VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

ZPRACOVÁNÍ JAZYKA C V PROHLÍŽEČI NA BÁZI .NET

.NET BASED C LANGUAGE PROCESSING IN A BROWSER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. MICHAL KUŽELA**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Dr. Ing. DUŠAN KOLÁŘ**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Kužela Michal, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Informační systémy a databáze
Název: **Zpracování jazyka C v prohlížeči na bázi .NET**
.NET Based C Language Processing in a Browser
Kategorie: Překladače
Zadání:

1. Studujte platformu .NET pro možnost kombinace s WebAssembly a offline nasazení v prohlížeči. Nastudujte možnost zpracovaní jazyka C touto metodou, studujte možnosti modelování ukazatelů, proměnlivého počtu parametrů, generování kódu do WebAssembly.
2. Po konzultaci s vedoucím, na základě výstupů z předchozího bodu navrhněte implementaci zpracování jazyka C uvnitř prohlížeče včetně editoru a možnosti vložit příkaz pro okamžité vyhodnocení.
3. Dle návrhu a podle doporučení vedoucího implementujte návrh z předchozího bodu - uvažte standardní knihovny a jejich korektní navázání pro přenos zdrojového kódu metodou "cut-n-paste".
4. Vše otestujte, na rychlost, spotřebu paměti, apod. ve více prohlížečích.
5. Zhodnoťte přínos vaší práce, diskutujte možná rozšíření směrem k ladění.

Literatura:

- Aho, Alfred Vaino; Lam, Monica Sin-Ling; Sethi, Ravi; Ullman, Jeffrey David (2006). Compilers: Principles, Techniques, and Tools (2 ed.). Boston, Massachusetts, USA: Addison-Wesley. ISBN 0-321-48681-1
- ISO/IEC 9899:2011 - Information technology -- Programming languages -- C
- Dle doporučení vedoucího

Při obhajobě semestrální části projektu je požadováno:

- První 2 body a rozpracovaný bod 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/
Vedoucí práce: **Kolář Dušan, doc. Dr. Ing.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 18. října 2021

Abstrakt

Cílem této práce je umožnit programování v jazyce C uvnitř prohlížeče a to i v režimu offline. Zaměřuje se na technologie .NET a WebAssembly. Implementace proběhla ve frameworku Blazor WebAssembly. Důraz byl kladen na pokrytí rozsahu jazyka C využívaného začínajícími programátory. Vytvořený interpret poskytuje prostředí pro programování v prohlížeči, obsahuje virtuální souborový systém a možnost vložení uživatelského vstupu. Výhodou tohoto řešení je možnost práce offline, krokování kódu a možnost okamžitého vyhodnocení kódu.

Abstract

The aim of this work is to create a tool for online programming in browser that would work also offline. It focuses on WebAssembly and .NET technologies. It was implemented in Blazor WebAssembly framework. Emphasis was placed on covering the range of C programming language used by beginner programmers. Solution is an interpreter that provides environment for programming in browser. It also contains virtual filesystem and an option to insert user inputs. The advantage of this solution is the possibility to work offline, code stepping and possibility to perform immediate code evaluation.

Klíčová slova

Kompilace, interpretace, jazyk C, WebAssembly, Blazor, .NET 6, offline zpracování, PWA, C#, .NET 7, krokování kódu, preprocesor

Keywords

Interpret, compilator, C language, WebAssembly, Blazor, .NET 6, offline processing, PWA, C#, .NET 7, code-stepping, preprocessor

Citace

KUŽELA, Michal. *Zpracování jazyka C v prohlížeči na bázi .NET*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Dr. Ing. Dušan Kolář

Zpracování jazyka C v prohlížeči na bázi .NET

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana doc. Dr. Ing. Koláře Dušana. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Michal Kužela
16. května 2022

Poděkování

Děkuji panu doc. Dr. Ing. Koláři Dušanovi za vedení a konzultace při zpracování této diplomové práce. Také děkuji manželce Simoně Kuželové za podporu, Kryštofu Klabanovi za pomoc s UX interpretu a také všem kteří se podíleli na uživatelském testování.

Obsah

Kapitola 1

Úvod

Tato práce se zabývá problematikou zpracování jazyka C v prohlížeči. WebAssembly je relativně nová technologie, která umožňuje přenos počítačových programů do prohlížeče a otevírá tak nové možnosti práce přímo v prohlížeči, bez nutnosti využití serverového připojení. V kombinaci s platformou .NET se naskytuje možnost využít síly této platformy k implementaci nástroje pro zpracování jazyka C.

Aktuálně existuje několik řešení pro zpracování jazyka C v prohlížeči, ale všechny nalezené řešení využívají pro svou činnost připojení k serveru, který zpracovává zdrojový kód. To znamená, že uživatelé těchto platforem nemohou v případě odpojení od internetové sítě pokračovat ve své práci. Cílem této práce je odstranit tento nedostatek poskytnutím možnosti pracovat i po odpojení uživatele od internetové sítě.

Tato práce se soustředí primárně na začínající programátory, konkrétně studenty předmětu IZP na škole FIT VUT, na jehož rozsah cílí. Třeba studenti, kteří tráví čas na cestách vlakem, se mohou potýkat s nekvalitním připojením k internetové síti. V případě, že využívají některý z existujících online nástrojů, pak nemohou spolehlivě pracovat na svých projektech a přichází o čas, ve kterém by se mohli vzdělávat. To jim navrhované řešení umožní, protože po prvotním načtení bude možné prostředí používat bez internetu.

Kromě možnosti offline práce nabízí navrhované řešení oproti běžným nástrojům možnost okamžitého vyhodnocení příkazu, rozšíření o ladění kódu, podporu cut-n-paste a také hlášení chyb ve zdrojovém kódu, například upozornění o neuvolnění paměti. Chybová hlášení mohou studentům pomoci pochopit principy programování v jazyce C.

Využití platformy .NET pak umožňuje implementaci pomocí objektově orientovaných principů, které usnadní znovupoužitelnost a rozšiřitelnost řešení. Je tak možné na těchto základech dále stavět i v rámci případných navazujících prací.

Text práce je rozdělen na několik hlavních částí, které jsou seřazeny chronologicky podle toho jak probíhala tato diplomová práce.

Kapitola 2 popisuje teorii, kterou bylo potřeba nastudovat v souvislosti s jazykem C, jeho zpracováním s ohledem na technologii WebAssembly a platformu .NET. V rámci studia bylo potřeba také prozkoumat existující řešení, které jsou zdokumentovány v kapitole 3. V kontextu nalezených řešení byl proveden průzkum využití existujících přístupů v kombinaci s platformou .NET a WebAssembly, ale také analýza zcela nových přístupů v kapitole 4. Získané znalosti byly využity v kapitole 5, která se zabývá návrhem řešení. Následně je řešení implementováno a popsáno v kapitole 6. V poslední kapitole 7 jsou uvedeny výsledky testování vytvořené aplikace napříč prohlížeči a srovnání s podobným nástrojem pro jazyk C++.

Vytvořený nástroj je umístěn na adrese `https://xkuzel06-learninc.cz/`.

Kapitola 2

Jazyk C, .NET a technologie souvisejících s tématem

Tato kapitola uvádí důležitou teorii přímo související s jazykem C a možnostmi jeho zpracování v prohlížeči s důrazem na požadavek možnosti využití bez internetového připojení. Kapitola pokrývá zásadní informace potřebné pro učinění rozhodnutí týkajících se návrhu a implementace. Cílem této kapitoly není popsat jednotlivé technologie do detailu, ale seznámit čtenáře se základními rysy využitých a zkoumaných technologií a tím dodat potřebný kontext pro pochopení učiněných rozhodnutí.

Vzhledem ke komplexitě a různosti jednotlivých témat v této kapitole bylo zvoleno členění do podkapitol, které značí jeden významový celek a podsekcí uvádějící potřebné informace.

V první podkapitole 2.1 bude čtenář seznámen s vybranými koncepty jazyka C, ve druhé podkapitole 2.2 s technologií WebAssembly, ve třetí 2.3 se dozví o platformě .NET a poslední podkapitola 2.4 jej seznámí se základními rozdíly mezi překladačem a interpretem.

2.1 Jazyk C

V této podkapitole jsou uvedeny komplexní části jazyka C, které výrazně ovlivnily rozhodování při návrhu a implementaci. Primárním zdrojem při studiu, návrhu a implementaci byl standard jazyka C [10], ve kterém lze dohledat i další informace neuvedené v této podkapitole pro jejich příliš velký detail.

2.1.1 Struktura paměti

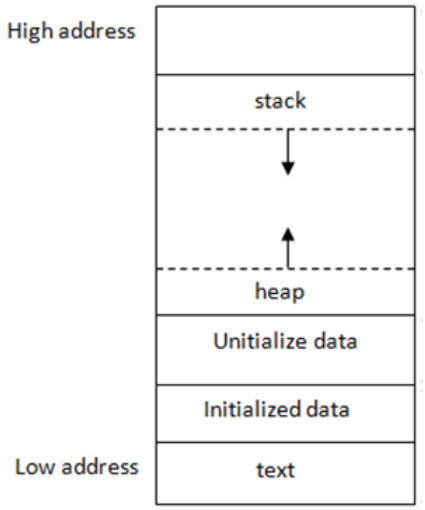
Paměť a práce s ní v kontextu jazyka C je důležitým prvkem, který je nutné dobře pochopit pro správnou implementaci. V této podsekci je rozvedena její struktura a obsah. Následující informace o struktuře paměti v programech jazyka C jsou převzaty primárně z knihy [13].

- Textový / kódový segment
  - Obsahuje spustitelný kód programu, typicky je tento blok paměti pouze pro čtení a pevné velikosti.
- Datový segment

  - Datový segment se dělí na inicializované a neinicializované data, je naplněn operačním systémem staticky deklarovanými proměnnými ještě před začátkem vykonávání programu.
  - Inicializované data obsahují předem inicializované globální, statické, konstantní a externí proměnné včetně jejich přiřazených hodnot.
  - Neinicializované data nemají přiřazenou konkrétní hodnotu před začátkem vykonávání programu, jejich hodnoty jsou vynulovány a přiřazují se až za běhu programu.
  - Alokované statické/globální proměnné existují po celý život aplikace. Statické proměnné jsou vázány na funkci, která je definuje.

- Hromada
  - Hromada obsahuje dynamicky alokovanou paměť, kterou programátor spravuje pomocí knihovních funkcí malloc, calloc, realloc a free. Paměť je sdílená globálně mezi vlákny, knihovnami i dynamicky načtenými moduly, ale přístup k datům v ní probíhá pouze přes ukazatele, které odkazují na lokaci paměti, kde se data nachází. Vzhledem k tomu, že alokaci paměti neřídí systém automaticky jako u ostatních částí paměti, ale děje se tak dynamicky za běhu programu, mohou v paměti vznikat nespojité oblasti uvolněných a alokovaných bloků. Informace o uvolněných a alokovaných blocích se udržují v lineárním seznamu a jsou využívány při alokaci nové paměti, kdy mohou tyto bloky být znovu využity pro alokaci nového bloku, pokud je uvolněné místo dostatečně velké. Veškerou alokovanou paměť je nutné uvolnit, pokud se již nepoužívá, jinak může docházet k přetečení paměti a pádu programu.
  - Alokovaná data existují až do explicitního uvolnění.

- Zásobník
  - Zásobník je implementován jako last-in-first-out fronta. Ukládají se na něj lokální proměnné funkcí a využívá se také pro předávání argumentů funkcím, při jejichž volání vzniká nový rámec na který se vloží argumenty a také lokální proměnné volané funkce, jejichž identifikátory mohou být shodné s identifikátory proměnných v jiných rámcích zásobníkové struktury – jednotlivé rámce jsou od sebe odděleny a nemohou navzájem přistupovat ke svým proměnným – mimo proměnné globální a předávané jako argumenty. Struktura zásobníku umožňuje volání rekurzivních funkcí, kdy každé rekurzivní volání má svůj vlastní rámec. Po opuštění funkce je rámec a jeho proměnné uvolněn. V případě přistoupení na adresu proměnné pocházející z již uvolněného rámce je chování nedefinováno, což může například zmást začínající programátory, jelikož do okamžiku přepsání hodnoty na dané adrese v paměti může mít proměnná stále původní hodnotu a tvářit se tak, že program funguje správně. V závislosti na implementaci pak zásobník obsahuje mimo lokální proměnné také adresu odkud byla funkce volána, ukazatel na předchozí rámce a ukazatel na vrchol zásobníku.
  - Alokované proměnné se uvolní po ukončení vykonávání funkce.



Obrázek 2.1: Struktura paměti programu v jazyce C ukazuje uspořádání jednotlivých částí paměti a směr jejich rozšiřování, tak jak je popsáno v 2.1.1. Převzato z [11]

Ukazatele

Následující informace jsou čerpány primárně z knihy [22].

Ukazatele v jazyce C slouží pro manipulaci s daty, umožňují psát rychlý a efektivní kód, podporují dynamickou alokaci a zjednodušují předávání dat. Jsou také způsobem, jak přistoupit k dynamicky alokované paměti.

Ukazatel je proměnná, která obsahuje adresu jiné proměnné, objektu nebo funkce. Objektem je myšlena paměť alokovaná na hromadě za použití funkce pro alokaci paměti, například malloc. Při práci s ukazateli by měla být vždy zajištěna správná inicializace. Jinak může ukazatel obsahovat náhodnou adresu a jeho chování není definované. To může mít za důsledek i pád aplikace. Jeden z možných způsobů ověřování inicializace je přiřazení hodnoty NULL po uvolnění ukazatele.

K ukazatelům se váže několik operátorů, které jsou uvedeny v tabulce 2.1. Mezi často používané patří operátor hvězdička, který slouží pro deklaraci ukazatele ale také jako dereference - získání hodnoty na adrese, kterou ukazatel obsahuje. Samostatnou kapitolou jsou operátory plus, minus a porovnávací operátory, ty se využívají pro aritmetiku nad ukazateli, která umožňuje vypočítat a přiřadit novou adresu ukazatele.

Aritmetika nad ukazateli umožňuje přičítat a odčítat číselné hodnoty k ukazatelům, přičítat a odčítat ukazatele navzájem a také je mezi sebou porovnávat. Užitečnou funkcí je právě přičtení nebo odečtení číselné hodnoty k ukazateli - může sloužit například k pohybování se v rámci pole. Je ovšem potřeba dbát na opatrnost a ujistit se, že přistupujeme na správná místa v paměti. Porovnávání se využívá například pro zjištění relativního pořadí prvků v poli.

Operátor	Název	Popis
*	-	deklarace ukazatele
&	adresa	získání adresy operandu
*	dereference	získání hodnoty na adrese
->	point-to	umožňuje přístup k polím struktury
+, -	sčítání, odčítání	slouží pro aritmetiku ukazatelů
==, !=	rovnost, nerovnost	porovnávání ukazatelů
>, >=	větší, větší rovno	
<, <=	menší, menší rovno	
(datový typ)	přetypování	změna typu ukazatele

Tabulka 2.1: Tabulka zobrazující operátory vztahujících se k ukazatelům. Viz. [22].

Ukazatele je možné také vícenásobně řetězit a tím vytvářet ukazatele na ukazatele, tato vlastnost své využití najde například při práci s více-dimenzionálními poli nebo pro deduplikaci paměti.

2.1.2 Fáze překladu zdrojového kódu

Prozkoumání průběhu překladu jazyka C bylo důležité pro správné pochopení a návrh architektury aplikace. Podle oficiální dokumentace se překlad zdrojového kódu dělí na osm fází, některé však mohou být v závislosti na konkrétní implementaci sjednoceny a popis je tedy spíše konceptuální.

V první fázi se zpracují více-bajtové znaky do odpovídající znakové sady a nahradí se trigraph sekvence (posloupnost tří znaků) za jejich jednoznakovou reprezentaci. Následně se ve druhé fázi zpracují nové řádky zakončené zpětným lomítkem tak, aby aby formovali logické zdrojové řádky.

Ve třetí fázi probíhá dekompozice do preprocessing tokenů a sekvencí bílých znaků včetně komentářů, které jsou nahrazeny za jeden bílý znak. Navazuje čtvrtá fáze, ve které se expandují makra a provádějí direktivy preprocesoru. Pokud zdrojový kód obsahuje direktivu #include, vloží se do zdrojového kódu obsah odkazovaného souboru, který je rekurzivně vyhodnocen první až čtvrtou fází překladu.

V páté fázi jsou všechny znaky zdrojové sady převedeny na znakovou sadu exekučního prostředí. Poté jsou v šesté fázi spojeny sousedící textové řetězce do jednoho. Po ukončení sedmé fáze jsou všechny preprocessing tokeny převedeny na tokeny, které podléhají syntaktické a sémantické analýze. Poslední fáze vyhodnotí všechny externí reference, knihovny a funkce, které nebyly definovány v aktuálně přeloženém kódu. [10]

2.1.3 Proměnlivý počet parametrů funkce

Jazyk C umožňuje definovat variadické funkce s předem neupřesněným počtem parametrů, takové funkce se vyznačují poslední parametrem v podobě tří teček, jedná se například o funkci printf.

```
int printf(const char * restrict format, ...);
```

Takto zadaný parametr musí být vždy na poslední pozici a lze jej použít jen jednou. Před variadickým parametrem se musí nacházet alespoň jeden pojmenovaný parametr. Při volání funkce probíhají nad variadickými argumenty běžné typové konverze.

Pro práci s parametry v rámci funkce se používají funkční makra va_start, va_arg, va_copy, va_end a va_list, které slouží k získání a práci s hodnotami parametrů.

2.2 WebAssembly

WebAssembly [26] [17] je relativně nová technologie, která umožňuje běh výpočetně náročných operací v prostředí moderních prohlížečů s výkonností blížící se nativnímu kódu. Cílem WebAssembly není nahradit v prohlížečích JavaScript, naopak je navržen tak, aby se tyto dvě technologie vzájemně doplňovaly a dokázaly společně komunikovat. Je tak možné volat JavaScriptové funkce z WebAssembly a naopak.

Jedná se o binární instrukční formát pro zásobníkový virtuální stroj, který je zamýšlen primárně jako cílový formát kompilace z jiných programovacích jazyků a to například C, C++, C# nebo Rust. Nejedná se tedy o programovací jazyk a neočekává se, že by někdo ve WebAssembly tvořil kód přímo. Poskytuje však i textový formát podobný tří-adresnému kódu, který lze využít ke čtení a editaci binárního formátu. Ten je ale navržen spíše pro ladění kódu v rámci browseru.

Pro ilustraci možností WebAssembly můžeme uvést tři projekty, které byly postaveny na WebAssembly nebo do něj byly převedeny z jiného jazyka.

- RustPython[1] - interpret jazyka Python napsaný v Rustu a převeden do WebAssembly
- John Conway's Game of Life[2]
- Commander Keen[3] - port starší počítačové hry do browseru pomocí nástroje Emscripten.

WebAssembly je stále v aktivním vývoji, momentálně je podporován čtyřmi hlavními prohlížeči (Chrome, Safari, Firefox, Edge) ale seznam podporovaných funkcí se u každého liší, stejně jako výkonnost. Lze tedy očekávat, že se v budoucnosti budou možnosti využití WebAssembly ještě rozšiřovat.

2.3 Platforma .NET

Platforma .NET [27] je vyvíjena společností Microsoft. Jedná se o prostředí pro vývojáře, umožňující tvorbu aplikací v různých, převážně objektových jazycích. Platforma cílí na multiplatformní nativní podporu a to jak na desktopových zařízeních tak i mobilních.

Microsoft poskytuje mimo samotnou platformu .NET také implementace specializované podle využití nebo platformy. Například .NET Framework pro Windows aplikace nebo Xamarin pro mobilní aplikace. Tyto implementace, navzdory tomu, že cílí na různé platformy, mohou využívat společné knihovny skrze rozhraní .NET Standard, které je možné použít při vytváření vlastních knihoven. Takto vytvořené knihovny lze sdílet mezi různými implementacemi .NET.

Jedním z podporovaných jazyků je C#, ve kterém je tato diplomová práce implementována. C# je obecně využitelný, typovaný, objektově orientovaný jazyk, který cílí na maximalizaci produktivity vývojáře. V dnešní době již C# běží na všech běžných platformách, a to konkrétně na Windows, Linuxu, macOS, iOS, Androidu i přímo na webu.

C# je závislý na CLR (*Common Language Runtime*), který řeší například správu paměti, garbage collector a ošetření výjimek. Poskytuje také dva typy kompilátorů. Běžně se používá

[1] Web RustPython: https://rustpython.github.io/demo/
[2] Web Game of Life: https://playgameoflife.com/
[3] Web Commander Keen: https://www.jamesfmackenzie.com/2019/10/28/commander-keen-ported-to-webassembly/

takzvaný JIT (*Just-In-Time*) kompilační režim, který překládá mezikód na nativní až před samotným vykonáváním a zvyšuje tak přenositelnost kódu. Druhým typem je AOT (*Ahead-of-Time*) kompilátor, který dokáže přeložit mezikód předem a tím zrychlit načtení a ušetřit zdroje zařízení. K dispozici je také reflexe, která umožňuje kontrolu metadat objektů za běhu aplikace, včetně tvoření nových objektů z předem neznámých typů, což může být vhodná vlastnost právě pro tvorbu kompilátoru či interpretu. [2]

2.4 Zpracování jazyka, kompilátor a interpret

Veškerý software byl napsán v nějakém programovacím jazyce, abychom jej mohli spustit na počítači, je potřeba kód přeložit z programovacího jazyka do formy, které počítač rozumí, aby daný program mohl vykonat. Nástroje, které provádí tento překlad se nazývají kompilátory. Informace v této podkapitole jsou čerpány primárně z knihy [1].

2.4.1 Kompilátor

Jak již bylo zmíněno v úvodu, kompilátor je program, který převádí program ze zdrojového jazyka na ekvivalentní reprezentaci v cílovém jazyce. Důležitou rolí kompilátoru je také hlášení chyb ve zdrojovém jazyce, které detekuje při překladu. Proces kompilace probíhá v několika navazujících fázích.

Lexikální analýza

První fází kompilace je lexikální analýza, občas nazývaná také *scanning*. Do této fáze vstupuje kód předzpracovaný preprocesorem. Ten je načten lexikálním analyzátorem a rozdělen na lexémy, které jsou následně reprezentovány ve formě tokenů (*symbolů*). Může jít například o identifikátor proměnné, operátory, čísla. Tyto tokeny jsou použity jako vstup do další fáze překladu – syntaktické analýzy. Nalezené identifikátory proměnných, parametrů a funkcí se ukládají do tabulky symbolů. Ta udržuje informace o jejich typu, rozsahu platnosti, názvu a dalších podrobnostech. Tabulka symbolů se doplňuje i v rámci syntaktické analýzy a je pak využívána ve všech následujících fázích pro kontrolu typů nebo optimalizace.

Lexikální analýza také odstraňuje komentáře a bílé znaky. Výstupem analýzy je posloupnost tokenů.

