění a ladění programů v jazycích mezikódu. Bylo nutné seznámit se s možnostmi vytváření rozšíření pro současné editory a integrovaná vývojová prostředí. Následně bylo třeba nastudovat vlastnosti jazyků mezikódu IFJcode a VYPcode a pro jeden z editorů navrhnout a implementovat rozšíření pro tyto jazyky.

Teoretická část práce obsahuje specifikaci obou jazyků mezikódu a jazyka IPPcode, který vychází z jazyka IFJcode. Poté následuje přehled editorů Eclipse, NetBeans a Visual Studio Code se zaměřením na možnosti doplnění požadovaných funkcí v rámci rozšíření pro tyto editory. Také je popsán přehled běžných funkcí a cílů ladění programů.

Dále se práce podrobněji zabývá vytvářením rozšíření pro editor VS Code, který byl zvolen pro implementaci rozšíření. Je popsána obecná struktura rozšíření a je zmíněn implementační jazyk TypeScript. Pro účely doplnění syntaktického zvýrazňování je popsána technologie TextMate gramatik. Dále jsou zmíněny protokoly pro komunikaci editorů s externími nástroji poskytujícími podporu pro editované jazyky. Zaměření je především na protokol Debug Adapter (DA), který slouží ke komunikaci s ladicím nástrojem. Také je uveden příklad několika existujících rozšíření pro interpretované jazyky.

Na základě získaných informací byl vytvořen návrh rozšíření pro jazyky IPPcode a IFJcode a dalšího rozšíření pro jazyk VYPcode. Ladění je implementováno adaptérem používajícím protokol TCP pro komunikaci s editorem přes protokol DA. Standardní vstup a výstup je tak k dispozici pro laděný program, který je vykonáván interpretem obsaženým v adaptéru. Instrukce jsou vykonávány v dalším vlákně, které lze kooperativně pozastavovat, a případné provádění blokujících instrukcí tak neomezuje hlavní řídicí vlákno.

Následuje popis implementace podpory pro jazyky mezikódu. Jsou popsány definice rozšíření, gramatiky jazyků mezikódu a kód rozšíření v jazyce TypeScript. Pro účely interpretace a ladění byl vytvořen interpret jazyka IPPcode v jazyce Python, který byl modifikován pro doplnění potřebné kompatibility s jazykem IFJcode. Řeší se také implementace komunikace přes protokol Debug Adapter a zpracování jeho zpráv, jelikož pro jazyk Python neexistuje sada vývojových nástrojů (*SDK*) pro protokol DA. Podpora pro jazyk VYPcode rozšiřuje existující referenční interpret napsaný v jazyce Java, jehož autorem je Ing. Radim Kocman. Interpret je modifikován a doplněn o metody potřebné pro účely ladění. Ladění je implementováno v odděleném projektu, který využívá knihovnu implementující komunikaci přes protokol DA pro jazyk Java.

Vytvořená rozšíření doplňují pro jazyky mezikódu syntaktické zvýrazňování, umožňují jejich ladění a případně jejich interpretaci ve vestavěném terminálu editoru. Cíle této práce tak byly úspěšně splněny. Definice jazyků vytvořené v rozšířeních je možné použít jako

základ pro doplnění podpory pro další verze jazyků mezikódu. Interpret pro jazyk IPPcode je možné upravit pro účely dalších verzí jazyka např. předefinováním chování jednotlivých instrukcí, jako tomu bylo při modifikaci pro jazyk IFJcode20. Ladicí adaptér je možné rozšířit o podporu pro podobné jazyky, jako je např. IPPeCode, pomocí předefinování ladicích funkcí pro práci s jiným paměťovým modelem atp. Provedené úpravy interpretu jazyka VYPcode by měly spolu s nově vytvořeným projektem ladicího adaptéru být začleněny do původního projektu a stát se tak součástí referenční implementace, která může být jako celek dále vyvíjena.

Literatura

[1] Aho, A. V., Lam, M. S., Sethi, R. a Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. 2. vyd. Addison Wesley, srpen 2006. ISBN 0321486811.

[2] The Apache Sotfware Foundation. *Apache NetBeans History* [online]. 2019 [cit. 2020-23-12]. Dostupné z: https://netbeans.apache.org/about/history.html.

[3] The Apache Sotfware Foundation. *JavaCC Lexer Generator Integration Tutorial for the NetBeans Platform* [online]. 2020 [cit. 2020-25-12]. Dostupné z: https://netbeans.apache.org/tutorials/nbm-javacc-lexer.html.