Syntaktická analýza

Druhá fáze je syntaktická analýza, která je také nazývána *parsing*. Syntaktická analýza přijímá na vstupu tokeny z lexikální analýzy a transformuje je podle definice jazyka na stromovou strukturu. Typicky do podoby syntaktického nebo více konkrétního derivačního stromu. Strom je sestaven s ohledem na precedenci jednotlivých operací.

Sémantická analýza

Navazující fází je sémantická analýza, která přijímá na vstupu syntaktický strom z předchozí fáze a společně s tabulkou symbolů jej používá pro kontrolu zdrojového programu, zda odpovídá pravidlům zdrojového jazyka. Při analýze jednotlivých uzlů aktualizuje informace o typech identifikátorů v tabulce symbolů. Během sémantické analýzy probíhá kontrola typů. Musí se ověřit, zda každý operátor má správné operandy, pokud tomu tak není, kompilátor

nahlásí sémantickou chybu. Pokud jazyk umožňuje typovou konverzi, poté může sémantický analyzátor, místo nahlášení chyby, označit operand jako kandidáta pro typovou konverzi.

Generování mezikódu

Po sémantické analýze může kompilátor převést syntaktický strom do mezikódu. Ten by měl být dostatečně jednoduchý na vygenerování a převedení do cílového jazyka. Mezikód může být reprezentován ve formě tří-adresného kódu.

Optimalizace kódu

Následuje optimalizační fáze, která se snaží mezikód optimalizovat tak, aby výstupní kód v cílovém jazyce byl lepší. To může znamenat například kratší kód nebo rychlejší vykonávání.

Generování kódu

Poslední fází je samotný překlad mezikódu do cílového jazyka podle jeho požadavků. Výsledkem generování kódu je série instrukcí v cílovém jazyce.

2.4.2 Kompilátor vs. Interpret

Dalším druhem programu pro zpracování jazyka je interpret. Od kompilátoru se liší tím, že negeneruje kód v cílovém jazyce, ale rovnou provádí instrukce zdrojového jazyka na vstupech zadaných uživatelem. Interpret provádí shodně první tři fáze překladu jazyka jako kompilátor – lexikální, syntaktickou a sémantickou analýzu, poté však probíhá místo generování kódu samotná interpretace kódu, řádek po řádku.

Kompilátory jsou typicky mnohem výkonnější než interprety. To kvůli tomu, že interpret musí každý zdrojový příkaz převádět pokaždé, když na něj narazí. Uvádí se, že interprety jsou 10x až 100x pomalejší než kompilátory [4]. Interprety ovšem mohou poskytovat kvalitnější výstupy při ladění kódu, jelikož vykonávají kód řádek po řádku a mohou tak zobrazit chybu v originálním tvaru. U kompilátorů je kód již po optimalizaci a převodu do cílového jazyka. Dalšími výhodami oproti kompilátoru jsou nezávislost na platformě a méně času stráveného při kompilaci.

Kapitola 3

Existující řešení

V této kapitole jsou uvedeny existující řešení v době průzkumu před započetím implementace. Průzkum existujících platforem se zaměřuje na řešení umožňující kompilaci nebo interpretaci kódu online v prohlížeči. Počet unikátních řešení tohoto problému pro jazyk C je poměrně omezený, proto je tato kapitola rozdělena na dvě podkapitoly. První podkapitola uvádí existující řešení pro jazyk C 3.1. Druhá podkapitola pak poskytuje přehled řešení pro jiné jazyky 3.2, které také mohou poskytnout cenné informace o možnostech realizace.

V rámci průzkumu bylo nalezeno více řešení, než je v této kapitole uvedeno. Nebyly zde uvedeny z důvodu, že se u nich vyskytují stejné principy řešení jako u níže uvedených a jejich popis by nepřinesl žádné nové informace.

3.1 Jazyk C

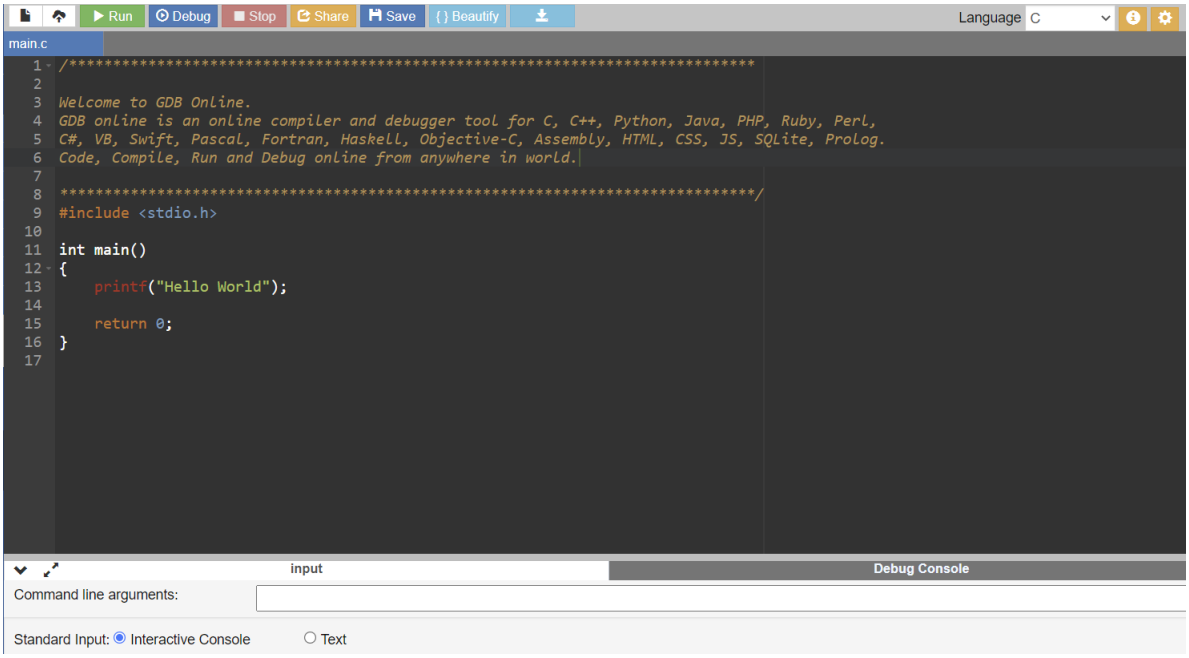
3.1.1 Kompilace na serveru - OnlineGDB, Programiz, JDoodle

OnlineGDB je online kompilátor a debugger pro jazyky C, C++ a další. Jako první online kompilátor nabízí možnost ladění skrze gdb[1] debugger [19] a kompilaci přes gcc[2].

Prostředí umožňuje vložení vstupního kód do textového pole s formátováním, zvýrazněním a našeptáváním názvů funkcí. Lze také vložit vstup ve formě argumentů příkazové řádky a textový vstup. Unikátní je využití ladícího programu gdb, ke kterému poskytuje také vizuální prostředí - krokování, sledování proměnných a nastavení breakpointů. Prostředí nabízí rozdělení kódu do více souborů. Nechybí ani možnost práce se soubory v dočasném adresáři.

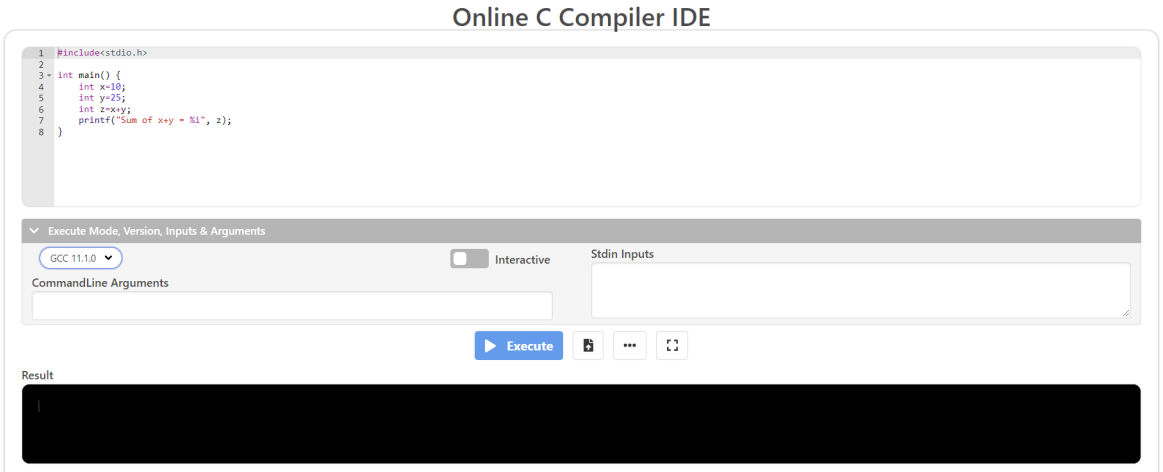
Tento nástroj je z uživatelského pohledu velmi vyspělý. Umožňuje programování v jazyce C v prohlížeči a to včetně ladění kódu. Projekt není open-source, nelze tedy přesně určit jakým způsobem je implementován. Dle analýzy chování a aktivity internetové sítě se zdá, že používá gcc a gdb na serveru, kam posílá vstupní kód ke zpracování. Pravděpodobně kvůli tomu je čas kompilace a doba provádění omezen na předdefinované maximální časy. Na první pohled se může zdát, že umožňuje také režim offline, jelikož využívá WebSockets[3], které ignorují nastavení odpojení od sítě ve vývojářských nástrojích prohlížeče, ale přesto komunikaci se serverem nutně potřebuje, ověřit si to lze odpojením počítače od internetové sítě.

[1] https://www.sourceware.org/gdb/
[2] https://gcc.gnu.org/
[3] https://developer.mozilla.org/en-US/docs/Web/API/WebSocket



Obrázek 3.1: Ukázka prostředí webového interpretu a debuggeru OnlineGDB, nahoře je místo pro psaní kódu a tlačítka na ovládání, dole poté vstupní argumenty, zobrazují se zde i výstupy programu a debugger.

Na podobném principu funguje i nástroj Programiz[4], který také využívá WebSockets. Stejný princip využívá také kompilátor od TutorialsPoint[5] a JDoodle[6], tyto nástroje požadavky posílají skrze XHR požadavky a zpracovávají je v PHP mezivrstvě, princip je ale stejný, využívají pro kompilaci gcc.



Obrázek 3.2: Nástroj JDoodle je zajímavý z pohledu GUI, má příjemné prostředí a elegantně vyřešeno předávání vstupu. Taky umožňuje volbu verze gcc.

[4] https://www.programiz.com/c-programming/online-compiler/
[5] https://www.tutorialspoint.com/compile_c_online.php
[6] https://www.jdoodle.com/c-online-compiler/

3.1.2 Kompilace a interpretace v prohlížeči

Během průzkumu byl nalezen i začínající projekt na implementaci kompilátoru jazyka C ve WebAssembly[7], který by prováděl kompilaci v rámci prohlížeče, ale ten byl v průběhu již ukončen, z podobných důvodů[8] jako jsou uvedeny při analýze v kapitole 4 u možnosti kompilace. Zároveň nebyl nalezen žádný žádný veřejně dostupný interpret jazyka C v prohlížeči.

Nejblíže se kompilaci nebo interpretaci přímo v prohlížeči blíží emulátory Linuxu psané v JavaScriptu[9], které poskytují i gcc. Použití takového emulátoru je ovšem o několik úrovní abstrakce mimo interpret nebo kompilátor a nesouvisí s technologiemi vybranými pro tuto práci. Tyto emulátory poskytují vlastní implementaci počítače s instrukční sadou x86, nad kterými je spuštěna instance Linuxu, jehož součásti se s používáním postupně stahují do prohlížeče.

3.2 Ostatní jazyky

Tato podkapitola nabízí přehled řešení, které se netýkají přímo jazyka C, ale řeší stejný problém v rámci jiných jazyků. Pro přehlednost byly vynechány řešení, které vyžadují komunikaci se serverem, jelikož taková řešení nemohou splnit zadání a byla informativně uvedena pouze pro řešení přímo související s jazykem C.

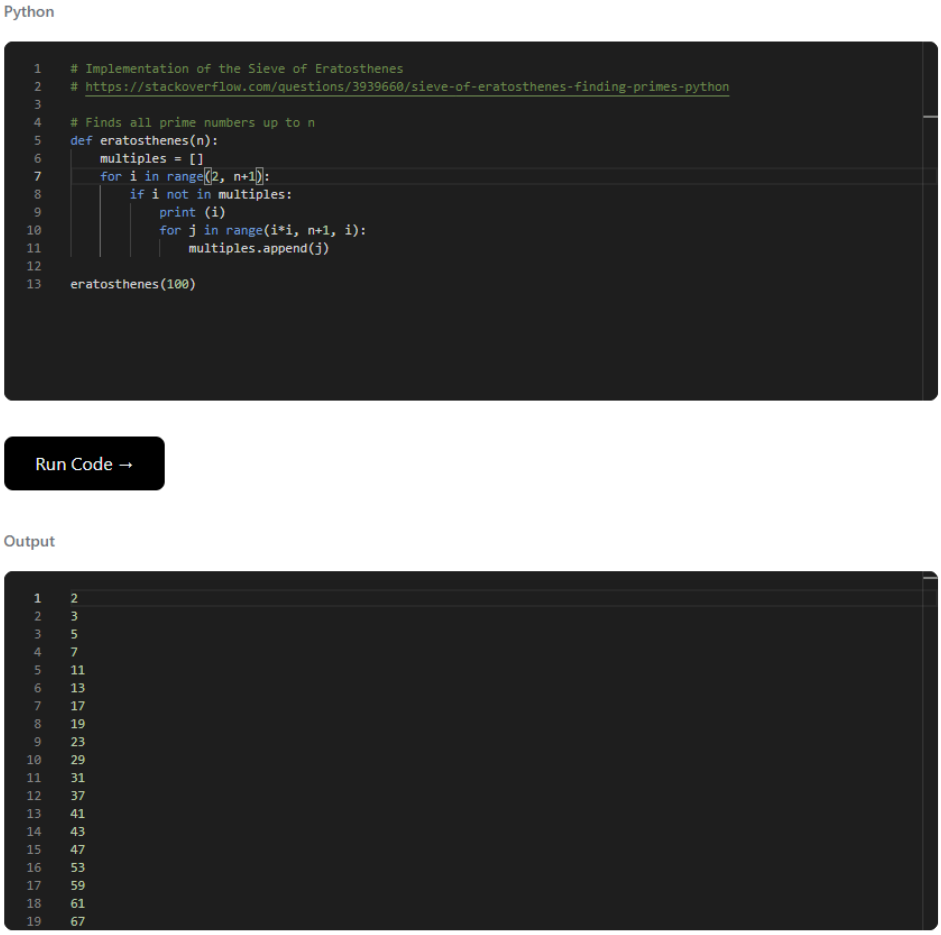
3.2.1 run-wasm

Run-wasm je nástroj pro jednoduché spouštění kódu v prohlížeči s využitím WebAssembly. Cílí na projekty, které potřebují demonstrovat ústřižky kódu na jejich webových stránkách [23].

Tento projekt nabízí prostředí pro spuštění kódu v jazyce Python a TypeScript v prohlížeči. Díky využití pyodide[10] má možnost běžet také offline. Pyodide je WebAssembly port pro CPython[11]. Tento nástroj nabízí velmi jednoduché prostředí, které se dělí na dvě části - vstupní okno pro kód a výstupní okno pro výsledek běhu programu - tyto okna využívají formátovací nástroj Monaco Editor[12] pro zvýraznění kódu.

Výhodou tohoto řešení může být jeho jednoduchost pro uživatele a také možnost offline spuštění. Aktuálně však řešení nepokrývá chybová hlášení a analýzu kódu. Pokud se v kódu vyskytne chyba, lze ji dohledat pouze v rámci vývojářských nástrojů prohlížeče. Taktéž postrádá možnost samostatné definice vstupních argumentů pro existující kód. Nejedná se zde o kompilaci kódu v pythonu do WebAssembly, vše běží přes již zkompilovaný pyodide, který je interpretem.

[7] https://dev.to/jerryhue/planning-an-online-c-compiler-for-ipc144-a6o
[8] https://dev.to/jerryhue/playing-with-webassembly-to-run-snippets-of-code-421p
[9] https://bellard.org/jslinux/
[10] https://pyodide.org/
[11] https://github.com/python/cpython
[12] https://microsoft.github.io/monaco-editor/



Obrázek 3.3: Ukázka prostředí webového interpretu run-wasm

3.2.2 Hackpad

HackPad[13] je online kompilátor jazyka Go, umožňující práci v režimu offline. Řešení se skládá ze tří částí – operačního systému, editoru a jedné nebo více konzolí. Operační systém poskytuje abstrakci nad souborovým systémem a procesy, zachytává systémová volání a nahrazuje vlastními funkcemi pro práci s virtuálním souborovým systémem. [24]

V tomto projektu lze vidět unikátní přístup využití WebAssembly pro spuštění kompilátoru jazyka GO, poskytující vše co je pro práci s jazykem potřeba. Chybí zde pouze možnost ladění. Analýzou veřejně dostupného kódu lze získat informace o tom, jak vše funguje.

- Prostředí pro interakci, psaní kódu a spouštění je napsáno v jazyce Go a doplněno obslužným JavaScriptem, který zachytává systémová volání s využitím API `syscall/js`. API poskytuje možnost komunikace mezi JavaScriptem a Go. Soubory napsané v Go jsou přeloženy do WebAssembly.
- Po načtení stránky se načte mezivrstva, automaticky se provede instalace jazyka Go, načtou se všechny potřebné WebAssembly soubory a spustí se demonstrační příkaz v jazyce Go.

[13] https://hackpad.org/

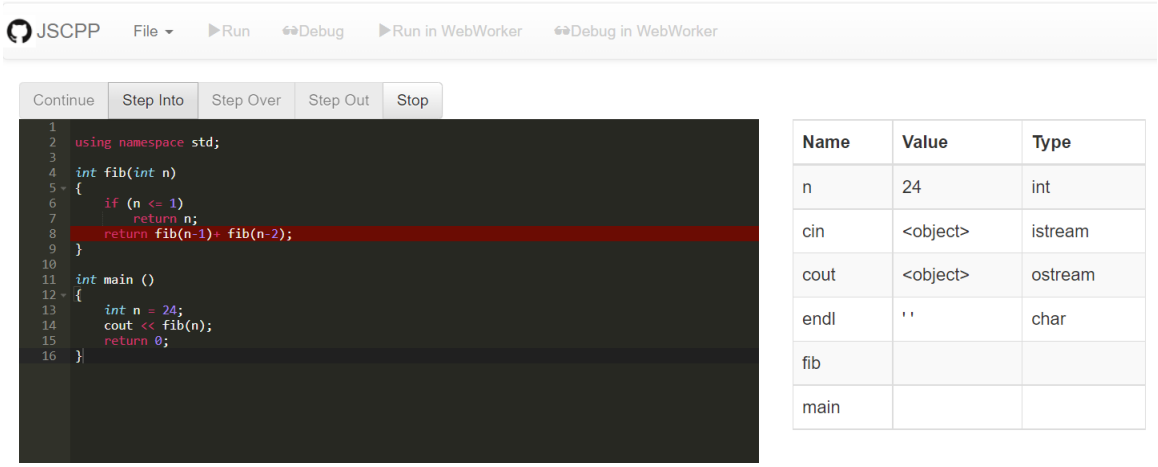
- Pro instalaci jazyka Go je využita možnost zkompilovat samotné balíčky jazyka Go do WebAssembly. To zjednodušuje implementaci, protože není potřeba emulovat kompletní operační systém, aby bylo možné spustit binární soubory.
- Pro spuštění instalace Go využívá JavaScript soubor `wasm_exec.js`, který je součástí balíčku Go. Ten umožňuje načtení WebAssembly souborů vytvořených v jazyce Go do prohlížeče. Hackpad tak pomocí tohoto skriptu načte instalaci jazyka Go, která byla předtím zkompilována do WebAssembly.
- Sestavení zdrojového kódu poté probíhá skrze instalaci jazyka Go, která vygeneruje WebAssembly soubor. Spuštění zkompilovaného kódu opět přes exec.

Po analýze lze vidět, že HackPad intenzivně využívá nativní schopnost kompilace jazyka Go do WebAssembly.

3.2.3 RustPython a JSCPP

RustPython[14] je interpret jazyka Python napsaný v jazyce Rust, který je možné zkompilovat do WebAssembly. Možnost kompilace Rustu do WebAssembly umožňuje běh interpretu v režimu offline. RustPython obsahuje okno pro zadání vstupního kódu v Pythonu, výstup ze syntaktické a sémantické analýzy a okno pro zobrazení výsledků. Oproti ostatním řešením obsahuje také *interactive shell*, který umožňuje okamžité vyhodnocení kódu. Neobsahuje možnost ladění kódu ani možnost zadání vstupů. Obsahuje ale mezivrstvu pro komunikaci s prohlížečem z Pythonu.

JSCPP[15] je také interpret, tentokrát však jazyka C++. Je založen na podobném principu jako RustPython, ovšem kompletně napsán v jazyku JavaScript bez využití WebAssembly. Nepodporuje všechny konstrukce ani funkce jazyka C++, ale oproti ostatním řešením poskytuje vizuální ladění kódu s možností krokování, viz. obrázek 3.4.



Obrázek 3.4: Ukázka prostředí webového interpretu JSCPP s možností vizuálního ladění.

[14] https://rustpython.github.io/demo/
[15] https://felixhao28.github.io/JSCPP/

3.2.4 wasm-clang

Posledním analyzovaným řešením je projekt wasm-clang[16]. Tento projekt je postaven na stejném principu jako HackPad 3.2.2. Využívá Clang[17], který umožňuje kompilaci do WebAssembly a zároveň je možné samotný clang a linker zkompilovat do WebAssembly. K tomu aby vše fungovalo je nutná opět JavaScriptová abstrakce souborového systému a systémových volání. Podpora standardních knihoven je zajištěna komprimovaným balíčkem, který obsahuje všechny standardní knihovny jazyka C++. Ty se rozbalí do virtuálního souborového systému a přilinkují při kompilaci.

Díky využití tohoto principu je wasm-clang schopný kompilovat soubory jazyka C++ i offline v prohlížeči. Kompilace vstupního kódu probíhá opět do WebAssembly, které se spouští přes JavaScriptovou mezivrstvu.

Prostředí neposkytuje možnost zadání vstupu nebo načtení souborů, pouze jednoduchý editor pro zadávání vstupního kódu a terminál zobrazující probíhající příkazy a jejich výstupy. Oproti ostatním řešením poskytuje i možnost zobrazení grafického výstupu s využitím canvasu. Ladění kódu tímto řešením není podporováno a pro jeho zprovoznění by bylo potřeba do WebAssembly vytvořit i port gdb. To se zatím zřejmě nikomu nepodařilo.

Vzhledem k tomu, že clang podporuje i kompilaci jazyka C, mělo by být možné toto řešení upravit, aby podporovalo kompilaci a spouštění programů v jazyce C.

3.3 Zhodnocení aktuálního stavu

Průzkum aktuálně existujících řešení poskytl čtenáři přehled více různých přístupů k programování ve webovém prohlížeči. Pro jazyk C nebyl nalezen žádný vhodný webový nástroj, který by dokázal splnit požadavek ze zadání na umožnění práce po odpojení se od sítě, jelikož všechny nalezené řešení pracují s využitím serverové aplikace ve které probíhá kompilace. Nejblíže řešení jsou emulátory operačních systémů, ty ale nejsou zaměřeny na programování v prohlížeči a možnost kompilace kódu v nich je spíše vedlejším účinkem.

V rámci ostatních programovacích jazyků se programy blíží požadovanému řešení o poznání více. Přes některé nedostatky umožňují práci offline a většina určitým způsobem využívá WebAssembly, žádná však v kombinaci s platformou .NET.

Tato práce se snaží poskytnout možnost kompilace zdrojového jazyka ideálně přímo do WebAssembly a nebo interpretace skrze program ve WebAssembly. Pro jazyk C nebyl nalezen takový kompilátor ani interpret. Pro ostatní jazyky splňují požadavek kompilace do WebAssembly nástroje HackPad 3.2.2 a wasm-clang 3.2.4, které poskytují kompilátor s možností kompilace do WebAssembly. Zbytek uvedených platforem pak odpovídá druhé cestě, poskytují interpret generovaný do WebAssembly. Interpret JSCPP 3.2.3 umožňuje dokonce i vizuální ladění. RustPython pak nabízí možnost okamžitého vyhodnocení příkazu – to jsou další body zadání, které výše uvedené kompilátory nesplňují.

[16] https://github.com/binji/wasm-clang/tree/master
[17] https://clang.llvm.org/

Kapitola 4

Kombinace .NET a WebAssembly

Tato kapitola poskytuje čtenáři vhled do samotného jádra této práce. Uvádí možnosti kombinace platformy .NET a WebAssembly takovým způsobem, aby mohl vzniknout plnohodnotný nástroj pro zpracování kódu jazyka C s možností práce bez internetového připojení, možností zadat příkaz pro okamžité vyhodnocení a také rozšiřitelností o případné vizuální ladění.

Kapitola se zaměřuje na dva hlavní směry možné implementace. Prvním směrem 4.1 je přímá kompilace kódu v prohlížeči do WebAssembly a následné spuštění v součinnosti s platformou .NET. Druhým směrem 4.2 je vytvoření interpretu na platformě .NET, který by byl následně zkompilován do WebAssembly a tím by poskytl možnost práce offline. Jsou zde uvedeny detaily jednotlivých přístupů a nástrojů pro jejich zavedení. Následně jsou tyto přístupy zhodnoceny dle jejich přínosů a rizik 4.3.

4.1 Kompilace vstupního kódu do WebAssembly

První přístup se zaměřuje na kompilaci kódu ve zdrojovém jazyce do WebAssembly. Cílem této varianty je umožnit uživateli pracovat offline v prohlížeči při zachování rychlosti blížící se nativnímu kódu.

Pro kompilaci jazyka C do WebAssembly skrze C#/.NET neexistuje nástroj. Tato podkapitola uvádí možné alternativní varianty, které by mohly být využity pro tento účel.

4.1.1 Kompilace jazyka C existujícím nástrojem

První zvažovanou variantou, která vyplynula z analýzy v předchozí kapitole 3, je možnost přímého převodu kódu v jazyce C do WebAssembly s využitím existujících nástrojů.

Emscripten, Clang

Emscripten poskytuje kompletní sadu nástrojů pro kompilaci do WebAssembly s využitím LLVM. Klade důraz na rychlost, výslednou velikost a webovou platformu. Dokáže zkompilovat jazyk C nebo libovolný jiný jazyk používající LLVM do WebAssembly a nabízí také spoustu API rozhraní. Vygenerovaný kód v základu obsahuje JavaScriptovou mezivrstvu, potřebnou pro spuštění WebAssembly v prohlížeči. [8]

Tento nástroj je již odladěný, oficiálně podporovaný a doporučovaný pro kompilaci do WebAssembly. K vygenerovanému WebAssembly poskytuje JavaScript, který je potřeba ke spuštění kódu a komunikaci mezi WebAssembly aplikací a prohlížečem. Například předávání nepodporovaných datových typů do WebAssembly je potřeba řešit JavaScriptem. Kompilátor

dokáže převést téměř každý existující program, jsou ale nutné drobné úpravy, například abstrakce práce se souborovým systémem podle standardů prohlížeče. O většinu se ale postará samotný Emscripten.

Emscripten využívá k překladu interně Clang, o kterém již byla zmínka v předchozí kapitole 3.2.4. Využití Clang bez Emscripten je možné, vygenerované soubory ale neobsahují mezivrstvu pro spuštění v prohlížeči. Pokud by to bylo cílem, je nutné tyto soubory doplnit vlastnoručně.

Zhodnocení

Tyto dva nástroje lze zcela jistě využít pro kompilaci kódu z jazyka C do WebAssembly způsobem, který zajišťuje rychlost podobnou nativnímu řešení. Velkou výhodou je, že po zprovoznění nástrojů by byl zbytek implementace téměř bez práce. Kompilátor by si s konstrukcemi a funkcemi jazyka C poradil po nalinkování standardních knihoven sám.

V případě Emscripten by bylo možné vytvořit provázání s platformou .NET skrze vyvolání procesu na pozadí, to ale vyžaduje serverovou aplikaci a tudíž nutnost připojení k internetové síti. Zkompilovat Emscripten do WebAssembly nelze, nebo je to příliš náročné.

Pro Clang nám nutnost připojení k internetové síti odpadá, protože je možné samotný Clang zkompilovat do WebAssembly a při dodání správné abstrakce operačního systému můžeme s jeho využitím kompilovat přímo v prohlížeči.

Tento přístup by v důsledku znamenal nevyužití platformy .NET, jelikož její přínos by pro takové řešení byl nulový. Architektura by byla podobná zmíněnému projektu wasm-clang 3.2.4 – pro kompilaci by se využil existující nástroj, ke kterému by byla vytvořena abstrakce souborového systému a dalších systémových volání – pro tyto účely je ale potřeba použít JavaScript a platforma .NET by tady neměla své místo.

Kromě toho, že by takové řešení bylo mimo zaměření této práce, splnění dalších bodů zadání se zdá být velmi obtížné. Konkrétně možnost rozšíření o ladění se z analýzy zdá být příliš složitá. Běžně se pro ladění kódu jazyka C používá GDB, které nemá port do WebAssembly, ačkoli byly nalezeny nějaké pokusy[1]. Je však pravděpodobné, že se v budoucnosti nástroje na ladění kódu objeví.

Tento přístup lze obecně doporučit pokud chceme implementovat kompilátor jazyka C v prohlížeči mimo platformu .NET, ale pro účely použití ve výuce s důrazem na možnost ladění, možnost případné definice nápomocných výjimek a dalších rozšíření, se zdá být bez vhodných nástrojů prozatím nevhodný.

4.1.2 Manuální vygenerování WebAssembly s využitím jazyka C#

Další zvažovanou variantou, která by opravdu využila platformu .NET, je generování kódu ve formě instrukcí WebAssembly.

Pro tento účel existují dva podobné nástroje, *dotnet-webassembly* 4.1.2 a *cs-wasm* 4.1.2.

dotnet-webassembly

Knihovna *dotnet-webassembly* je v aktivním vývoji, umožňuje načítat, modifikovat a spouštět existující WebAssembly soubory na platformě .NET a taktéž i vytvářet nové WebAssembly soubory. Nedokáže převádět kód jazyka C# do WebAssembly, ale umožňuje skládat jednotlivé instrukce, které odpovídají standardu WebAssembly a ty následně vygenerovat do

[1] https://blog.wokwi.com/running-gdb-in-the-browser/

souboru. Knihovna až na několik příkladů zcela postrádá dokumentaci, ale její funkčnost je dokumentována alespoň desítkami testů nad jednotlivými instrukcemi. [12]

Nástroj využívá strukturu WebAssembly modulu odpovídající oficiální specifikaci[2]. Jednotlivé části jsou pojmenovány dle dokumentace WebAssembly. To zlepšuje orientaci mezi tímto nástrojem a dokumentací WebAssembly. Kód má svá pravidla, například funkce musí být ukončeny instrukcí End(), tyto pravidla mohou být vyčteny z příkladů a testů, opět odpovídají definici WebAssembly.

Vytvořené moduly a jejich funkce lze v rámci C# převést do WebAssembly a následně uložit do souboru nebo spustit přímo bez nutnosti ukládání na disk. V případě vygenerování na disk je potřeba dodat spouštěcí JavaScript pro WebAssembly a nebo soubor opět načíst skrze dotnet-webassembly. Ten si jej interně převede opět na reprezentaci v jazyce C# a zkompiluje skrze JIT.

```
var module = new Module();
module.Types.Add(new WebAssemblyType
{
    Parameters = new[] { WebAssemblyValueType.Int32 },
    Returns = new[] { WebAssemblyValueType.Int32 },
});
module.Functions.Add(new Function { Type = 0 });
module.Codes.Add(new FunctionBody
{
    Code = new Instruction[] { new LocalGet(0), new Int32CountOneBits(), new End() },
});
module.Exports.Add(new Export
{
    Kind = ExternalKind.Function,
    Index = 0,
    Name = "Demo",
});
var instanceCreator = module.Compile<dynamic>();
using (var instance = instanceCreator(new ImportDictionary()))
    Console.WriteLine(instance.Exports.Demo(0));
```

Výpis 4.1: Ukázka kompilace do WebAssembly a následné interpretace s využitím dotnet-webassembly. Na ukázce lze vidět logické dělení kolekcí podle specifikace WebAssembly a kompilaci kódu s následnou interpretací.

cs-wasm

Knihovna *cs-wasm* je alternativou ke knihovně *dotnet-webassembly* 4.1.2. U této knihovny byl aktivní vývoj již ukončen a nelze tedy počítat s tím, že bude reflektovat aktuální specifikaci WebAssembly.

Funkcionalita této knihovny je velmi podobná, dokáže číst, zapisovat, interpretovat a navíc i optimalizovat WebAssembly soubory, poskytuje také možnost generování kódu z textové formy WebAssembly [7].

Nevýhodou oproti **dotnet-webassembly** je struktura kódu, která u této knihovny není příliš podobná struktuře WebAssembly. Používá například odlišné pojmenování jednotlivých částí modulu, sekce jsou spojeny do jednoho pole a rozlišují se pouze dle typu. Při použití této knihovny se nelze v kódu orientovat pouze podle definice WebAssembly, ale je potřeba také důkladně nastudovat implementaci této knihovny, dokumentace chybí.

[2] https://webassembly.github.io/spec/core/syntax/modules.html#syntax-module

```
var file = new WasmFile();
var typeSection = new TypeSection();
file.Sections.Add(typeSection);
var memorySection = new MemorySection();
file.Sections.Add(memorySection);
memSection.Memories.Add(new MemoryType(new ResizableLimits(1, 1)));
```

Výpis 4.2: Ukázka vytvoření modulu s typem a pamětí, lze vidět odlišné pojmenování a přístup k práci s jednotlivými sekcemi, které jsou shlukovány do pole Sections.

Zhodnocení nástrojů pro generování kódu

Tyto nástroje jako jediné nalezené nabízí možnost provést kompilaci na klientské straně a je možné je využít v režimu offline. Pro spuštění kompilovaného kódu na klientské straně je z hlediska uživatele vhodné WebAssembly soubory skrze tyto nástroje interpretovat na platformě .NET. To ovšem znamená ztrátu hlavní výhody oproti druhému možnému přístupu 4.2. V opačném případě je nutné ke zkompilovaným souborům dodat obslužný HTML, JavaScript a připravit pomocný kód pro převod vstupních parametrů a další mezivrstvy. Tyto soubory by uživatel spouštěl na svém lokálním serveru, nelze je spustit automaticky ze strany aplikace, jelikož JavaScript podporuje nahrání a spuštění WebAssembly souboru pouze ze sítě. Pro odstranění tohoto nedostatku by bylo potřeba vytvořit abstrakci nad souborovým systémem v prohlížeči.

Oproti předchozím nástrojům by zde bylo velké množství práce, veškerý kód jazyka C by musel být převeden na instrukce ve formátu WebAssembly a to nejen funkce ale i kompletní práci s pamětí, ukazatele a podobně. To může být velmi náročný úkol, vzhledem k tomu, že WebAssembly prozatím nemá k dispozici dostatečně silné ladicí nástroje.

4.1.3 Celkové zhodnocení přístupu

Generování kódu do WebAssembly přináší výhodu v podobě vysoké efektivity výstupního kódu, uživatel by dostal svůj kód zkompilovaný do WebAssembly a mohl by jej spustit v prohlížeči bez pomalé interpretace.

Využitelné pro tento účel za splnění hlavního bodu zadání – možnosti práce v režimu offline a realizace na platformě .NET – jsou pouze nástroje *cs-wasm* a *dotnet-webassembly*, z nichž preferovaná varianta je *dotnet-webassembly*, kvůli své kompatibilitě se standardem WebAssembly.

Tento přístup byl v rámci analýzy i testován a bylo identifikováno několik nevýhod a implementačně náročných detailů:

- Nelze efektivně řešit hlášení chyb za běhu, ladění kódu a případné krokování.
  - Chrome DevTools (vývojářské nástroje prohlížeče) již nějakou dobu podporují ladění WebAssembly s využitím formátu DWARF, který poskytují běžné kompilátory jazyka C. Například nástroj emscripten s vlajkou -g vygeneruje tyto ladicí informace.
  - S využitím těchto informací je možné v prohlížeči krokovat kód v původní podobě, avšak v našem případě je tato varianta nevyhovující, neboť koncový uživatel by byl nucen si kód krokovat přímo v DevTools, nelze je provázat JavaScriptem nebo C#.

  - Ani jeden nástroj splňující zadání navíc neposkytuje generování ladících informací z původního kódu v jazyce C a nebylo by tedy možné uživateli nahlásit původ chyby jinak, než ve formě instrukcí zkompilovaného WebAssembly. I ty by byly zobrazeny pouze v rámci DevTools, nelze k nim z Javascriptu nebo C# přistoupit z důvodu zabezpečení prohlížeče.

- Vygenerovaný kód je z uživatelského pohledu vhodnější spustit v .NET, tím by ale ztratil na efektivitě.

  - Kód vygenerovaný přímo do WebAssembly souboru lze i v offline režimu stáhnout do počítače, ovšem k jeho spuštění je potřeba využít JavaScriptovou mezivrstvu, která se stará o jeho obsluhu a mít soubor uložený na serveru. To by pro uživatele znamenalo stáhnout vygenerované WebAssembly, HTML a JavaScript a ten následně hostovat na serveru, jelikož Fetch API umožňuje načítání pouze ze sítě, viz. [14] a [9].
  - Z toho vyplývá, že by z uživatelského hlediska bylo lepší kód následně interpretovat skrze .NET, vybrané nástroje to umožňují, ovšem poté postrádá kompilace do WebAssembly význam, neboť tím nejen ztratíme některé možnosti, které bychom měli při použití přístupu 4.2, ale zvýšíme i čas strávený na kompilaci a následné interpretaci uživatelského kódu. Nástroj *dotnet-webassembly* však kód převádí zpátky do jazyka C# a provádí JIT kompilaci, zde lze očekávat stále lepší výkon než u interpretu.
  - Nedostatek Fetch API lze teoreticky odstranit vlastní implementací souborového systému.

- Pro plnou podporu datových typů je třeba kooperovat s JavaScriptem a vytvořit pomocné funkce pro konverze do číselné reprezentace.

  - WebAssembly podporuje pouze číselné datové typy (int a float), pro plnou podporu datových typů z jazyka C je nutné napsat převodník datového typu na číselnou podobu ve WebAssembly. Navíc je tyto konverze nutné poskytnout i pro předání vstupních parametrů skrze JavaScript. K předání komplexních objektů přes JavaScript lze využít referenční typ ve WebAssembly[3]. Například Emscripten tyto převodníky obsahuje[4].

- Použitelné nástroje neobsahují pokročilé optimalizace kódu.

  - Emscripten využívá Binaryen[5], který dokáže provést optimalizace na zkompilovaném WebAssembly kódu. Kvůli offline režimu jej nemůžeme použít na serveru, ale existuje port, který lze použít i z JavaScriptu. To by znamenalo vytvořit komunikační vrstvu mezi JavaScriptem a platformou .NET, která by umožnila před vygenerováním WebAssembly kódu odeslat binární formát do Binaryen a až následně stáhnout všechny potřebné soubory.

[3] https://webassembly.github.io/spec/core/syntax/types.html

[4] https://emscripten.org/docs/api_reference/preamble.js.html#conversion-functions-strings-pointers-and-arrays

[5] https://github.com/WebAssembly/binaryen

4.2 Interpretace vstupního kódu s využitím WebAssembly

Druhým přístupem je interpretace vstupního kódu skrze interpret napsaný v jazyce C#. Tento interpret by byl zkompilovaný do WebAssembly a tím umožnil práci v režimu offline. Podkapitola se soustředí především na framework Blazor WebAssembly, který je součástí platformy .NET.

V rámci analýzy byly prozkoumány i alternativní varianty, například generování kódu do JavaScriptu skrze projekt *JSIL*[6] nebo převod existujícího interpretu *picoc*[7] do WebAssembly. Vzhledem k jejich nepoužitelnosti na klientské straně nebo nesplnění dalších podmínek zadání byly však z textu vynechány, žádné revoluční myšlenky, které by již nebyly součástí předchozí analýzy z nich nevyplývaly.

4.2.1 Blazor WebAssembly

Následující informace pochází převážně z [3].

Blazor WebAssembly je multiplatformní framework postavený na frameworku .NET Core, který umožňuje spouštět kód jazyka C# v prohlížeči na klientské straně. Lze jej použít pro nahrazení JavaScriptu jazykem C# s možným využitím knihoven z plaftformy .NET, ale dokáže také komunikovat oboustranně s JavaScriptem (*JS Interoperability*). V .NET 5 není kód vytvořený v Blazoru kompilovaný přímo do WebAssembly, ale je interpretovaný skrze .NET runtime, který běží ve WebAssembly. Jednotlivé knihovny a soubory jazyka C# jsou ve formě DLL obsluhovány tímto runtime prostředím. Pro zvýšení výkonu je kód před publikováním optimalizován, knihovny jsou v prohlížeči uloženy v paměti cache.

V .NET 6.1 vyšla aktualizace, která umožňuje místo interpretace skrze Blazor WebAssembly runtime také Ahead-of-Time[8] kompilaci kódu do WebAssembly. Všechny C# knihovny jsou kompilovány a ačkoliv trvá publikování delší dobu a také výsledná velikost je větší, tak program dosahuje vyššího výkonu. Většina funkcionalit již není interpretována ale spouštěna nativně ve WebAssembly. Tato možnost se vyplatí zvláště, předpokládáme-li využití pro náročné výpočty. Lze dosáhnout i 16x rychlejšího běhu než při využití interpretovaného Blazoru [25].

Další novinkou je možnost použití nativního kódu z C, C++, který je následně zkompilován pomocí Emscripten v rámci *build-tools* pro Blazor WebAssembly. To umožňuje interpretu napsanému v C# využívat nativní funkce z jazyka C skrze P\Invoke a tedy některé funkce, které interpret podporuje, je možné dodat přímo v jazyce C. Tato možnost by ale nemohla sloužit ke kompilaci kódu, který dodá uživatel, jelikož kompilace probíhá v rámci publikování aplikace. Lze ji tedy využít jen pro součásti samotného interpretu.

Využití nativního kódu, případně rozsah jeho využití je nutno zvážit v rámci realizace, jelikož zde vznikají rizika v případě jeho využití pro alokaci paměti a práci s ukazateli. Vysokoúrovňový programovací jazyk C# totiž nepodporuje garbage collector nad pamětí, která je nízkoúrovňovým jazykem alokována a nemůže ji tedy ani uvolnit. Je možné ji uvolnit ručně, ale to pouze v případě, že známe ukazatele, jejichž udržování v případě zpracování uživatelského vstupu může být složité [5]. Využití nativních souborů je tedy na zvážení při návrhu implementace, bude důležité vzít v potaz kompletní architekturu aplikace a zvážit, zda takový prvek do struktury zapadá.

[6] http://jsil.org/

[7] https://gitlab.com/zsaleeba/picoc

[8] https://docs.microsoft.com/en-us/aspnet/core/blazor/host-and-deploy/webassembly?view=aspnetcore-6.0#ahead-of-time-aot-compilation

Blazor má také alternativní verzi **Blazor Server**[9], která poskytuje několik výhod oproti WebAssembly. Jedná se například o menší velikost výsledné aplikace pro klienta, výkon a plné využití možností .NET Frameworku. Nevýhodou ovšem je nemožnost použití offline a není tedy dále zvažován.

4.2.2 Zhodnocení interpretace

Vytvoření interpretu skrze Blazor WebAssembly umožňuje běh v režimu offline a jedná se o implementaci v rámci frameworku .NET. Vytvoření interpretu v objektově orientovaném jazyce C# také umožňuje snadnou rozšiřitelnost a to jak o možnost okamžitého vyhodnocení příkazu, tak i o případnou možnost ladění kódu. Výhodou jsou oproti přímé kompilaci do WebAssembly také možnosti zobrazení plných chybových hlášek s výpisem z trasování zásobníku a podpora všech základních datových typů jazyka C.

Na rozdíl od předchozího přístupu je vstupní kód interpretován mezivrstvou v C#, je tedy předpoklad, že bude výrazně pomalejší, než kompilovaný kód. S příchodem .NET 6 je možné převést celý interpret do WebAssembly a tím navýšit efektivitu vykonávání oproti dosavadní verzi Blazoru na .NET 5, ale stále nelze očekávat srovnatelnou rychlost s kompilátorem. Navíc Blazor nemusí podporovat kompilaci všech funkcí do WebAssembly a může se tak vracet do interpretovaného režimu.

Při zvolení tohoto přístupu budou dodrženy všechny body zadání včetně možnosti offline režimu. Rizikem je možnost prodloužení doby vývoje vzhledem k tomu, že ladění kódu na klientské straně není plně podporováno existujícími .NET nástroji – toto riziko lze snížit až eliminovat zavedením více prostředí pro vývoj, například konzolová aplikace pro vývoj a Blazor pro výslednou aplikaci. Je potřeba ale ověřovat, zda mají obě prostředí shodné možnosti.

4.3 Celkové zhodnocení nástrojů

Po analýze se jeví jako nejvhodnější možnosti využití kompilace skrze *dotnet-webassembly* 4.1.2 a nebo interpretace skrze aplikaci běžící na *Blazor WebAssembly* 4.2.1.