[4] Beaton, W., Weinstein, J., Windatt, C., Arthorne, J. et al. *Eclipse Project* [online]. 2020 [cit. 2020-25-12]. Dostupné z: https://wiki.eclipse.org/index.php?title=Eclipse_Project&oldid=441688.

[5] Bettini, L. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*. 2nd. Packt Publishing, 2016. ISBN 1786464969.

[6] Eclipse Foundation. *Syntax coloring* [online]. 2020 [cit. 2020-26-12]. Dostupné z: https://help.eclipse.org/2020-12/index.jsp?topic=/org.eclipse.platform.doc.isv/guide/editors_highlighting.htm.

[7] Gray II, J. E. *TextMate: Power Editing for the Mac*. 1. vyd. Pragmatic Bookshelf, 2007. Pragmatic Bookshelf Series. ISBN 0-9787392-3-X.

[8] Ho, E. *Creating a text-based editor for Eclipse*. Hewlett-Packard Company, červenec 2003 [cit. 2020-26-12].

[9] International Business Machines Corporation. *Eclipse Platform Technical Overview* [online]. 2020 [cit. 2020-25-12]. Dostupné z: https://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf.

[10] Kaplan, V. Writing Your Own Debugger and Language Extensions with Visual Studio Code. *CODE magazine* [online]. 1. vyd. EPS Company. 2018, September/October, [cit. 2020-28-12]. ISSN 1547-5166. Dostupné z: https://www.codemag.com/article/1809051/Writing-Your-Own-Debugger-and-Language-Extensions-with-Visual-Studio-Code.

[11] Křivka, Z., Kocman, R. a Regéciová, D. VYPa*: THE COMPILER PROJECT SPECIFICATION*. 2020.

[12] Křivka, Z., Zobal, L. a Regéciová, D. *Zadání projektu z předmětu IFJ a IAL*. 2020 [cit. 2021-18-03].

[13] Křivka, Z. *Zadání projektu z předmětu IPP 2019/2020*. 2020.

[14] MacroMates Ltd. *Language Grammars* [online]. 2021 [cit. 2021-12-01]. Dostupné z: https://macromates.com/manual/en/language_grammars.

[15] Medeiros, B. D. O. a Leitão, A. P. T. d. M. C. *Creation of an Eclipse-based IDE for the D programming language*. Lisabon, 2007. Diplomová práce. Technická universita v Lisabonu. Dostupné z: https://fenix.ist.utl.pt/dissertacoes/37436.

[16] Microsoft Corporation. *Documentation for Visual Studio Code* [online]. 2020 [cit. 2020-28-12]. Dostupné z: https://code.visualstudio.com/docs.

[17] Microsoft Corporation. *Extension API* [online]. 2020 [cit. 2020-29-12]. Dostupné z: https://code.visualstudio.com/api.

[18] Microsoft Corporation. *Official page for Language Server Protocol* [online]. 2020 [cit. 2020-29-12]. Dostupné z: https://microsoft.github.io/language-server-protocol/.

[19] Microsoft Corporation. *Debug Adapter Protocol* [online]. 2021 [cit. 2021-17-03]. Dostupné z: https://microsoft.github.io/debug-adapter-protocol/.

[20] Oracle Corporation. *SchliemannNBSLanguageDescription* [online]. 2009 [cit. 2020-25-12]. Dostupné z: http://wiki.netbeans.org/SchliemannNBSLanguageDescription.

[21] Oracle Corporation. *GLFTutorial* [online]. 2010 [cit. 2020-25-12]. Dostupné z: http://wiki.netbeans.org/GLFTutorial.

[22] Rosenberg, J. B. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. 1. vyd. USA: John Wiley & Sons, Inc., 1996. ISBN 0471149667.

Příloha A

Obsah přiloženého paměťového média

- `IFJc20-syntax` – rozšíření doplňující syntaktické zvýrazňování pro jazyk IFJcode20
- `IPPc-IFJc` – rozšíření pro jazyky IPPcode a IFJcode
  - `src` – kód rozšíření
  - `debug` – interpret, syntaktický analyzátor a adaptér ladění
- `text` – zdrojové soubory textu této práce
- `VYPcDebug`
  - `VYPcExtension` – rozšíření pro jazyk VYPcode
  - `vypint` – interpret jazyka VYPcode
  - `vypadapter` – adaptér ladění pro jazyk VYPcode