Výhodou kompilace je nesporně čas provádění kódu, lze očekávat desetkrát až stokrát vyšší výkon než u interpretované verze. Nevýhodou ovšem zůstává nejasná budoucnost ladění kódu a nic neříkající chybové hlášky po kompilaci do WebAssembly. To může být v případě využití pro začínající programátory značná nevýhoda. Další nevýhodou je vysoká náročnost převodu jazyka C do instrukcí ve formátu WebAssembly tímto způsobem, lze očekávat náročnou a dlouhou práci s nezaručeným výsledkem.

U interpretu lze očekávat násobně nižší výkon než u kompilátoru, ale oproti kompilátoru dokáže poskytnout kvalitní chybové hlášení. Principiálně taky umožňuje rozvoj o vizuální, krokové ladění, kdy můžeme uživateli zobrazit stav při každém kroku. I zde lze očekávat náročnou a dlouhou práci, bude potřeba implementovat kompletní model paměti, souborového systému, ukazatele, potřebné funkce a podobně. Zde je ovšem výhodou možnost plného využití objektově orientovaných principů, které celou implementaci zpřehlední. Na rozdíl od WebAssembly, kde by bylo potřeba programovat přímo na úrovni instrukcí.

[9] https://docs.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-6.0

Kapitola 5

Návrh řešení

Tato kapitola se věnuje návrhu implementace s ohledem na informace posbírané a zdokumentované ve fázi průzkumu. Jako první popisuje zvolený přístup k implementaci 5.1, poté pokračuje popisem návrhu samotné architektury 5.2 a následuje podrobný popis modelu paměti včetně ukazatelů, předávání argumentů a dalších podrobností 5.3.

V dalších podkapitolách se věnuje návrhu rozsahu implementace s ohledem na předmět IZP 5.4, možnosti okamžitého vyhodnocení příkazu 5.5 a modulárnosti řešení 5.6. Poslední podkapitola navrhuje možný vzhled uživatelského rozhraní 5.7.

Cílem je navrhnout nástroj, který by byl použitelný pro začínající programátory, prakticky tedy pokrývající kurz IZP.

5.1 Zvolený přístup

Po důkladném zvážení výhod a nevýhod jednotlivých přístupů zmíněných v kapitole 4, byl zvolen přístup druhý 4.2 a tedy interpretace kódu jazyka C v prohlížeči. Pro implementaci navrhuji využití frameworku Blazor WebAssembly 4.2.1. Implementace bude podporovat verzi .NET 6, to ji činí přenositelnou mezi jednotlivými operačními systémy a umožňuje AOT kompilaci. Vývoj bude probíhat na operačním systému Windows v IDE Visual Studio 2019.

Tento přístup maximalizuje užitek pro uživatele webového rozhraní a také umožní snadnou rozšiřitelnost v případě budoucího rozvoje. V kombinaci s výslednou kompilací interpretu do WebAssembly je očekávána také dostatečná rychlost pro běžné použití a samozřejmostí je kompilace offline. Nelze však očekávat stejnou výkonnost jako v případě kompilátoru do WebAssembly, výhody zde však z mého pohledu převažují.

5.2 Návrh architektury

Základním kamenem bude projekt typu Blazor WebAssembly využívající .NET 6. Tento projekt bude obsahovat grafické uživatelské rozhraní, které bude zpracovávat vstupy a posílat je do jádra interpretu. Vzniklé výstupy zobrazí v grafickém rozhraní. Vzhledem k tomu, že ladicí nástroje nejsou v rámci Blazoru plně podporovány, bude řešení obsahovat ještě jeden projekt poskytující stejnou funkcionalitu, napsaný ve WPF[1], který bude sloužit pouze pro účely ladění. V rámci WPF je ladění spolehlivé a lze tak očekávat úsporu času.

Interpret bude složený ze tří hlavních částí:

[1] https://docs.microsoft.com/cs-cz/visualstudio/designers/getting-started-with-wpf

- Lexikální a syntaktická analýza 2.4.1 - Analýza bude probíhat pomocí parseru ANTLR 6.3.2, který je na tento účel vhodný. Pro správnou funkci potřebuje definovanou gramatiku jazyka C. Jednu takovou lze získat přímo z github repozitáře nástroje ANTLR. Je pravděpodobné, že gramatika nebude kompletní a bude potřeba ji upravit.

- Sémantická analýza a generování mezikódu 2.4.1 - Sémantická analýza bude využívat nástroj ANTLR, který před-generuje takzvané *visitors*. To jsou třídy umožňující průchod výstupním stromem ze syntaktické analýzy. Pro každý uzel lze nadefinovat akci, která bude v případě navštívení uzlu provedena. S využitím této možnosti bude nadefinována sada akcí, během kterých se bude kontrolovat sémantická validita vstupu. Také bude generovat mezikód do paměti ve formě definice funkcí a jednotlivých řádků kódu. Takto získané informace se využijí při běhu programu.

- Interpretace - Interpretace bude probíhat nad zpracovaným kódem, který bude uložen ve strukturovaném objektu v paměti. Pro interpretaci bude existovat obslužná třída, která bude sloužit ke spuštění programu ve správném vstupním bodě, udržování lokálního kontextu a provádění příkazů. Před interpretací se načtou argumenty příkazové řádky, další vstupy se budou načítat v průběhu. Všechny příkazy, funkce a výrazy budou reprezentovány svou vlastní třídou, určující jejich chování a možné akce nad nimi.

Kód bude implementován s důrazem na zobecnění tříd a využití návrhových vzorů. Pro příkazy jazyka C bude využit návrhový vzor Command, čímž se zajistí jednoduchá rozšiřitelnost a zobecnění těchto příkazů. U obslužných tříd bude ve vhodných případech použita injektáž závislostí a obecně využita možnost dědičnosti a rozhraní.

5.3 Model paměti

Struktura paměti v programech jazyka C je popsána v 2.1.1. Cílem interpretu je model paměti přiblížit původní struktuře. To umožní lépe simulovat chování paměti při alokaci, dealokaci a dalších operacích. Pro tyto účely jsem zvolil následující reprezentaci jednotlivých částí paměti.

- Kódový segment - Kódový segment bude sloužit pro uchovávání funkcí a jejich jednotlivých příkazů a výrazů. Bude reprezentován vlastní třídou odvozenou ze slovníku klíč-hodnota. Klíčem je název funkce a hodnota je objekt reprezentující funkci, který uchovává její specifikaci (návratový typ, parametry, název, ...) a jednotlivé příkazy a výrazy - ty budou uloženy ve struktuře typu fronta, podle způsobu přístupu k jejich zpracování. Kód mimo funkce bude reprezentován také objektem funkce s prázdným identifikátorem.

- Inicializované data a neinicializované data - Budou reprezentovány pomocí vlastní třídy obsahující inicializované i neinicializované globální, statické i konstantní proměnné. Rozlišení uvedených dvou sekcí bude probíhat na úrovni hodnoty, která určuje jejich inicializaci, typ i rámec (globální nebo vztažen k funkci). V implementačním pohledu se zde opět bude jednat o třídu odvozenou od slovníku klíč-hodnota. Klíč bude název a hodnota bude reprezentovat konkrétní proměnnou.

- Zásobník - Bude obsahovat lokální proměnné v rámci funkce. Pro reprezentaci zásobníku bude vytvořena třída, která je odvozena stejně jako datová sekce na slovníku

klíč-hodnota, ovšem s tím rozdílem, že kvůli udržování rámce aktuální funkce v řetězených a rekurzivních voláních bude tento slovník zanořen v generické struktuře jazyka C# Stack[2] z knihovny *System.Collections.Generic*. Pro snadné udržování této struktury budou připraveny obslužné funkce na ovládání aktuálního rámce. V souvislosti s touto strukturou může být implementována také simulace chyby Stack Overflow při dosažení definovaného počtu proměnných na zásobníku. V případě definování stejnojmenné proměnné na zásobníku jako v datové sekci (globální proměnná), má přednost lokální proměnné, až poté se bude hledat v rodičovském rámci, případně v datové sekci. V rodičovském rámci se hledá jen v případě zanořených bloků.

- Hromada - Bude sloužit pro dynamickou alokaci proměnných, ke kterým se bude přistupovat přes ukazatele. Reprezentována bude vlastní třídou odvozenou od generické struktury jazyka C# LinkedList[3] z knihovny System.Collections.Generic neboli dvousměrně vázaný lineární seznam s vlastním způsobem hledání dat na odpovídající adrese podle indexu a velikosti prvku v seznamu. To umožní simulovat pointerovou aritmetiku. Tato třída bude obsluhovat zároveň i dynamickou alokaci podle definice funkcí malloc, calloc, realloc a free. Volány budou jako samostatné příkazy ve zdrojovém kódu. Pokud se hodnota na hromadě nenajde, navrácen bude ukazatel na NULL, který bude definovaný jako obecná konstanta. S touto strukturou může být také implementována simulace Heap Overflow po přesažení definované velikosti hromady, Memory Leak při neuvolnění paměti pomocí free, občasné selhání mallocu (a tedy vynucení kontroly ukazatele na NULL) i Segmentation Fault v případě pokusu o uvolnění paměti pomocí free na pozměněném ukazateli.

V souvislosti s pamětí a proměnnými je potřeba zmínit několik dalších částí:

- Simulace null-terminated textových literálů – některé funkce v jazyce C dle [10] vyžadují nulou ukončené znakové pole. Jak napovídá název, jedná se o pole znaků, které jsou ukončeny bajtem s nulovou hodnotou [18]. Tyto pole mohou být tisknuty na výstup například funkcí printf, chovají se podobně jako datový typ *string*, který je známý v jazyce C#. Jelikož chování programu bez uvedení nulového bajtu je v jazyce C nedefinováno, bude na jeho nepřítomnost upozorněno při vykonávání programu, pokud bude překročena hranice literálu.

- Předávání vstupních parametrů argc, argv – vstupní parametry argc a argv slouží pro předání uživatelského vstupu při spuštění programu. Tento vstup bude předávaný do funkce main. Podle [10] mohou a nemusí být v rámci funkce main uvedeny, pokud ale jsou uvedeny, tak parametr argc je nezáporné číslo určující počet hodnot v parametru argv, který obsahuje pole ukazatelů na textové řetězce. Tyto hodnoty budou pro interpret speciálním případem předání hodnoty na zásobník. Zdrojem daných hodnot nebude předání skrze jinou funkci, ale budou předány z vnějšího prostředí interpretu.

- Automatické typové konverze - automatické převedení podporovaných datových typů dle implicitních konverzí uvedených ve standardu [10]. Budou řešeny při vyhodnocování hodnot. Ověří se, zda je typová konverze možná, pokud ano bude provedena, jinak bude nahlášena chyba.

[2] https://docs.microsoft.com/en-us/dotnet/api/system.collections.stack?view=net-6.0

[3] https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.linkedlist-1?view=net-6.0

5.3.1 Vyhodnocení proměnných, výrazů

Při zpracování kódu v sémantické analýze budou načteny všechny proměnné do odpovídajících pamětí – globální a statické proměnné do datové sekce, dynamicky alokované na hromadu a proměnné v rámci bloků do zásobníku.

Před přiřazením hodnoty během interpretace bude jejich hodnota uložena jako výraz obsahující konstantní hodnoty, operátory a podobně. V případě nestatické proměnné může obsahovat také odkazy na jiné proměnné. Po vykonání přiřazení hodnota proměnné nabude hodnotu vyhodnoceného výrazu a v případě, že se jedná o konstantní proměnnou, zůstává tato hodnota dále neměnná. Pro úspěšné vyhodnocení přiřazení bude nutné, aby všechny odkazované proměnné v rámci výrazu byly již definovány.

Pro zjednodušení kódu interpretu a zachování všech uvedených vlastností uvedených v této kapitole budou hodnoty v paměti reprezentovány různými objekty s jednotným rozhraním. Rozhraní umožní přistoupit k libovolnému výrazu, konstantní či proměnlivé hodnotě jednotným způsobem. Implementace rozhraní se bude lišit podle specifikací jednotlivých typů hodnot - tzn. konstantu nelze změnit, statická hodnota nemůže obsahovat výraz s proměnnou, ukazatele odkazují na adresu a podobně.

5.3.2 Funkce, předávání argumentů, proměnlivý počet parametrů

Funkce se budou dělit na dva typy – předdefinované a vytvořené uživatelem. Oba tyto typy budou v kódovém segmentu paměti, která umožní jejich jednoduché volání. Součástí tohoto úložiště budou také požadované parametry funkce, abychom mohli ověřit při volání funkce, zda argumenty předávané funkci jsou správné.

Předávání argumentů funkcím bude probíhat skrze zásobník. Při detekci volání funkce se vytvoří nový rámec na zásobníku, do kterého se přiřadí proměnné podle parametrů dané funkce. Do těchto proměnných se nahrají hodnoty argumentů. Pokud se jedná o proměnnou nebo například volání jiné funkce, prvně se vyhodnotí a až poté se nahrají hodnoty. Při vstupu do této funkce budou všechny potřebné proměnné již na zásobníku připraveny a inicializovány a lze s nimi tedy dále pracovat. Pokud některé parametry budou chybět nebo přebývat, program nahlásí chybu. V případě, že funkce obsahuje návratovou hodnotu, bude vložena na zásobník s příznakem, že se jedná o návratovou hodnotu. Tato hodnota bude následně ještě před uvolněním rámce zpracována odpovídajícím způsobem a tedy pravděpodobně přiřazením hodnoty do jiné proměnné.

Speciální případ funkcí jsou funkce s proměnlivým počtem parametrů. Tyto funkce vyžadují alespoň jeden fixní parametr, zbylé parametry jsou uvedeny pomocí tří teček, které značí, že se očekává libovolné množství předem nespecifikovaných argumentů. Pro implementaci tohoto chování v interpretu bude využita nová třída, která bude implementovat rozhraní reprezentující hodnotu v zásobníkové struktuře. Tato nová třída bude přijímat seznam parametrů, který může nabývat proměnlivého počtu položek. Tuto třídu naplníme při zpracování příkazu volání dané funkce.

Případný přístup k jednotlivým hodnotám pomocí va_list, va_start, va_arg a va_end by měl být díky této struktuře také triviální. Pro použití v předdefinovaných funkcích bude ale primárně připravena možnost získat tyto parametry hromadně ze zásobníku v podobě pole.

```
1 int SumNumbers(int x, ...)
2 {
3     int sum = 0;
4     // vytvoreni ukazatele na list parametru
```

```
5     va_list ptr;
6     // inicializace ukazatele na ukazatel na posledni fixovanou hodnotu z parametru
7     va_start(ptr, x);
8     for (int i = 0; i < n; i++)
9     {
10       // nacteni aktualniho parametru a posun na dalsi
11        sum += va_arg(ptr, int);
12    }
13    // ukonceni pruchodu listem
14    va_end(ptr);
15    return sum;
16 }
17
18 int main()
19 {
20  int i = SumNumbers(0, 1, 2, 3);
21 }
```

Výpis 5.1: Ukázka funkce a jejího volání s využitím variadického parametru v jazyce C.

5.3.3 Ukazatele

Ukazatele budou řešeny jako speciální typ hodnoty, který bude obsahovat odkaz na blok paměti alokované na hromadě nebo hodnotu na zásobníku. Také bude obsahovat offset, který umožní využívat základní aritmetiku nad ukazateli, jako je například posun adresy viz. 2.1.1.

Pokud dojde k přístupu do paměti mimo hranice objektu na hromadě a nebo mimo rozsah hodnoty ze zásobníku (bude možné použít aritmetiku například na poli, ale nepůjde použít na konkrétní hodnotě – číslo, znak), bude uživatel upozorněn výjimkou, aby věděl, že pracuje s nedefinovaným chováním. Alternativně lze implementovat tak, že ukazatel propadne na hromadě do dalšího bloku a na zásobníku do dalšího rámce, ovšem to by se týkalo případného rozšíření. V základu nebude uživatel podporován v používání nedefinovaného chování, respektive překračování hranic alokovaných objektů.

I zde bude potřeba řešit také další implementační detaily, jako jsou například operátory *, &, předávání ukazatelů funkcím, konverze ukazatelů na jiné datové typy, vytvoření pole ukazatelů a návrat ukazatele z funkce. Interpret bude také simulovat přístup k nedefinované hodnotě – pokud program přistoupí přes ukazatel na hodnotu z vyprázdněného zásobníku nebo nealokovaného bloku hromady, bude vyhozena výjimka nebo navrácena nulová hodnota.

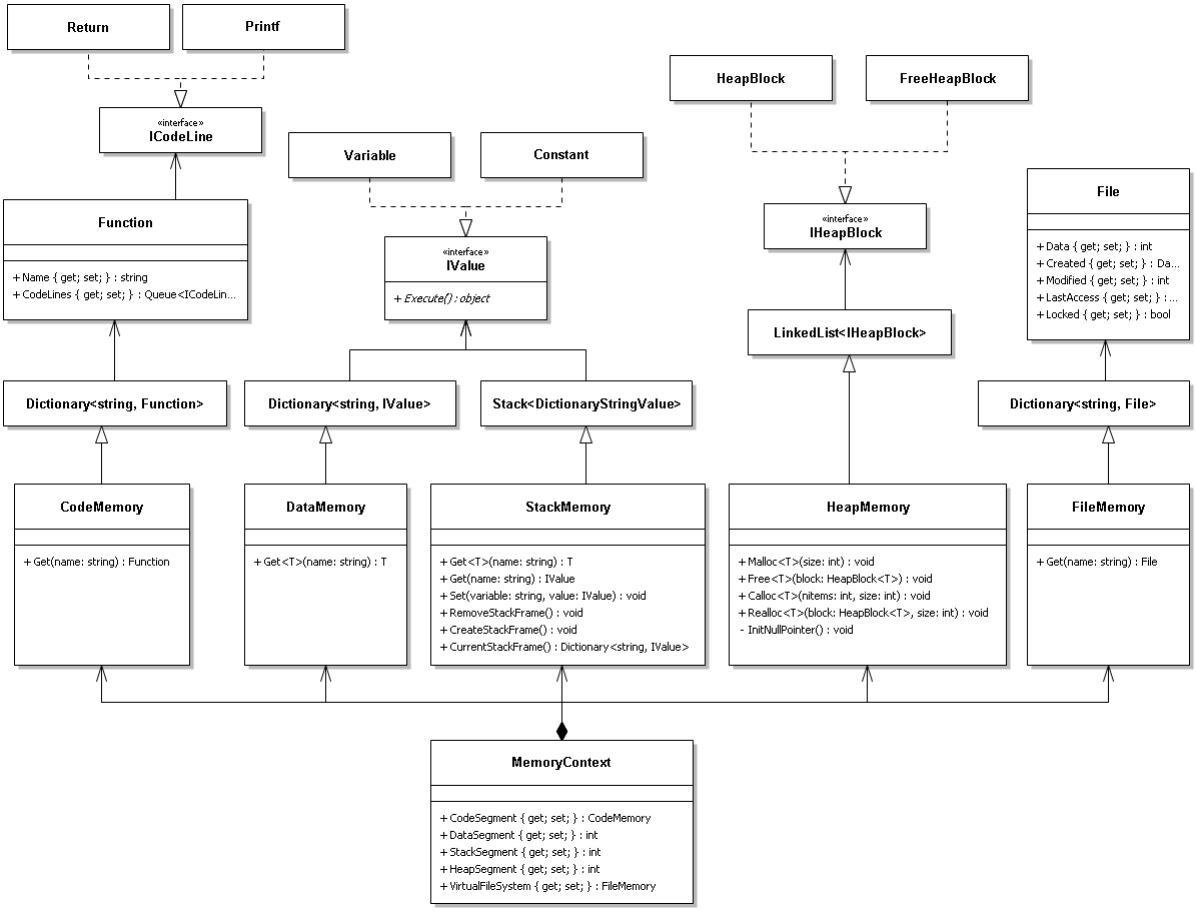
5.3.4 Simulace práce se soubory

V rámci implementace bude zahrnuta také práce se soubory, ta bude pouze simulovaná, jelikož nelze přistupovat k souborovému systému uživatele. Funkce uvedené v sekci 5.4 v knihovně `stdio.h` budou pracovat přímo s tímto simulovaným systémem souborů.

Implementován bude jako nový blok v paměti odvozený od slovníku klíč-hodnota, který bude udržovat název, respektive cestu k souboru, objekt obsahující metadata souboru a textovou proměnnou pro data. Metadata mohou obsahovat datum vytvoření, editace a posledního přístupu k souboru pro případné další rozšíření interpretu. Taktéž bude existovat zámek, určující zda je soubor právě otevřen. Tak můžeme určit zda do něj můžeme zapisovat.

5.3.5 Přehled navrženého modelu paměti

Pro lepší představu uvádím ještě celkový pohled na model paměti vycházející z informací uvedených v této sekci, viz. obrázek 5.1.



Obrázek 5.1: Diagram tříd reprezentující model paměti založený na průzkumu a textovém návrhu v této kapitole. Vzhledem k obsáhlosti byly vynechány některé implementační podrobnosti, ty budou dále rozvedeny v rámci konkrétní implementace.

5.4 Podporované funkce, knihovny, příkazy a další konstrukce

Výčet níže uvádí části jazyka C na které se implementace zaměřuje. Výčet byl sestaven podle projektů, které studenti tvoří v předmětu IZP. Byly extrahovány z mých provedených projektů a z aktuálních webových stránek předmětu IZP. Implementace zmíněných částí umožňuje využití právě v úvodních fázích studia jazyka C a tedy i případné využití v předmětu IZP. Implementace bude probíhat na základě oficiální dokumentace [10]. Tento seznam bude pravděpodobně rozšířen v průběhu implementace a testování o více součástí

jazyka, které vyplynou z reálného použití.

- Obecné
  - Příkazy: if/else, switch, return, cykly: while, for
  - Operátory: (aritmetické, binární, logické, porovnávání), sizeof
  - Typy: signed a unsigned číselné typy, n-dimenzionální pole, základní struktury (bez bitových polí a typedef), ukazatele, funkce, char
  - Direktivy: ifdef, define
- stdio.h
  - Funkce pro práci se vstupem, výstupem: printf, getchar, getc, perror
  - Funkce pro práci se soubory: fprintf, fopen, fscanf, feof, fclose, fgetc
- stdlib.h
  - Převod textu na číselné hodnoty - strtol, atoi
  - Řídicí příkazy: exit
  - Práce s pamětí: malloc, free, realloc
- stdbool.h – podpora datového typu Boolean
- limits.h – Limity datových typů: INT_MIN, INT_MAX, ...
- string.h – Práce s textem: strlen, strncpy, strcmp
- ctype.h – Typová kontrola: isprint, isblank
- assert.h – assert
- math.h
  - Matematické funkce: fabs
  - Konstanty: INFINITY, NAN

5.5 Okamžité vyhodnocení příkazu

V rámci interpretu bude dostupné také okno pro okamžité vyhodnocení příkazu. Pro představu o existujících implementacích lze uvést například příkazové pod-okno v rámci Visual Studio[4], tam se avšak jedná o okno dostupné i při ladění programu a nabízí tedy funkcionality, které jsou v daném kontextu relevantní. V navrhovaném interpretu půjde o provedení příkazu v době, kdy program zrovna neběží. Takové lze vidět v interpretu RustPython 3.2.3.

Tato funkce umožní uživateli volat funkce ze vstupního kódu i jiné funkce příkazy jazyka C bez nutnosti spuštění celého programu, částečně s výsledky volaných funkcí i pracovat. Předpokládaný rozsah příkazu je jeden řádek, práce s výsledky bude tedy značně omezena,

[4] https://docs.microsoft.com/en-us/visualstudio/ide/reference/immediate-window

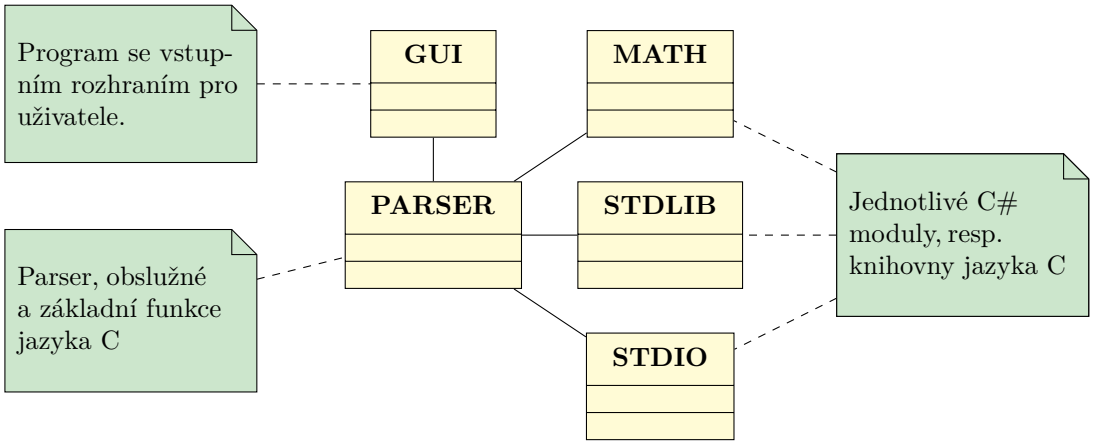
to ale nebrání uživateli zadefinovat si ladící nebo pomocné funkce, které bude volat pomocí okamžitého vyhodnocení. Přínosem této funkce je možnost testování konkrétních funkcí bez nutnosti spouštět celý program. Je možné tak otestovat součásti programu jednotlivě a odladit je nezávisle na sobě.

Implementace by byla provedena podobně jako interpretace běžného kódu. Interpret si načte příkaz a obalí jej funkcí main (případně s jiným identifikátorem) a přidá k němu i ostatní funkce z kódu od uživatele a z použitých knihoven. Následně provede nad tímto kódem lexikální, syntaktickou a sémantickou kontrolou a začne interpretovat. Výpis výsledků v případě použití výstupních funkcí bude vidět v klasickém okně pro výstup. Bude možné použít také vstupní okno jako při běžném spuštění.

5.6 Rozšiřitelnost, modulárnost

Návrh cílí také na možnost rozšíření funkcionalit řešení modulárním způsobem. Pro případ, že by byl interpret dále rozšiřován pro použití v hodinách IZP. Rozdělení jednotlivých částí na moduly zvyšuje přehlednost kódu, umožňuje znovupoužitelnost, samostatné ladění a také případnou dělbu práce [15]. Využití rozdělení do modulů je tedy pro rozvoj o další knihovny v rámci školních projektů více než vhodný.

Jednotlivé moduly budou rozděleny podle knihoven jazyka C - ponesou jejich název a budou do nich dle původního rozložení řazeny i funkce. Při startu programu proběhne registrace jednotlivých funkcí pod danou knihovnou a v případě využití knihovny pomocí direktivy include v uživatelském programu, budou tyto knihovny prohledány, zda obsahují uživatelem volané funkce. Pokud bude implementace požadované funkce nalezena, program deleguje obsluhu požadavku na daný modul a zpracuje výsledek této funkce.

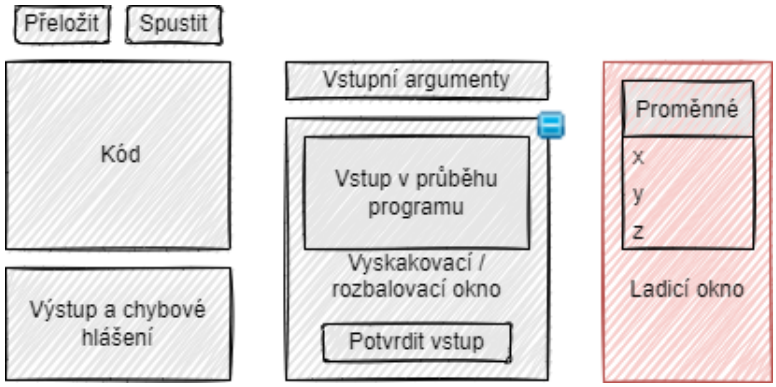


Obrázek 5.2: Diagram balíčků navrhovaného řešení. Poukazuje na důraz na vytvoření rozšiřitelného, modulárního řešení a ukazuje propojení jednotlivých prvků.

5.7 Wireframe grafického uživatelského prostředí

Součástí práce je také uživatelské rozhraní, které je důležitým prvkem pro práci s vytvořeným interpretem, pro zadávání kódu ale i pro získání výstupů a zadávání dalších vstupů. Je tedy vhodné si předem stanovit kompozici prvků, abychom měli alespoň rámcovou představu

o tom jak bude po implementování uživatelské rozhraní vypadat a co bude obsahovat. Tato kompozice je vizualizována pomocí drátěného modelu[5]. Očekávaný formát aplikace je jedna HTML stránka bez dalšího rozpadu struktury na podstránky, případně bude obsahovat textovou stránku s vysvětlivkami.



Obrázek 5.3: Drátěný model uživatelského prostředí, zobrazuje rozmístění jednotlivých prvků včetně případného okna pro ladění kódu.

Na obrázku 5.3 lze vidět následující části prostředí interpretu:

- Tlačítko "Přeložit" – spustí analýzy nad kódem, reportuje chyby ve vstupním kódu bez zahájení interpretace.
- Tlačítko "Spustit" – začne interpretovat mezikód, který vznikl zpracováním po stisku tlačítka přeložit.
- Okno "Kód" – záměrně umístěno blízko tlačítek pro ovládání interpretu, zobrazovaný kód bude vizuálně upraven pomocí nástroje Monaco Editor.
- Okno "Výstupní a chybové hlášení" – umístěno pod kódem, jelikož je úzce navázán na výstup z přeložení nebo spuštění kódu.
- Textový řádek "Vstupní argumenty" – řádek pro vložení argumentů, které se běžně vkládají na příkazovou řádku.
- Okno "Vstup v průběhu programu" – okno pro zadání vstupu na vyžádání za běhu aplikace, může být realizován pomocí vyskakovacího okna a nebo rozbalitelného panelu. Zde bude umístěno také textové okno pro zadání příkazu pro okamžité vyhodnocení.
- Okno "Ladicí okno" - návrh umístění ladění v případě rozšíření o tuto funkcionalitu.

[5] https://wwwproskoly.cz/co-je-to-wireframe-webu/

Kapitola 6

Implementace

Tato kapitola čtenáře seznamuje s průběhem a výsledkem implementace interpretu jazyka C na platformě .NET. Implementace využívá znalostí uvedených v předchozích kapitolách a vychází z uvedeného návrhu implementace, oproti kterému se ale v některých částech liší. Rozdíly mezi návrhem a implementací jsou součástí textu této kapitoly.

Při implementaci bylo dbáno na splnění zadání práce ale také na pokrytí co největšího rozsahu implementace aby mohla být přínosná pro začínající programátory, případně studenty předmětu IZP.

První podkapitola 6.1 popisuje použité technologie platformy .NET pro implementaci. Druhá podkapitola 6.2 uvádí stručný přehled struktury řešení. Třetí podkapitola 6.3 se detailně věnuje zpracování kódu preprocesorem a provedení analýz nad kódem. Interpretace kódu je pak rozebrána ve čtvrté podkapitole 6.4. Následující pátá podkapitola 6.5 uvádí implementaci modelu paměti. Na model paměti se pak podrobněji zaměřuje i šestá podkapitola 6.6, která rozvádí práci s ukazateli včetně diagramu paměti. V sedmé podkapitole 6.7 je popsáno jak je dosaženo možnosti práce offline. Osmá podkapitola zmiňuje kompatibilitu s metodou cut-n-paste 6.8. Devátá podkapitola 6.9 dokumentuje rozsah implementovaných metod. Desátá podkapitola uvádí využité kódy třetích stran 6.10. A nakonec jedenáctá podkapitola 6.11 předvádí výsledné rozhraní interpretu.

6.1 Použité technologie platformy .NET

Na základě návrhu aplikace byly zvoleny technologie, které vycházejí z navrženého řešení.

- Jazyk C# – objektový jazyk, který umožnil implementovat koncepty popsané v návrhu implementace.
- Platforma .NET 7 Preview – v rámci implementace byla použita novější verze platformy, než byla původně navržena a to z výkonnostních důvodů.
- Blazor WebAssembly – platforma pro vytvoření prezentační vrstvy interpretu. Nástroje spojené s touto platformou umožňují kompilaci knihoven v jazyce C# do WebAssembly.
- Windows Presentation Foundation – knihovna pro vytvoření grafického rozhraní. Byla využita čistě pro zrychlení testování, protože Blazor WebAssembly ještě nemá dostatečně spolehlivé ladicí nástroje.

- Visual Studio 2022 – vývojové prostředí pro práci se zmíněnými technologiemi. V době vzniku této práce existuje pouze Preview verze, která je mírně nestabilní ale podporuje .NET 7. Začátek implementace probíhal na verzi 2019 a .NET 6.
- XUnit – open-source nástroj pro testování na platformě .NET.

6.2 Struktura implementovaného řešení

Jedním z cílů navrženého řešení je poskytnutí modulární struktury, která umožní jednoduše zapojit do ekosystému další knihovny. Mohou to být například knihovny, které implementují nestandardní funkce jazyka C. Využití modulární struktury pomůže, v případě budoucího rozvoje, zapojit do vývoje více lidí bez nutnosti podrobně studovat jádro aplikace. To nutně neznamená, že nenastane případ, kdy bude potřeba upravit jádro, ale cílem je tyto případy minimalizovat.

Pro tyto účely byla implementována modulární struktura, která byla naznačena v sekci 5.6. Knihovny jsou rozděleny na samostatné projekty typu "Knihovna tříd" pro .NET Core. Tyto knihovny seskupují kolekce tříd, které reprezentují funkce jazyka C a jsou jedním z prvků modularizace. Mohou obsahovat také definice nových typů (obdoba *typedef*), například typ *FILE* a definice makro výrazů. Postup jakým knihovnu nebo funkce do stávajícího řešení přidat je popsán v příloze A.

Implementace je poměrně rozsáhlá a přehled její kompletní struktury není pro její pochopení nezbytný. Pro základní vysvětlení je namísto toho použitý stručný přehled doplněný o statistiky z měření kódu na obrázku 6.1. Na výsledku analýzy lze vidět, že i přes poměrně vysoký počet řádků kódu – přes 35 tisíc řádků, respektive 10 tisíc řádků spustitelného kódu – je dosažen vysoký index udržovatelnosti kódu[1]. Rozdíl mezi metrikami počtu řádků je jak v prázdných řádcích, tak i v abstraktních třídách a rozhraních, ty se do spustitelných nepočítají. Cyklomatická složitost vyjadřuje strukturální složitost kódu, vyčísluje počet různých cest v toku programu, čím více cest, tím je program méně udržovatelný a vyžaduje více testů [6].

[1] Výpočet udržovatelnosti kódu: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-maintainability-index-range-and-meaning?view=vs-2022

Hierarchie	Index udržovatelnosti	Cyklomatická složitost	Řádky zdrojového kódu	Řádky spustitelného kódu
Interpreter.Core (Debug)	82	3 739	20 194	6 832
Interpreter.Tests (Debug)	75	312	7 528	1 491
Library.Core (Debug)	89	819	3 896	952
Library.StdIO (Debug)	73	527	2 491	634
LearnInC (Debug)	81	95	1 422	295
Library.StdLib (Debug)	70	100	605	227
Library.String (Debug)	68	43	347	120
Library.Math (Debug)	74	37	382	87
TestWPF (Debug)	82	25	245	61
Library.CType (Debug)	77	20	157	45
Library.Assert (Debug)	84	11	69	14
Library.Limits (Debug)	82	4	113	4
Library.StdBool (Debug)	88	4	45	4
Library.Errno (Debug)	88	3	24	3
Library.StdInt (Debug)	94	3	17	2

Obrázek 6.1: Výsledky analýzy kódu. Zobrazují veškeré implementované knihovny a součásti aplikace. U každého celku jsou uvedeny základní statistiky z analýzy kódu. Knihovny implementující funkce jazyka C jsou označeny prefixem *Library* a suffixem dle originálního názvu. *Library.Core* poskytuje funkce, které tyto knihovny využívají. *Interpreter.Core* je jádro celého interpretu. *LearnInC* je prezentační vrstva vytvořená v Blazoru, *TestWPF* je aplikace pro testování napsaná ve WPF a *Interpreter.Tests* je soubor xUnit testů.

Kromě budoucí udržitelnosti kódu bylo dodržení této struktury důležité i pro samotnou implementaci. Dekompozice do jednotlivých modulů a tříd umožnila jednoduše testovat kód a rozšiřovat ho bez ovlivnění existujícího řešení.

6.3 Zpracování vstupního zdrojového kódu

Tato podkapitola se zabývá zpracováním zdrojového kódu v jazyce C a přípravou jeho reprezentace pro interpret, který bude kód následně vykonávat. Popisuje jakým způsobem je kód předzpracován a jednotlivé součásti analýzy kódu nad předzpracovaným kódem.

Zpracování kódu začíná v Blazor aplikaci, ve které uživatel zadá zdrojový kód a stiskne tlačítko pro přeložení zdrojového kódu. V kontextu implementovaného interpretu neprobíhá překlad jako u kompilátoru, ale slouží zde k tomu, aby se do paměti načetly potřebné informace o kódu a ten mohl být následně prováděn efektivněji. Spustit takhle připravený kód lze opětovně bez nového překladu, ten je vyžadován až při změně zdrojového kódu.

Součástí vstupu, předávaného skrze Blazor do dalších fází zpracování, jsou také všechny kódové soubory editoru i stav virtuálního souborového systému. Navíc předává také delegáty funkcí pro práci se vstupem a výstupem.

6.3.1 Předzpracování kódu

Před analýzou kódu probíhá jeho předzpracování preprocesorem, které bylo zapracováno dle popisu ze standardu jazyka C uvedeného v teorii 2.1.2. Předzpracování implementuje první až pátou fázi, zbylé fáze jsou řešeny v rámci analýzy kódu a interpretu.

Preprocesor přijme vstup z Blazor aplikace a předzpracuje ho následujícím způsobem.

1. Z předaných souborů extrahuje soubor *main.c*. Tento název je povinný pro hlavní soubor.

2. V první fázi jsou nahrazeny trigraph sekvence. Využívá jednoduché nahrazení skrze funkce jazyka C# a mapovací tabulku trigraph sekvencí podle standardu.

3. Ve druhé fázi jsou nahrazeny zpětné lomítka na konci řádků, které značí pokračování řádku. Kód je rozdělen na jednotlivé řádky, které se postupně prochází a ukládají do pole řádků. Pokud je na konci řádku nalezeno ukončení zpětným lomítkem, provádí se spojování následujících řádků, které se ukládají na místo prvního takového řádku. Toto spojování pokračuje dokud není nalezen řádek, který nekončí zpětným lomítkem. Tento proces se opakuje, dokud nenarazí na konec souboru.

4. Třetí fáze nahrazuje komentáře v kódu za bílý znak. Pokud komentář obsahuje nový řádek na konci, je tento nový řádek zachován. Vyhledání komentářů k nahrazení je provedeno pomocí regulárních výrazů.

5. Následuje čtvrtá fáze, ve které kód vstupuje do *makro procesoru*. Zpracování maker probíhá na bázi regulárních výrazů, které jsou kompilované pro efektivnější zpracování v cyklech.

   Před spuštěním makro procesoru jsou zaregistrovány předdefinovaná makra __LINE__, __TIME__, __FILE__, __DATE__ a další se přidávají postupně podle zaregistrovaných knihoven. Kód je opět rozdělen na jednotlivé řádky, které se zpracovávají samostatně.

   Pokud je nalezena direktiva *#include* odkazující na soubor, je provedeno načtení kódu tohoto souboru a rekurzivně se na něm vykoná tento proces znova od první po čtvrtou fázi.

   Pokud je nalezena direktiva *#include* odkazující na knihovnu definovanou implementací, pak je tato implementace zaregistrována do paměti. Součástí registrace je přidání implementací funkcí knihovny a registrace definovaných maker.

   Pokud je nalezena direktiva *#define*, zaregistruje se nová definice makra. Makro může obsahovat parametry, včetně proměnlivého počtu parametrů. V parametrech identifikátoru se značí třemi tečkami, v řetězci tokenů pro nahrazení pak identifikátorem pro variadické parametry `#define PRNT(x, ...) do(x, __VA_ARGS__);`.

   Po zpracování všech direktiv se zpracovává zbývající zdrojový kód a provádí se expanze maker. Pokud je nalezen identifikátor, který odpovídá definici regulárního výrazu pro identifikátor makra, prohledá se slovník s uloženými definicemi maker. Pokud je nalezeno odpovídající makro, provede se substituce na místě nálezu, včetně nahrazení parametrů.

   Makro procesor podporuje také direktivy ifdef, ifndef a endif. Pracuje také s operátorem #, který nahrazuje parametr textovým řetězcem a operátorem ##, který spojuje dva parametry do jednoho tokenu (viz. [10], strana 122).

   Výstupem makro procesoru je jeden zdrojový soubor, který obsahuje všechny nalinkované soubory pomocí direktivy `#include`. Ze zdrojového kódu jsou všechny aplikované direktivy odstraněny. Po ukončení jsou v paměti uloženy všechny odkazované knihovny a výstup předán na analýzu zdrojového kódu.

6. Pátá fáze není explicitně řešena, exekuční prostředí je stejné jako prostředí na kterém se zpracovává vstup.

Následující tři fáze překladu jsou součástí lexikální, syntaktické a sémantické analýzy. Po předzpracování kódu jsou nad ním provedeny analýzy, které se dělí na fáze popsané

v teorii 2.4. Jako první probíhá lexikální a syntaktická analýza prováděná pomocí nástroje ANTLR.

6.3.2 ANTLR 4

ANTLR verze 4 je nástroj sloužící k vygenerování parseru, který lze použít pro čtení, zpracování, vykonání nebo přeložení strukturovaného textu nebo binárních souborů. Je široce používán napříč odvětvími a programovacími jazyky, například jej používá Twitter pro analýzu vyhledávacích dotazů nebo Lex Machina pro zpracování právních dokumentů [20]. Oproti předchozí verzi nástroje ANTLR 3, generuje přímo derivační stromy namísto abstraktních syntaktických stromů.

Na vstupu přijímá bezkontextovou gramatiku, která neobsahuje nepřímou nebo skrytou levou rekurzi. Syntaxe gramatiky je podobná gramatikám nástroje *yacc* s operátory z notace EBNF. Z této gramatiky vygeneruje při sestavení projektu dvě C# třídy – Lexer a Parser s rekurzivním sestupem, který využívá strategii adaptivního LL(*) parsování. Tyto dva soubory jsou sestaveny podle pravidel definovaných v gramatice, která se dělí na lexikální pravidla a syntaktická pravidla [21].

Pro vygenerování parseru byla použita existující gramatika pro nástroj ANTLR 4 (6.10). V této gramatice bylo provedeno několik změn, které jsou zdokumentovány přímo ve zdrojovém souboru. Menší změny se týkali například podpory *default* větve v rámci příkazu *switch*, nebylo možné definovat strukturu mimo blok funkce a další drobností, které nebyly gramatikou podporovány.

V gramatice bylo ale také potřeba vyřešit složitější problém. ANTLR přijímá na vstupu bezkontextovou gramatiku, ale gramatika jazyka C není plně bezkontextová a měla problémy s rozpoznáním typu a identifikátoru proměnné. To způsobilo problémy při parsování výrazů, které mohly být výrazem nebo přetypováním, pro příklad výraz `((i*i)*(i*i))`, kde `i` je proměnná, by měl být zpracován pravidlem pro multiplikativní výrazy, ale místo toho byl rozdělen na operátor přetypování, ukazatel, dereferenci a výraz.

Řešení tohoto problému se obvykle nazývá *"lexer hack"*, klíčem je ukládání názvů typů do tabulky symbolů a jejich předání zpět do lexikálního analyzátoru. Při předání tokenu do parseru se následně porovná, zda jde o identifikátor nebo název typu. Podle výsledku porovnání kategorizuje token jako název typu nebo název proměnné [16].

Pro vyřešení tohoto problému byly do gramatiky přidány sémantické predikáty, jejichž využití umožňuje nástroj ANTLR. Predikáty fungují tak, že ANTLR najde všechny použitelné alternativy pravidel a následně zamítne ty, které jsou chráněny nesplněnými sémantickými predikáty. Do pravidel gramatiky byly přidány predikáty `isDeclaration()` a `isType()`, které ověřují, zda se jedná o deklaraci, respektive o název typu. Predikát provádí porovnání lexikálního tokenu a existujících typů definovaných z jazyka C a navíc vlastních typů zaregistrovaných v rámci implementací knihoven. Pokud nalezne, že se jedná o typ, pokračuje pravidlem pro deklaraci, respektive přetypování. Jinak pravidlo zamítne a pokračuje dalším pravidlem.

6.3.3 Lexikální a syntaktická analýza

Lexikální a syntaktickou analýzu po vygenerování odpovídajících tříd z gramatiky dokáže nástroj ANTLR provádět téměř samostatně. Na vstupu se převede předzpracovaný zdrojový kód, který se převede na vstupní tok znaků. Tento datový tok je předán jako vstupní parametr do lexeru, který se stará o lexikální analýzu a generuje tokeny pro syntaktickou analýzu.

Syntaktickou analýzu provádí parser, který na vstupu přijímá datový tok lexikálních tokenů. Parser úzce spolupracuje s lexerem a dokáže si vyžádat další token, nad kterým je možné vyhodnotit syntaktické predikáty.

Pro parser i lexer jsou připraveny také posluchače chyb, které se starají o standardizaci chybového výstupu a zaznamenávání chyb při lexikální a syntaktické analýze. Pokud se ve zdrojovém kódu nějaké chyby vyskytují, jsou po analýze zobrazeny uživateli.

S výjimkou predikátů a chybových hlášení je tahle fáze kompletně vygenerovaná a obsluhovaná skrze runtime nástroje ANTLR.

6.3.4 Sémantická analýza

Implementace sémantické analýzy spojuje analýzu kódu a přípravu na interpretaci do jedné fáze. Tato sekce se nejdříve věnuje obecnému přehledu implementace fáze sémantické analýzy. Následně popisuje, co vše se předem připravuje do paměti pro interpretaci a až na závěr se věnuje samotné sémantické analýze ve smyslu typové kontroly a významové korektnosti zdrojového kódu.

Pro sémantickou analýzu je využita třída `CVisitor` odvozená od třídy `BaseVisitor` vygenerované nástrojem ANTLR z pravidel pro parser, jejíž metody implementuje a rozšiřuje. Tato třída využívá návrhového vzoru *Visitor*, který umožňuje provádět operace nad uzly derivačního stromu vygenerovaného parserem, bez modifikace třídy parseru. ANTLR umožňuje alternativně využití třídy `Listener`, který je použitý i pro hlášení chyb v lexikální a syntaktické analýze, pro implementaci sémantické analýzy se však lépe hodil `Visitor`, který umožňuje kontrolovaný průchod stromem a tak například ukončit expanzi uzlů pokud již máme posbírána všechna podstatná data. Pro přehlednost je třída `CVisitor` rozdělena na několik dalších částečných tříd v samostatných souborech, které seskupují metody podle významu – zpracování výrazů, zpracování příkazů, zpracování struktur a zpracování nepodporovaných částí jazyka.

Každé syntaktické pravidlo je reprezentováno ve třídě `Visitor` svou vlastní metodou, kterou je možné přepsat a nahradit vlastní implementací. Ve výchozím stavu tyto metody expandují své potomky a vrací výsledky metod nad nimi definovanými. `Visitor` prochází derivační strom, který je výstupem z parseru a aplikuje na něj jednotlivá pravidla. Aplikací pravidla dochází k vyvolání metod instance implementované třídy `CVisitor`. Do těchto metod jsou z derivačního stromu předány odpovídající uzly stromu, jejichž potomci se expandují pomocí metody `VisitChildren`, která vyvolá zpracování pravidla pro tyto potomky.

Sémantická analýza byla vytvořena implementací těchto metod. Během sémantické kontroly probíhá současně částečná konstrukce obsahu paměti, po sémantické analýze je kompletně naplněn kódový segment a částečně datový segment.

Pro každé pravidlo, aplikované na derivační strom, je implementováno zpracování jejich potomků. Například pokud `Visitor` prochází pravidlem `VisitTypeName` odpovídající pravidlu parseru pro získání názvu typu `typeName`, vyhodnotí prvně své potomky, které mohou název typu ještě obohatit o kvalifikátory nebo ukazatele. Poté z nich vygeneruje instanci třídy `TypeSpecifier`. Tato instance se dále používá pro ověření typů a předává se do dalších pravidel, například do pravidla pro deklarace `VisitDeclaration`.

Aby se s potomky jednotlivých uzlů lépe pracovalo, jsou agregovány do kolekce typu `List`, kterou je možné dynamicky rozšiřovat. Pro tento účel byla implementována vlastní agregační metoda, která výsledky jednotlivých pravidel ukládá do `Listu` a v případě, že jeho výstupem je již agregovaný list, extrahuje jeho položky a vloží do původního, tak aby

byla zachována plochá struktura – zanořené bloky kódu jsou reprezentovány vlastní třídou `CompoundStatement` místo zanořeného `Listu`.

Tabulka symbolů, ukládání funkcí a typů struktur

Během průchodu derivačního stromu se plní daty také tabulka symbolů. Tato tabulka obsahuje identifikátory a parametry metod, reprezentované pomocí symbolů v tabulce. Tyto symboly obsahují především název, typ a velikost identifikátorů a parametrů, které se v kódu vyskytují.

Protože má jazyk C pravidla pro viditelnost identifikátorů podle umístění jejich deklarace v závislosti na rámci – viditelnost v bloku, ve funkci, v celém souboru – je využita hierarchická struktura tabulky symbolů. Při každém vstupu do funkce se vytvoří nový rámec a do tabulky symbolů se zaregistrují symboly pro parametry této funkce. Následně při vstupu do každého složeného příkazu se vytvoří nový rámec v tabulce symbolů, který má také možnost přistupovat k rodičovskému rámci. Je možné z rámce získat symboly rodičovského rámce, ale ne naopak. Stejně tak je možné provést opětovnou deklaraci identifikátoru v rámci ve kterém se symbol ještě nevyskytuje, ačkoliv existuje v rodičovském rámci. Aby nebylo možné přistupovat k rámcům mimo blok do kterého patří, případně mezi funkcemi, je při opuštění bloku rámec z tabulky symbolů smazán. Ve chvíli kdy je rámec smazán, je již převeden do kódového segmentu v podobě definice, deklarace nebo jiného konstruktu. Pro ovládání rámců jsou definovány metody `NewScope` a `ExistScope`.

V tabulce symbolů nejsou symboly pro funkce. Jejich názvy a návratové typy se vkládají rovnou do kódového segmentu paměti, ze kterého je možné s nimi dále pracovat. Samostatně jsou do paměti také ukládány deklarace struktur, které obsahují informace o jejich názvu a členech struktury. Deklarované typy struktur jsou uloženy ve slovníku, který obsahuje jejich typ, název, velikost a funkce pro inicializaci. Do stejné struktury se ukládají také typy definované v knihovnách, v budoucnosti by se také mohla využívat pro `typedef`.

Naplnění kódového segmentu

Během průchodu se všechny rozpoznávané struktury jazyka C ukládají do objektů, které seskupují jejich atributy do lépe zpracovatelných celků. Tyto objekty se poté, co je dokončen průchod celým podstromem pro konkrétní příkaz nebo výraz jazyka, uloží do kódového segmentu paměti jako řádek kódu do funkce pod kterou patří. Řádky kódu jsou reprezentovány třídami odvozenými od třídy `CodeLine`, např. `Assignment : CodeLine`. Všechny části kódu, které jsou samostatně vykonatelné, lze přetypovat na typ `CodeLine`. To nám umožňuje jednotný přístup a využití návrhového vzoru *Command*. Při procházení kódu tak máme k dispozici pro každou funkci frontu obsahující objekty typu `CodeLine`, nad kterými lze jednotně volat metodu `ExecuteCodeLine`. Tato metoda zajistí vykonání operací podle definice daného objektu. Pro definici vytvoří novou proměnnou v paměti, pro cyklus opakuje zadaný příkaz dokud je splněna podmínka a tak dále. Tento přístup je použit pro příkazy i výrazy.

Prvky, které se nemohou vyskytovat samostatně jako řádek kódu jsou součástí jiných objektů. Například aritmetické operátory se samostatně v kódu vyskytovat nemohou, ale v kombinaci s hodnotou tvoří výraz. Výrazy a operátory jsou zanořeny v objektu typu `Expression`, který vyhodnocuje operátory podle jejich precedence. Tyto výrazy mohou být libovolně zřetězené i zanořené.

Globální definice

Pro globální definice existuje speciální třída udržující řádky kódu mimo funkce. Do této třídy jsou uloženy globální definice a deklarace, které jsou vyhodnoceny ještě před samotným během a výsledky jsou uloženy do datového segmentu. Stav datového segmentu před spuštěním je uložen v kopii paměti, která se opětovně nahraje do paměti před každým spuštěním interpretace zdrojového kódu.

Do datového segmentu se ukládají také řetězcové literály. Jedná se o konstantní textovou hodnotu, která je do paměti uložena jako pole znaků. Pro přístup k hodnotě literálu jsou použity zamaskované ukazatele, které obsahují adresu odkazující na tento literál. Tyto ukazatele nejsou z kódu dostupné (nemají identifikátor a nejsou uloženy v paměti). Je možné k nim přistoupit pouze při vykonávání konkrétního řádku kódu, na kterém jsou uloženy jako hodnota. Tento přístup k hodnotě probíhá automaticky při vyhodnocení řádku.

Sémantická kontrola

Podstatnou částí sémantické analýzy je ověření významu symbolů v rámci výrazů a příkazů ve zdrojovém kódu. Jedná se například o ověření typů a ověření deklarace. Sémantická analýza probíhá částečně také při interpretaci, při které se zejména kontroluje deklarace identifikátoru, typová korektnost při přiřazení nebo překročení limitů pole.

Pro kontrolu kódu byla připravena statická třída `ErrorHelper`, která poskytuje různé variace metody `ReportError`. Tato metoda přijímá v základní variantě na vstupu objekt typu `CodeLine` a konkrétní chybovou hlášku. Výstupem je standardizovaná chyba, obsahující řádek na kterém se chyba vyskytla, textovou reprezentaci uzlu, který se zrovna zpracovává a předanou chybovou hlášku. Tato třída se využívá napříč sémantickou analýzou i interpretací. Chybová hláška je předána na výstup vyvoláním výjimky, která je odchycena na nejvyšší úrovni.

V rámci analýzy jsou kontrolovány především následující chyby:

- Neodpovídající počet argumentů funkce – pokud je funkce volána s více nebo méně argumenty, než je v její definici, je nahlášena výjimka. Pracuje i s proměnlivým počtem parametrů.
- Chybějící deklarace – pokud není identifikátor zaregistrován ale využívá se, je nahlášena výjimka. Kontroluje se také přístup k nedeklarovaným polím struktury.
- Chybné využití operátorů – ověřuje zda se nad hodnotou používají správné operátory, například není možné přistoupit pomocí operátoru `->` ke členu struktury, která není odkazována ukazatelem. Nebo použít index, pokud není hodnota pole nebo ukazatel.
- Typová kontrola – většina typové kontroly je prováděna až při provádění kódu interpretem, v sémantické analýze se však ověřuje například přiřazení řetězcového literálu do proměnné typu znak.
- Nepodporované funkce – v rámci analýzy jsou hlášeny také nepodporované části implementace, aby uživatel nebyl zmaten pokud mu jeho kód nefunguje. Tyto hlášení se dělí na chyby, které zabraňují spuštění (mají příliš mnoho vedlejších účinků) a upozornění. Upozornění jsou například u nepodporovaných kvalifikátorů typů, protože v mnoha případech může být kód plně funkční i bez nich. Nebo také u gcc atributů. Upozornění

nevyužívají ReportError, ale samostatnou kolekci sémantických varování, protože ReportError by ukončil vykonávání kontroly. Pokud je to možné, je doporučena možnost jak chybějící funkcionalitu nahradit využitím jiného přístupu.

6.3.5 Reprezentace a mapování datových typů

Jak bylo naznačeno v této podkapitole, každá hodnota, ať už proměnná nebo konstanta, má svoji reprezentaci pro interpretaci pomocí objektu. Tyto objekty se předem vytváří na základě pravidel v průběhu sémantické analýzy. Ačkoli ještě nejsou uloženy v paměti, předpis pro jejich uložení do paměti je již připraven v kódovém segmentu.

Implementace obsahuje následující třídy pro reprezentaci hodnot, některé z těchto tříd budou popsány dále v textu. V této kapitole jsou uvedeny pro dosazení do kontextu zpracování kódu.

- Constant – třída reprezentující konstantu. Tato třída je specifická tím, že při inicializaci hodnoty dokáže sama převést textovou hodnotu na číselnou, včetně sufixů podle specifikace jazyka C. Hodnoty konstant jsou neměnné a neukládají se při interpretaci do paměti, jsou pouze součástí kódového segmentu a mohou být výsledkem operací.
- Variable – třída reprezentující pojmenované proměnné hodnoty. Na rozdíl od konstant se ukládá do paměti a obsahuje tedy navíc adresu a velikost v paměti.
- StringConstant – třída reprezentující řetězcový literál. Je uložena v datovém segmentu paměti a je možné ji inicializovat pouze ze sémantické analýzy. Přístup k instancím probíhá pouze přes ukazatele viz. 6.3.4.
- Expression – třída reprezentující výraz. Obsahuje list operací, které jsou v rámci získání hodnoty provedeny. Výraz může obsahovat libovolné množství operací, včetně zanořených výrazů.
- ArrayVariable – třída reprezentující n-dimenzionální pole. K této třídě se ve většině případů nepřistupuje na přímo, místo toho se při přístupu rozkládá na ukazatel.
- Pointer – třída reprezentující ukazatele. Její hodnotou je adresa jiné proměnné z paměti.
- StructVariable – třída pro reprezentaci struktur. Slouží jako hlavička obsahující adresu, velikost a názvy jednotlivých členů struktury. Sama o sobě je v paměti uložena pouze o velikosti reprezentující 1 bajt, aby měla adresu a byla možnost na ni odkazovat.

Tyto třídy opět využívají jednotného rozhraní a obsahují metodu `GetValue`, která slouží pro získání hodnoty. Způsob zpracování pak zůstává na dané třídě, pole, ukazatele a struktury však mají metod pro interakci více.

Koncovým typem obsaženým v hodnotě těchto objektů jsou ve většině případů primitivní datové typy, mapování těchto typů je uvedeno v příloze B. Některé rozsahy datových typů nejsou stejné jako v jazyce C, to ale ve většině případů není problém, protože programátor by měl s možností různých velikostí datových typů počítat. Během implementace a testování byl nalezen pouze jeden případ, kdy nefungoval kód, který přistupoval na poslední index v poli znaků pomocí `sizeof(chararray)`, to lze ale jednoduše ošetřit pomocí dělení velikostí datového typu `sizeof(chararray)/ sizeof(char)`.

Typová konverze

Při provádění operací nad primitivními datovými typy může snadno nastat situace, kdy probíhá aritmetická, logická nebo jiná operace nad různými typy. Aby byly typy před provedením operace jednotné a splnily tak typovou kontrolu, využíváme typové konverze.

V implementaci využíváme dva typy konverze, implicitní a explicitní. Implicitní typová konverze probíhá automaticky, když jsou nalezeny rozdílné typy. Hodnota je automaticky převedena, pokud je to možné, na větší datový typ. V případě rozdílu mezi znaménkovým a bez-znaménkovým datovým typem je zvolen datový typ, který je rozsahově větší. Explicitní typová konverze probíhá pomocí operátoru `cast`, který je nutné explicitně použít před hodnotou určenou ke konverzi `c = (int)num`.

6.3.6 Stav po zpracování kódu

Po sémantické analýze je ukončena fáze zpracování zdrojového kódu. Pokud byly nalezeny chyby v rámci některé fáze analýzy, jsou vytisknuty na výstup a uživatel může ověřit a opravit svůj zdrojový kód na základě vzniklých chyb. V případě, že celá analýza proběhla bez problému, spustí se inicializace globálních deklarací a definicí, která inicializuje globální proměnné v datovém segmentu paměti. Zároveň se vytvoří kopie paměti ve stavu po analýze, tak aby byla zajištěna její neměnnost mezi jednotlivými běhy programu – při každém spuštění bez provedení analýzy se do paměti nahraje tato kopie. Následně je uživateli na výstup vytisknuto oznámení o úspěšném zpracování kódu.

6.4 Interpretace

Tato podkapitola popisuje implementaci interpretu a jednotlivé fáze interpretace. Vytvořený interpret využívá model paměti, který je naplněn z fáze zpracování kódu. Tento model je reprezentován obecnou třídou `AppContext` a její implementující třídou `MemoryContext`, dále v textu také pouze jako *paměťový kontext*. Zobecněná třída `AppContext` je navržena pro významové oddělení obecného kontextu od paměťového, se zachováním možnosti společného sdílení napříč aplikací. Interpretace je zahájena poté, co uživatel v rozhraní Blazor aplikace stiskne tlačítko pro spuštění připraveného kódu.

6.4.1 Příprava interpretace

Po zadání pokynu k zahájení interpretace probíhají prvně přípravy. Do jádra interpretu jsou poslány soubory virtuálního souborového systému, které se následně uloží do paměti, aby bylo možné s nimi pracovat. Zároveň se z Blazor aplikace předají delegáty funkcí pro provádění vstupně-výstupních operací. Tyto delegáty se taktéž ukládají do paměťového kontextu, aby byly dostupné napříč aplikací. Před uložením jakékoliv informace do paměťového kontextu je tento kontext nejprve vyčištěn, tím jsou odstraněny výsledky předchozích běhů.

Po zpracování argumentů interpretační funkce proběhne vyhledání hlavní funkce. Typicky se jedná o funkci `main`, ale v některých situacích může být vyvolána i jiná funkce. Pokud je funkce nalezena, vytvoří se rámec v zásobníkové části paměti. V případě, že definice funkce obsahuje parametry argc a argv, jsou tyto argumenty předány do interpretu z Blazor aplikace a zpracovány. Argumenty jsou vloženy na rámec zásobníku pro hlavní funkci. Poté následuje zavolání hlavní funkce, to zajistí ve vykonání všech příkazů a výrazů, které obsahuje.

6.4.2 Průběh interpretace

Interpretace je zahájena voláním hlavní funkce programu, která je reprezentována instancí třídy `FunctionCall : CodeLine`. Tato třída obsahuje frontu všech příkazů a výrazů (dále označováno řádky kódu), které funkce ve zdrojovém kódu obsahuje. Pro provedení funkce je využito jednotné rozhraní třídy `CodeLine`, přes které se zavolá metoda na vyhodnocení funkce.

Třída `FunctionCall` je jedním ze základních kamenů interpretace. Zajišťuje veškeré volání funkcí v průběhu interpretace a to jak hlavní funkce a uživatelsky definovaných funkcí, tak i knihovních funkcí. Na začátku jejího zpracování proběhne příprava nového rámce na zásobníkové části paměti. V rámci přípravy jsou všechny argumenty tohoto volání umístěny na nový rámec a zároveň probíhají implicitní typové konverze. Přidání hodnoty na zásobník se liší podle typu argumentu, který je zpracováván. Pro operátory na získání adresy, dereferenci, indexaci a další operace, například také volání funkce a výrazy, je prvně vyhodnocena jejich hodnota a až ta je uložena na zásobník. Speciálním případem pak je předání pole, které se rozloží na ukazatel. Veškeré předané argumenty jsou před jakýmkoli zpracováním zkopírovány, aby byla zajištěna konzistence paměti. Následně jsou z nich vytvořeny nové instance, které se předávají na zásobník – obalení do nové instance probíhá, aby nemohla funkce modifikovat hodnotu z původního rámce. Pokud chceme pracovat s hodnotami mimo rámec, můžeme použít ukazatele, které obsahují adresu hodnoty v paměti.

Po naplnění je nový rámec umístěn do zásobníkového segmentu paměti a začíná volání jednotlivých řádků kódu. Tyto řádky jsou vykonávány postupně v cyklu, opět s využitím jednotného rozhraní a metody `ExecuteCommand`. Této metodě je předán jako argument paměťový kontext, aby mohly jednotlivé příkazy a výrazy pracovat s naplněným rámcem, ale také s globálními proměnnými a dynamicky alokovanou pamětí. Také je předána reference na volající objekt, aby bylo možné pomocí příkazu `return` provádění funkce ukončit. Provádění je ukončeno po vyhodnocení všech řádků kódu náležejících funkci a nebo v případě předání řízení zpět volající funkci. Poté je vytvořený rámec ze zásobníkového segmentu opět odebrán a hodnota vyhodnocené funkce vrácena volající funkci. V případě volání hlavní funkce je tato hodnota vytištěna na výstup a interpretace ukončena. Pokud se jedná o volání z jiné funkce, pokračuje interpretace navrácením řízení do této volající funkce.

Tímto způsobem probíhá celý proces interpretace, vše začíná a končí voláním metody `ExecuteCommand`. Samozřejmě každý jednotlivý příkaz nebo výraz jazyka C má svou vlastní implementaci a průběh vyhodnocení této metody je odlišný. Například vyhodnocení cyklu `while` se chová podobně jako vyhodnocení volání funkce. Na rozdíl od něj podporuje řídicí příkazy `break` a `continue`. V případě výskytu příkazu `return` pak vrací řízení až do nadřazeného volání funkce. Specifickým prvkem v této třídě je pak způsob práce s rámci zásobníku. Jazyk C umožňuje uvnitř zanořených bloků provádět redefinice proměnných, takže je potřeba pro tělo cyklu vytvořit nový rámec, stejně jako pro argumenty funkcí. Zde je rozdíl v tom, že je předem vyhodnoceno, zda cyklus definici nebo deklaraci obsahuje. Pokud neobsahuje, nejsou rámce vytvářeny a šetří se tak výpočetní čas.

Popis jednotlivých implementací příkazů, výrazů a funkcí by byl příliš konkrétní a nebudou zde z tohoto důvodu uvedeny. V případě zájmu o tyto detaily je možné vycházet ze struktury projektu a prozkoumat kód interpretu, ve kterém lze tyto implementace dohledat a pochopit. Rozsáhlejší implementace budou rozvedeny v rámci samostatných sekcí, například ukazatele, pole a struktury.

Asynchronní volání

Průběh interpretace je celý tvořen asynchronními voláními. Využití asynchronních volání umožňuje provádět během výpočtů vstupně-výstupní operace, jejichž delegáty máme k dispozici. Tyto operace komunikují s uživatelským rozhraním vytvořeném v Blazor aplikaci. Asynchronní zpracování interpretace je nutné, protože Blazor prozatím nepodporuje více vláken a vše probíhá na UI vlákně.

Až budou vlákna implementovány, má smysl změnit provádění na synchronní provádění na samostatném vlákně pro odstranění režijních nákladů. Implementace je pro synchronní přístup připravena.

6.4.3 Dokončení interpretace

Po úspěšném dokončení interpretace je na výstup vytisknuta hláška o dokončení běhu a celkovým časem stráveným interpretací v milisekundách. Do Blazor aplikace se zpětně nahrají modifikované soubory virtuálního souborového systému a proběhne kontrola uvolnění paměti. Pokud programátor neuvolnil veškerou dynamicky alokovanou paměť pomocí funkce `free`, je na výstup vytištěno varování. Toto varování obsahuje informační text a počet neuvolněných bloků paměti.

Pokud je detekována v průběhu interpretace chyba, je vytisknuta na výstup a interpretace je ukončena okamžitě.

6.5 Implementovaný model paměti

Tato podkapitola popisuje konečný stav modelu paměti po implementaci. Oproti návrhu v podkapitole 5.3 se liší, ale základní princip je zachován. Byly provedeny především doplnění o další části a to z důvodů globalizace kontextu, zjednodušení a zrychlení práce s pamětí. Největší změnou je přidání nového paměťového segmentu. Na obrázku 6.2 je zobrazen diagram tříd reprezentující strukturu implementovaného modelu paměti.

6.5.1 Segmenty paměti

Výsledný model je rozdělen na pět segmentů paměti a jeden pomocný segment pro efektivnější práci s ukazateli. Tento model je reprezentován třídou `MemoryContext`, jejíž instance je sdílená pro analýzu kódu i interpretaci. Pro práci s pamětí jsou připraveny také pomocné třídy, které sjednocují přístup k paměti z datového, zásobníkového segmentu i hromady. Veškeré data v segmentech paměti jsou před navrácením kopírována do nové instance objektu, tak aby zůstala paměť neměnná, pokud není změna vyžádána explicitně. Metody pro kopírování objektu jsou implementovány přímo u konkrétních implementací hodnotových typů.

Každý segment paměti má k dispozici předem vypočtenou adresu, na kterou se mají ukládat nové proměnné. Celkový prostor pro adresy je společný a proto jsou sdíleny i poslední využité adresy každého segmentu, aby bylo možné ověřit, že paměť již není alokována jiným segmentem. Celková velikost paměti odpovídá maximální hodnotě typu `ulong` jazyka C#, reprezentující počet bajtů. V paměti je rezervováno 100 bajtů pro interní použití. K dispozici tak je `2^64-101` bajtů paměti. To znamená, že dříve dosáhneme limitu paměti RAM, než vyčerpání paměti v tomto modelu. Paměť je tak značně nadimenzovaná, volné bloky je pak možné využít k interním účelům v budoucnosti. Celý paměťový prostor není předem

alokovaný v paměti prohlížeče, na počátku jsou alokovány pouze interní části, zbytek se alokuje až při vkládání do paměti.

- Kódový segment – kódový segment obsahuje zpracované funkce ze zdrojového kódu. Každá funkce obsahuje frontu řádků zdrojového kódu, implementujících třídu `CodeLine`. Tato část paměti se v průběhu interpretace nemění. Je nežádoucí, aby jakákoli změna proběhla, protože každý řádek může být vyhodnocen vícekrát. K hodnotám v této paměti nelze přistoupit přes adresu.
- Datový segment – datový segment je určen pro globální, konstantní a statické proměnné. Jsou do něj během zpracování kódu ukládány inicializované i neinicializované data, řetězcové literály a globální proměnné. Během interpretace se již tento segment nerozrůstá. Kódový segment při vkládání roste směrem dolů od nejvyšší adresy. Neinicializované data obsahují výchozí hodnoty.
- Hromada – hromada je implementována jako provázaný seznam, který obsahuje objekty reprezentující bloky paměti. Tyto bloky jsou alokovány dynamicky za běhu programu pomocí funkcí `malloc`, `calloc` a `realloc`. Před ukončením programu by měly být uvolněny pomocí funkce `free`. Logika těchto funkcí je implementována jako součást třídy `HeapMemory`. Na rozdíl od ostatních segmentů paměti nejsou objekty uvnitř alokovaných bloků reprezentovány proměnnými typu `IValue`. Místo toho jsou reprezentovány polem bajtů, což umožňuje změny velikosti a reinterpretaci alokované paměti. Toto chování u ostatních segmentů není povoleno. Pro práci s polem bajtů obsahuje konverzní funkce pro datové typy. Adresy na hromadě rostou směrem dolů, od poslední adresy datového segmentu. Při vytváření bloku je také provedena kontrola, zda existuje dostatečně velké uvolněné místo. Pokud existuje, je využito místo alokace nového místa. Sousedící volné bloky jsou spojovány do jednoho.
- Zásobníkový segment – zásobníkový segment slouží pro předávání argumentů funkcím. Je implementován pomocí zásobníku, na který se vkládají rámce s proměnnými. Rámce mohou mít odkaz na rodičovský rámec, ten umožňuje sdílet proměnné mezi rámci. Této vlastnosti se využívá u zanořených bloků, například u cyklů. Adresy v tomto segmentu paměti rostou od nejnižší adresy nahoru.
- Virtuální systém souborů – v paměti jsou uloženy také virtuální soubory. Tyto soubory jsou synchronizovány s uživatelským rozhraním. Tento segment je reprezentován slovníkem s klíčem i hodnotou typu `string`. Klíčem je název souboru a hodnotou jeho obsah, jehož velikost může být až 2 GB. Soubory v této paměti lze vytvářet z uživatelského rozhraní i pomocí knihovních funkcí. V paměti jsou uloženy také otevřené soubory. Otevřené soubory jsou reprezentovány listem objektů typu `FileType`. Tento typ je implementací typu jazyka C `FILE`. Obsahuje název souboru, aktuální pozici v souboru, příznak zda je soubor otevřen ke čtení nebo zápisu a vlajky určující zda bylo dosaženo konce souboru nebo došlo k chybě.
- Mapa paměti – mapa paměti je nový segment paměti přidaný až v rámci implementace. Vznikl pro zjednodušení práce s pamětí skrze jejich adresu. Obsahuje slovník s klíčem typu `ulong` (adresa) a hodnotou typu `IPointable` (objekt, na který lze odkazovat). Při vložení do tohoto slovníku se přidá počet položek slovníku odpovídající počtu bajtů alokovaného objektu. Aby nedošlo ke zbytečnému několikanásobnému alokování paměti pro stejné objekty s různými klíči, je uložena pouze reference objektu, která odkazuje do původní paměti – hromady, souborového, datového nebo zásobníkového segmentu.

6.5.2 Proměnlivý počet parametrů funkce

Model podporuje také proměnlivý počet parametrů funkce. V aktuální implementaci jsou podporovány pouze v rámci definice maker a u knihovních funkcí, napsaných v rámci jazyka C#. Parametry jsou předávány, stejně jako běžné parametry, skrze rámec zásobníku alokovaný při volání funkce. Na rozdíl od běžných parametrů jim je přiděleno automaticky vygenerované unikátní jméno, tak aby se neshodovalo s jinými parametry na zásobníku. Jsou prefixovány označením `var_par_`. Vložení proměnlivého počtu parametrů je povoleno pouze u knihovních funkcí, které tuto možnost explicitně umožňují (viz. definice funkce, příloha A). Pokud je předán jiný počet parametrů u funkce, která proměnlivý počet parametrů nepovoluje, je to vnímáno jako chyba a interpretace je ukončena.

Pro získání parametrů ze zásobníku je k dispozici metoda `GetStackVariadicVariableValue`, která vrací parametry označené jako variabilní v podobě kolekce hodnot. Zpracování této kolekce je ponecháno na implementaci dané funkce.

6.6 Ukazatele

Ukazatel je implementován jako speciální hodnotový typ `Pointer`. Tento typ implementuje třídu `CodeLine` a realizuje rozhraní `IPointerArithmetic`. Toto rozhraní označuje proměnné, nad kterými lze provádět aritmetiku nad ukazateli a přidává šablony metod pro použití indexace nad objektem. Ukazatele jsou implementovány ve třech variantách – obecná třída `Pointer`, specifická třída `Pointer<T>` a speciální případ `NullPointer`.

Každý ukazatel má při vytvoření přiřazen typ hodnoty, na kterou odkazuje, ať už se jedná o primitivní typ, pole nebo další ukazatel. Pro vytvoření typovaného ukazatele se využívá třída `Pointer<T>`, která implementuje nadřazenou třídu `Pointer`. Ve většině případů se však pracuje přímo s nadřazenou obecnou třídou. Je implementována také možnost přetypování ukazatelů za běhu.

Ukazatele mohou být uloženy v libovolném segmentu paměti a jejich velikost je u běžného ukazatele vždy velikostí typu `ulong`, tedy osm bajtů. Hodnota ukazatele je také typu `ulong` a významově jde o adresu proměnné z libovolného segmentu paměti. Pro získání a zápis hodnoty do paměti pomocí adresy používá mapu paměti. Pokud přistupuje na pozici uvnitř pole nebo bloku dynamické paměti, pracuje také s odsazením od adresy prvního elementu v tomto poli nebo bloku. Výpočet odsazení probíhá již při návratu z paměti a je důležité pro správný zápis i čtení z odkazovaných proměnných. Podle kontextu se odsazení většinou převádí na index v poli nebo bloku.

Pro práci s ukazateli jsou definovány následující operátory:

- Adresa – slouží pro získání adresy ukazatele a nebo jiné proměnné. Tento operátor získá adresu proměnné, na kterou je aplikován, a vytvoří z ní ukazatel. Vytvořený ukazatel má speciální příznak `StoredInMemory = false` a `ToMerge = true`. Tyto dva příznaky označují, že se jedná o vygenerovaný ukazatel, který není uložen v paměti a že se při přiřazení do jiného ukazatele má využít vnitřní hodnota ukazatele – adresa na kterou odkazuje, místo jeho adresy. Tento operátor pracuje i v kombinaci s indexací nad proměnnou. V případě, že se jedná o multidimenzionální pole s indexací, která nedosáhne koncové dimenze, získá ukazatel, který reflektuje adresu včetně odsazení dimenze.

- Dereference – tento operátor slouží k získání hodnoty na adrese odkazované ukazatelem. Implementace nejprve vyhodnotí, zda je použita dereference na čistém ukazateli, nebo

se jedná o vícenásobnou dereferenci, adresu, indexaci nebo komplexní výraz. Pokud se nejedná o ukazatel, je prvně vyhodnocena hodnota uvedených operátorů a až poté se získá odkazovaná proměnná a její hodnota.

- Point-to – operátor umožňující přístup k poli struktury, skryté za ukazatelem. Získá odkazovanou strukturu a následně přistoupí ke členům této struktury.
- Přetypování – vytváří instanci ukazatele s novým typem.
- Aritmetika ukazatelů – aritmetika nad ukazateli je řešena jako součást výrazů. Při vyhodnocování operátorů uvnitř výrazu se postupuje po dvojicích. Nejprve ověří, zda alespoň jeden člen dvojice je typu ukazatel. Pokud ano, je operátor aplikován na odkazovanou adresu uvnitř ukazatele. Pokud je ukazatel pouze jeden z dvojice, je druhá hodnota vynásobena velikostí ukazatele a posun tedy vždy probíhá po násobcích daného typu.
- Indexace – slouží k získání hodnoty na daném indexu. U jednoúrovňových ukazatelů provádí posun odsazení od aktuální odkazované adresy. Pokud se jedná o vícenásobný ukazatel, například dynamicky alokované n-rozměrné pole, může být uvedeno i více indexů, které pak slouží pro pohyb v dimenzích dle uvedených indexů.

Speciálním případem ukazatele je rozklad pole na ukazatel. V těchto případech je při předávání a práci s polem provedena konverze na ukazatel. Pro zjednodušení a identifikaci ukazatelů, které vznikly automatickou konverzí z pole, obsahuje ukazatel zvláštní příznaky. Ukazatel pro tyto účely obsahuje příznak určující, že se jedná o takzvaný *decay pointer* a referenci objektu držící hodnoty a vlastnosti tohoto pole. Pro ulehčení konverze na ukazatele jsou pole již při definici pod tímto maskovacím ukazatelem ukládány do paměti.

Ukazatele mohou odkazovat na libovolnou dimenzi n-dimenzionálního pole a také na jednotlivé dimenze, které jsou součástí vícenásobného ukazatele. Pokud ukazatel odkazuje na neexistující adresu, změní se jeho typ na `NullPointer`, respektive odkazuje na nulovou adresu.

6.6.1 Pole

Pole je implementováno pomocí třídy `ArrayVariable`, tato třída, stejně jako ukazatel, implementuje třídu `CodeLine` a realizuje rozhraní `IPointerArithmetic`. Ve většině kontextů je skryta pod zvláštním ukazatelem, který slouží například pro předávání do funkcí. Většinu akcí však deleguje do této třídy. Třída obsahuje běžné pole pro práci s pamětí a stejné možnosti indexace jako ukazatele.

Na rozdíl od ukazatelů obsahuje také počet všech dimenzí pole a počet prvků v aktuální dimenzi. Při průchodu dimenzemi v případě použití indexace nepotřebuje získávat data z paměťového kontextu, neboť si všechny reference na data drží ve své instanci. V paměti je tak uložena pouze rodičovská dimenze, rezervována je ale paměť pro všechny dimenze. Při získávání hodnot z pole se používají dle kontextu dva různé přístupy. Jedna z možností je přistoupení přímo na koncový index a získání hodnoty na tomto indexu. Druha varianta je optimalizovaná pro inkrementální pohyb v poli – prvky poslední dimenze jsou mezi sebou jednosměrně provázány a při inkrementaci indexu mají možnost použít ukazatel na svého pravého souseda v paměti.

Třída obsahuje také auto-inicializační funkci, která při definici pole inicializuje předanou hodnotou všechny dimenze. V případě, že pole není inicializováno, provede se líná inicializace

až před prvním přístupem na některý index pole – ta nastaví všechny prvky pole na nulovou hodnotu a zajistí tak, že aplikace nespadne. Inicializaci až při přístupu si interpret může dovolit, neboť paměť pro prvky pole je vyhrazena už při jeho deklaraci.

V případě přístupu mimo rozsah pole je uživatel informován o této skutečnosti a interpretace je ukončena. Jedná se o opatření proti přístupu mimo zamýšlenou paměť. Této chyby se začínající programátoři neúmyslně dopouštějí a pak nemusí jejich kód fungovat správně. Pole nepodporují dynamické stanovení velikosti za běhu programu, uživatel je na tuto skutečnost upozorněn.

6.6.2 Struktury

Struktury mají také svoji vlastní třídu `StructVariable`, která implementuje třídu `CodeLine` a realizuje rozhraní `IPointable`. Umožňuje uložení do paměti, ale už ne použití indexu nebo aritmetiky.

U implementace struktury byl použit rozdílný přístup než u polí, které jsou v paměti reprezentovány jako jeden objekt, který se stará o veškerou interakci s dimenzemi. Tento přístup byl u pole zvolen, abychom měli plnou kontrolu nad tím, zda uživatel přistupuje do správného pole a nepřesahuje jeho dimenze. Tento přístup vyžadoval relativně vysoké množství dodatečných podmínek pro ověření typu objektu, se kterým pracujeme. Proto jsem se rozhodl u struktur pro jiný přístup, který lépe vystihuje organizaci tohoto typu. Struktura je v paměti reprezentována objektem typu `StructVariable` o velikosti jednoho bajtu. Tento objekt v paměti slouží pro označení místa, kde struktura v paměti začíná. Obsahuje také informace o typu, celkové velikosti struktury, ukazateli na prvního člena struktury, typech a názvech členů struktury a v rámci optimalizace také seznam odsazení adres jednotlivých členů struktury. Seznam odsazení členů struktury je vygenerován již při analýze kódu a slouží pro rychlé získání počáteční adresy člena struktury bez dodatečných výpočtů.

Tato třída definuje metody pro inicializaci struktury a přístup ke členům struktury, konkrétně čtení, zápis a získání jejich jména, typu a velikosti. Protože členové struktury nejsou součástí této třídy, ale jsou v paměti uloženy samostatně, bylo potřeba vyřešit předávání na zásobník při volání funkce. Pro tento účel je vytvořena metoda, která iteruje všechny členy struktury a vloží je na rámec zásobníku. Členové struktury jsou na rámec vloženy pod unikátním identifikátorem, skládajícím se z názvu struktury a názvu typu, tak abychom zachovali vazbu a nezpůsobili interferenci s jinými proměnnými.

Struktury jsou v rámci interpretu implementovány pouze částečně. Vzhledem k tomu, že se tento interpret zaměřuje na začínající studenty předmětu IZP a struktury jsou většinou součástí až posledního projektu, tak byly upřednostněny jiné komponenty jazyka. Nepodporované jsou dynamické alokace struktur, bitová pole, zanořené struktury, inicializace rovnou při definici a předávání struktur jako výsledků funkcí. I přesto jsou použitelné pro celou řadu případů – inicializace lze řešit po jednotlivých členech, předávání z funkcí pomocí ukazatelů.

6.6.3 Simulace práce se soubory

Implementace podporuje také virtuální souborový systém, který je uložen v paměťovém kontextu. Pro práci se soubory je implementována knihovna `Library.StdIO`, která reprezentuje standardní knihovnu jazyka C pro práci se vstupem a výstupem. Tato knihovna definuje také vlastní datový typ `FILE`, který slouží pro práci se soubory ve zdrojovém kódu.

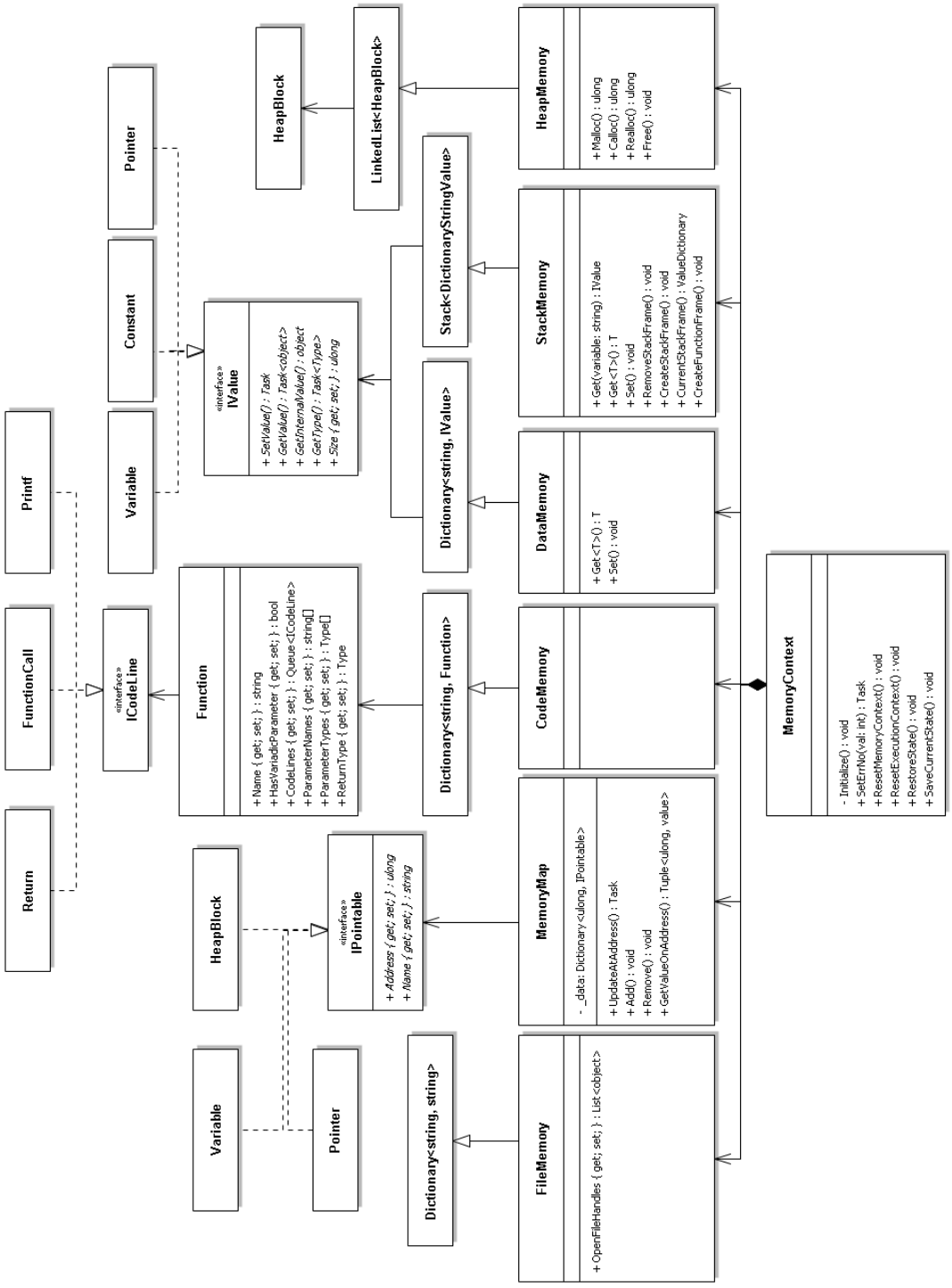
Funkce `fopen` pro otevření souboru podporuje otevření souboru v režimech `r, w, a, r+, w+, a+`. Po otevření souboru se vytvoří záznam v seznamu otevřených souborů s metadaty

určujícími stav souboru. Tento seznam se používá pro určení, zda je soubor již otevřený a je možné s ním pracovat.

Pro práci se soubory byly implementovány také chybové stavy `ERRNO`, které se předávají do paměťového kontextu při selhání. Tato selhání mohou nastat například pokud soubor neexistuje, není otevřený nebo pokud se funkce snaží zapisovat do souboru otevřeného pouze pro čtení.

Implementované metody pro práci se soubory jsou zaměřeny na získávání a zapisování textových řetězců. Otevřené soubory jsou v paměti reprezentovány s odsazením adresy o deset adres. Těchto prvních deset adres je rezervovaných pro interní použití. Na tyto rezervovaná místa odkazují například identifikátory `stdout` a `stderr`, které přesměrovávají výstup na odpovídající kanál.

6.6.4 Diagram implementovaného modelu paměti



Obrázek 6.2: Diagram tříd implementovaného modelu paměti. Tento diagram zobrazuje všechny dostupné segmenty paměti a jejich metody. Vzhledem k rozsáhlosti reálné implementace byly vybrány pouze důležité metody a vynechány parametry funkcí. Realizace rozhraní hodnotových typů nejsou kompletním výčtem.

6.7 Práce offline a generování kódu do WebAssembly

Tyto dva body zadání spolu úzce souvisí. Aplikace ve WebAssembly, které nevyužívají serverové připojení, umožňují práci offline. Pro prvotní načtení je připojení k síti potřeba, ale poté je aplikace uložena v prohlížeči a lze ji spustit i pokud připojení není dostupné.

Pro převod kódu interpretu do WebAssembly je použita AOT kompilace (viz. 2.3). Pro povolení kompilace jsou do projektu přidány atributy, které označují projekt pro AOT kompilaci. Aby mohla být kompilace provedena, je potřeba na počítač, který kompilaci provádí, nainstalovat nástroj `wasm-tools`. Výsledkem kompilace jsou původní knihovny i knihovny převedené do WebAssembly, ale také HTML stránky a JavaScriptové funkce pro spuštění a práci s WebAssembly soubory. Výstupní velikost interpretu se pohybuje kolem 100 MB, při spuštění se však nenačítá celý. I přesto může prvotní načtení trvat desítky sekund.

6.8 Možnost přenosu metodou cut-n-paste

V rámci implementace má být umožněn přenos kódu metodou *cut-n-paste*. Tento požadavek spočívá v možnosti zkopírovat kód jazyka C z počítače nebo jiného zdroje a spustit jej v interpretu a naopak. Souvisí to s nežádaným chováním některých online kompilátorů a interpretů, které do kódu implicitně doplňují `#include` direktivy. Tím, že v online nástroji běží kód i bez doplnění knihovny, nelze vždy takto vytvořený kód zkopírovat do offline kompilátoru a přeložit.

Tento požadavek je v rámci implementace splněn, až na jednu výjimku a to definice typů `size_t` a `intN_t`, které byly přidány přímo do gramatiky. Tato úprava byla provedena kvůli potřebě jejich globální přístupnosti v knihovnách.

Mimo uvedenou výjimku je možné kód kopírovat tam i zpět. V interpretu ale nelze spustit kód, který obsahuje neimplementované funkce, v takovém případě zahlásí chybu. Chybějící funkce je také možné dodefinovat vlastním kódem.

6.9 Implementované funkce, knihovny a konstrukce jazyka C

Tato podkapitola uvádí rozsah implementovaných konstrukcí jazyka a knihovních funkcí. Základními kameny pro implementaci byly části uvedené v analýze 5.4. Tento seznam byl oproti původnímu očekávání výrazně rozšířen, a to na základě vlastního i uživatelského testování.

Implementace konstrukcí jazyka i knihovních funkcí vychází ze standardu jazyka [10]. U časově náročných knihovních funkcí byla snaha využít externí kód, viz. 6.10. Kde to bylo proveditelné, tam byly využity odpovídající funkce z jazyka C#. Zdaleka ne všechny funkce se shodují s implementací v jazyce C# a tak bylo i pro jednoduché funkce nutné ošetřit některé cesty.

Konkrétní implementace nebudou v rámci této podkapitoly rozsáhle popisovány. Jejich prozkoumání v rámci zdrojového kódu je ponecháno na čtenáři, protože jejich popis by byl příliš detailní a stejné informace lze získat prozkoumáním kódu.

6.9.1 Obecné konstrukce jazyka

Implementované části:

- Cykly: do-while, while, for

- Větvení: if-else, switch
- Řídicí příkazy: break, continue, volání funkce, return
- Příkazy: přiřazení, deklarace, definice
- Typy: všechny primitivní typy, konstanty, proměnné, ukazatele, vícerozměrné pole, struktury, funkce, řetězcové literály, null ukazatel, možnost definovat vlastní v rámci knihovny, size_t, FILE
- Operátory: aritmetické, logické, bitové, operátory pro práci s ukazateli, indexování pole, sizeof, ternární operátor, unární operátory (+, -, , !), operátor přístupu ke členům struktury, přetypování, ...
- Výrazy: hodnotové typy lze vložit do výrazů a vyhodnotit s ohledem na precedenci použitých operátorů a uzávorkování
- Direktivy: define, ifdef, ifndef, endif, include

Neimplementované části:

- Enumerace, goto
- Struktury: bitová pole, dynamická alokace, použití jako návratový typ, typedef, definice včetně hodnot (nutné definovat členy zvlášť), union
- Kvalifikátory typů: const, static
- Pole: dynamická velikost pole při deklaraci
- Ukazatele: ukazatel na funkci
- Typy: ptr_diff
- GCC rozšíření, zarovnání
- Direktivy: if, else, undef, elif, ...

6.9.2 Knihovna assert.h

Obsahuje pouze funkci `assert`, kterou lze vypnout pomocí definice makra `NDEBUG`.

6.9.3 Knihovna ctype.h

Obsahuje funkce `isblank, isdigit, isprint, isspace, isxdigit, tolower, toupper`. Tyto funkce z většiny využívají funkce z jazyka C#, doplněny o podrobnosti ze specifikace jazyka C.

6.9.4 Knihovna errno.h

Neobsahuje žádné funkce, pouze zpřístupňuje identifikátor `errno`, který je uložen jako globální proměnná na datový rámec při registraci knihovny.

6.9.5 Knihovna limits.h

Neobsahuje žádné funkce, při registraci přidává definice maker pro velikosti celočíselných datových typů.

6.9.6 Knihovna math.h

Při registraci přidává makra `INFINITY, NAN`. Obsahuje funkce `fabs, fabsf, fabsl, log, logf, pow, powf, sqrt, sqrtf, isinf, isnan, exp, expf, round, roundf, roundl`.

Tyto funkce využívají převážně implementace z jazyka C#, které jsou doplněny o zaznamenávání stavu do `errno`.

6.9.7 Knihovna stdbool.h

Neobsahuje žádné funkce, přidává makra `bool, true, false, __bool_true_false_are_defined`.

6.9.8 Knihovna stdint.h

Neobsahuje funkce ani makra, přidána pouze pro kompatibilitu kódu. Z této knihovny jsou v řešení dostupny typy `intN_t`.

6.9.9 Knihovna stdio.h

Využívá externí implementace printf a scanf, viz. 6.10.

Přidává typ `FILE`, registruje identifikátory `stdin, stdout, stderr`. Obsahuje funkce `fclose, feof, ferror, fgetc, fgets, fopen, fprintf, fputc, fputs, fscanf` pro práci se soubory. A také `getc, getchar, perror, printf, putchar, puts, scanf, sprintf`.

Implementace těchto funkcí nemohou využít žádných ekvivalentních implementací z jazyka C#. Bylo potřeba je, mimo printf a scanf, implementovat vlastnoručně podle specifikace. Funkce printf nepodporuje všechny formáty (například "%*d"), ale většinu standardních ano.

6.9.10 Knihovna stdlib.h

Obsahuje `atoi, atol, atoll, calloc, exit, free, malloc, realloc, strod, strol, strtoul`.

Funkce byly implementovány vlastnoručně podle specifikace, neexistují ekvivalentní funkce jazyka C#.

6.9.11 Knihovna string.h

Obsahuje `memcpy, memset, strcat, strcmp, strcpy, strchr, strlen, strncat, strncpy`.

Tyto funkce také nemají ekvivalenty využitelné pro implementaci a byly tak implementovány vlastnoručně podle specifikace.

6.10 Použité kódy třetích stran

V této sekci jsou explicitně uvedeny převzaté části kódu z veřejně dostupných zdrojů, které byly využity jako celek v rámci optimalizace času na projektu. Licence a autoři jsou zdokumentovány v hlavičkách souborů v projektu s vyznačenými změnami.

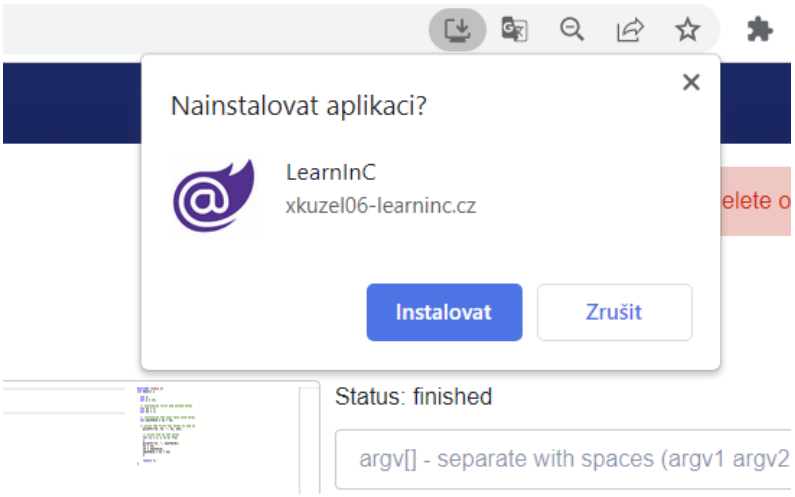
Nejsou zde uvedeny knihovny a součásti, které jsou probírány v sekci s technologiemi 6.1 a v jiných částech textu – například nástroj ANTLR nebo xUnit. Tyto součásti jsou instalovány do projektu jako balíčky, které mají své licence uvedené u sebe a nebyly na nich prováděny žádné změny.

V případech kdy byla některá minoritní část implementace inspirována veřejně přístupným článkem nebo diskuzí k problému, je taková skutečnost uvedena u každého výskytu přímo v kódu.

1. Gramatika jazyka C – soubor C.g4 – gramatika jazyka C pro ANTLR. Provedeny úpravy a opravy.
2. Funkce printf – soubor PrintfImplementation.c – C# implementace funkce printf z jazyka C. Provedeny úpravy, opravy a rozšíření.
3. Funkce scanf – soubory ScanfImplementation.c a TextParser.c – C# implementace funkce scanf z jazyka C. Drobné úpravy pro kompatibilitu.

6.11 Uživatelské prostředí

Tato podkapitola popisuje implementované uživatelské prostředí. Toto prostředí využívá framework Blazor WebAssembly pro provázání jádra interpretu s vizuálními prvky. Využití Blazoru umožňuje následnou kompilaci celku do WebAssembly a tím umožňuje práci offline a také instalaci do počítače viz. obrázek 6.3.



Obrázek 6.3: Z prohlížeče je možné interpret také stáhnout a nainstalovat do počítače, to je umožněno využitím PWA (progresivní webová aplikace). Po instalaci je aplikace dostupná, v systému Windows, přímo z nabídky Start.

Obrázek 6.4: Prostředí implementovaného interpretu. Toto prostředí je dostupné online (a poté i offline) na adrese `https://xkuzel06-learninc.cz/`. Obrázek zobrazuje stav, kdy jsou přidány do souborového systému kódové a textové soubory.

- Ovládací panely – ovládací panely slouží pro řízení práce s programem. Tlačítko *Build* spustí zpracování zdrojového kódu, tlačítko *Execute* spustí interpretaci zpracovaného kódu. Běh je možné ukončit tlačítkem *Halt*. Další dvě tlačítka slouží ke spuštění a ovládání krokovacího režimu, který je součástí rozšíření směrem k ladění kódu.
- Správce souborů – ovládání souborového systému. Umožňuje vytvořit soubor s definovaným názvem, tento název musí být unikátní napříč kódovými i virtuálními soubory. Soubory je možné také mazat nebo vytvářet přímo pomocí funkcí jazyka C za běhu programu. Kódové soubory mohou být vloženy jako modul pomocí direktivy `#include`.
- Okno pro zdrojový kód, výstup – okno pro zdrojový kód slouží pro zadání zdrojového kódu, formátování je zajištěno editorem Monaco. Okno pro výstup zobrazuje standardní i chybový výstup programu. Navíc zobrazuje také čas provádění.
- Práce se vstupem – textové okno nahoře slouží pro zadání vstupních argumentů, které jsou poté vloženy na zásobník jako proměnné `argc, argv`. Okno níže slouží pro zadání vstupu, pokud je vyžádán během provádění programu, vstup je nutné potvrdit tlačítkem *commit stdin input*. Pokud se čeká na uživatelský vstup, je tato skutečnost oznámena textem nad okny a také podbarvením tlačítka pro zadání vstupu.

6.11.1 Okno pro okamžité vyhodnocení

Grafické rozhraní interpretu obsahuje také okno pro okamžité vyhodnocení. Toto okno umožňuje zadat výraz nebo příkaz a potvrdit pomocí tlačítka *Run command*. Na vstupu se očekává pouze krátký, jednořádkový kód, například `printf("print from immediate window");`. To je ale pouze předpoklad z UX hlediska, implementačně rozsah omezen není. Okno je vhodné pro částečné testování funkcionality programu.

Okamžité vyhodnocení probíhá tak, že vložený kód se obalí funkcí, která dostane unikátní identifikátor. Tato funkce je následně předána i s kódem uvedeným v okně pro zdrojový kód. Při spuštění interpretu je předán argument, který identifikuje předanou funkci jako novou hlavní funkci. Nespouští se tak funkce main, která je definovaná ve zdrojovém kódu, ale funkce obalující vložený kód pro okamžité vyhodnocení.

6.11.2 Rozšíření směrem k ladění

Součástí zadání práce byla také diskuze na téma rozšíření směrem k ladění. Z průběhu analýzy a implementace vyplynulo, že by ladění mohlo být velmi užitečné při práci s interpretem. Proto místo pouhé diskuze byl proveden POC (*proof-of-concept*, neboli zkušební provedení), který možnost rozšíření demonstruje zavedením krokového režimu.

Krokový režim poskytuje uživateli možnost krokovat jednotlivé akce prováděné interpretem. Je zahájen stiskem tlačítka *Stepping mode*. Prozatím je možné se v kódu pohybovat pouze směrem vpřed a to pomocí tlačítka *Next step*, které kód posune o řádek dál. Toto tlačítko je podbarveno, pokud se čeká na uživatele. Při každém kroku je v okně *stepping mode output* zobrazen aktuální stav proměnných v zásobníkovém rámci. To zahrnuje i případné proměnné z rodičovského rámce, například pokud jsme v zanořeném bloku. Aktuální prováděný řádek je podbarven červeně, aby uživatel měl kontext, kde se právě pohybuje.

Implementace byla díky návrhu architektury relativně snadná. Klíčovým konceptem při implementaci bylo využití jednotného rozhraní skrze třídu `CodeLine` a sdílený paměťový kontext.

Kapitola 7

Testování

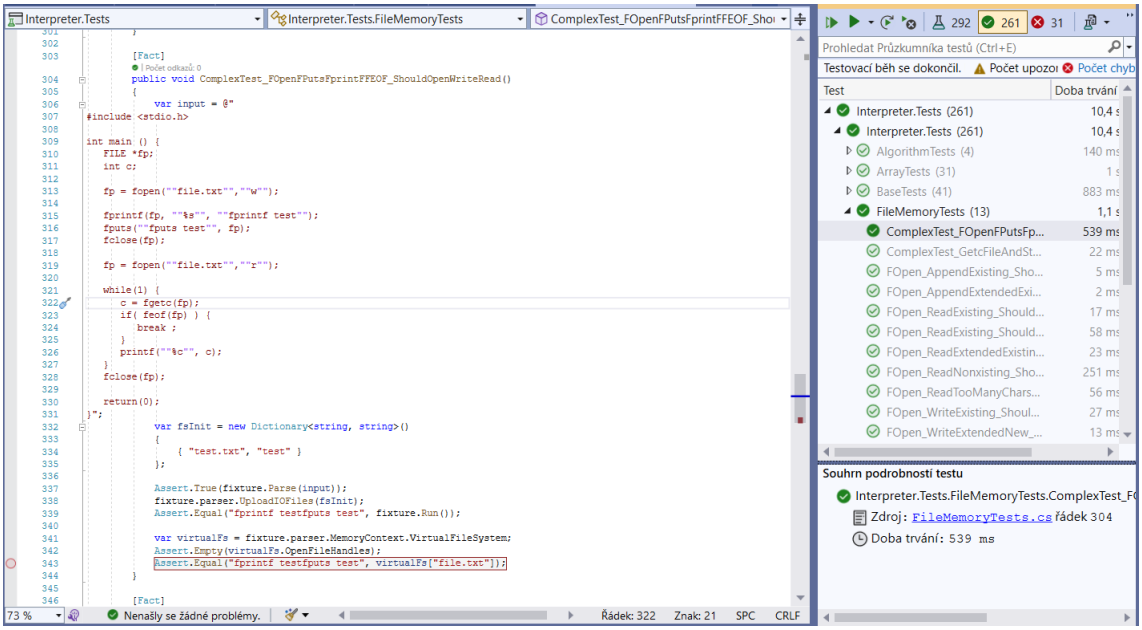
Tato kapitola se věnuje testování interpretu. Testování interpretu probíhalo v několika krocích. Jako první byly připraveny automatické testy 7.1, které byly používány v průběhu implementace. Následně byly provedeny uživatelské testy 7.2, které poskytly důležitou zpětnou vazbu a podněty pro rozvoj. Po dokončení implementace byly provedeny výkonnostní testy 7.3, které slouží pro vyhodnocení rychlosti výsledného řešení.

7.1 Automatické testování

Pro stabilizaci vývoje byly po implementaci základní struktury interpretu přidány automatické testy s využitím testovacího nástroje xUnit. Tyto testy pomohly v průběhu řešení odhalit množství chyb a také sloužily jako připomínka ještě nedokončených funkcionalit.

Ve fázi po stabilní implementaci základů interpretu byla provedena změna přístupu k vývoji na takzvaný *test driven development*. Jedná se o přístup, kdy programátor jako první krok nadefinuje testy funkcionalit, které chce implementovat a až poté provádí implementaci. Pokud je implementace provedena správně tak předem definované testy budou po implementaci úspěšné.

Testy byly navržené jako *funkcionální testy*, které, na rozdíl od unit testů a integračních testů, testují systém jako celek včetně všech jeho závislostí. Pro každý jeden test je nadefinován vstupní zdrojový kód, tak jak by ho vložil uživatel. Po spuštění testu je inicializováno zpracování zdrojového kódu a interpret, až na pár výjimek stejně, jako když uživatel spustí kód z Blazor aplikace. Testovací běh ignoruje některé varování a také jiným způsobem pracuje se vstupem a výstupem – hodnoty, které posílá na vstup má předdefinované. Pro pojmenování testů byl zvolen vzor pojmenování *Funkce_Akce_Výsledek*, viz. obrázek 7.1.



Obrázek 7.1: Ukázka funkcionálních testů z prostředí pro automatické testování ve Visual Studio 2022 Preview. Celkem bylo vytvořeno 292 testů pokrývajících funkcionality interpretu. 31 z těchto testů slouží jako upomínka na nepodporované funkce pro případ budoucího rozvoje.

7.2 Uživatelské testování

Následující fází bylo uživatelské testování do kterého se zapojilo celkem 5 bývalých studentů předmětu IZP. Testování probíhalo na projektech, které studenti tvořili v rámci svého studia do předmětu IZP. Celkem tak bylo otestováno včetně dvou online zdrojů 21 různých projektů z let 2014, 2015, 2016, 2017 a 2019. Testování probíhalo také na příkladech z wiki stránek předmětu IZP.

Během testování se odhalilo několik chyb v implementaci a také chybějících funkcí, které studenti běžně ve své práci používali. Jednalo se o funkce `putchar`, `isspace`, `strchr`, `exp`, `expf`, `round`, `roundf`, `isinf`, `isxdigit`, `fgets`, `strtoul`, `isnan`, které byly na základě zpětné vazby implementovány.

Ve výsledném stavu implementace bylo ve většině případů možné zpracovat a provést první dva projekty předmětu IZP bez úprav. U některých projektů bylo však nutné provést změny na základě omezení, nejčastěji nemožnosti použít dynamickou hodnotu pro definici pole.

Naopak shodně selhává třetí projekt z předmětu IZP, který pravidelně obsahuje pokročilou práci se strukturami. U některých zdrojových kódu bylo pro zprovoznění dostatečné odstranit typedef. Jakmile však došlo na zanořené struktury a vracení struktur jako návratové hodnoty, tak byly testy ukončeny. Takové projekty by vyžadovali příliš velké zásahy do jejich zdrojového kódu, aby je bylo možné spustit a to není cílem.

7.3 Výkonnostní testování

Poslední fází testování je výkonnostní testování, které se zaměřuje na měření stráveného času při provádění interpretace, využití procesoru a spotřebu paměti. Pro výkonnostní testování bylo zvoleno pět algoritmů uvedených v příloze C. Tyto algoritmy byly zvoleny na základě rozdílných časových složitostí a počtu operací, které interně interpret provádí.

Testování probíhalo částečně již v průběhu vývoje a podle jeho výsledků byly provedeny optimalizace paměťového modelu. Také byla provedena aktualizace na preview verzi .NET 7, která obsahovala změny v kompilaci WebAssembly a přinesla zrychlení přibližně o čtyřicet procent.

Pro testování bylo vybráno pět algoritmů. Algoritmy jsou očištěny o IO funkce, aby se eliminovalo nepředvídatelné zpoždění interakce s UI vláknem. Implementace jsou uvedeny včetně testovacích vstupů v příloze C:

- FIB1 – implementace rekurzivního výpočtu fibonacciho posloupnosti C.1. Použitý algoritmus má časovou složitost $\mathcal{O}(2^n)$ a prostorovou složitost $\mathcal{O}(n)$.
- FIB2 – implementace iterativního výpočtu fibonacciho posloupnosti C.2. Použitý algoritmus má časovou složitost $\mathcal{O}(n)$ a prostorovou složitost $\mathcal{O}(1)$.
- SEL1 – implementace řadícího algoritmu selection sort C.3. Použitý algoritmus má časovou složitost $\mathcal{O}(n^2)$ a prostorovou složitost $\mathcal{O}(1)$.
- BIN1 – implementace rekurzivního binárního vyhledávacího algoritmu C.4. Použitý algoritmus má časovou složitost i prostorovou složitost $\mathcal{O}(logN)$.
- KNAP1 – implementace optimalizačního algoritmu knapsack C.5. Použitý algoritmus má časovou složitost $\mathcal{O}(2^n)$ a prostorovou složitost $\mathcal{O}(1)$.

Testy byly prováděny napříč hlavními prohlížeči, které podporují WebAssembly. Každý prohlížeč může implementovat WebAssembly jiným způsobem a lze tedy očekávat, že se budou časy vykonávání lišit. Prohlížeče Chrome, Opera a Edge mají stejné jádro, zde je předpoklad podobných výsledků. Prohlížeče Firefox a Firefox Nightly se liší v tom, že verze Nightly poskytuje implementace některých částí WebAssembly, které ještě nejsou oficiálně součástí Firefoxu.

Pro srovnání jsou uvedeny také časy interpretace pomocí JSCCP 3.2.3, který je svou funkcionalitou nejpodobnější z nalezených řešení a uvedené zdrojové kódy na něm lze spustit beze změny.

Ve výsledcích je uveden referenční čas běhu stejné implementace v aplikaci napsané ve WPF 6.1. Tato implementace využívá spouštění mimo UI vlákno, sdílí ale společné jádro interpretu s Blazor aplikací a používá tak asynchronní operace, které navyšují režijní čas.

Výsledky byly dosaženy provedením 13 běhů každého algoritmu, ze kterých se odstranil první, nejlepší a nejhorší výsledek a zbylých 10 bylo zprůměrováno.

Testování probíhalo na stroji Dell XPS 15 9500 s povolenou hardwarovou akcelerací v prohlížečích. Každý test probíhal se stejnými podmínkami – prohlížeč byl před testem aktualizován, byly vypnuty všechny aplikace až na měření spotřeby paměti a procesoru. V prohlížeči byla otevřena pouze záložka s interpretem.

- Procesor – Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz, 2592 Mhz, jádra: 6, logické procesory: 12
- Nainstalovaná fyzická paměť (RAM) – 16,0 GB

- Grafická karta – NVIDIA GeForce GTX 1650 Ti
- Opera – 86.0.4363.59
- Chrome – 101.0.4951.67
- Firefox – 101.0b6
- Firefox Nightly – 102.0a1
- Edge – 101.0.1210.47
- Operační systém: Windows 10

7.3.1 Výpočetní čas

	Chrome	Firefox	Nightly	Edge	Opera	JSCPP	WPF
FIB1	11021.9	18618.7	18023.2	11232.5	10832.7	34218.33	571.1
FIB2	776.8	1385.9	1352.8	804	757.6	error	48.2
SEL1	10741.8	18310.5	17869.4	11358.1	10711.6	6177	608.8
BIN1	584.9	954.9	930.5	609.9	587.2	984.75	38
KNAP1	16936.5	27453.4	26228.6	18117.5	16784	61501	1067.5

Tabulka 7.1: Tabulka výsledků výkonnostních testů. Uvádí délku provádění algoritmu napříč různými prohlížeči v milisekundách. Ve dvou posledních sloupcích je zobrazena délka provádění pomocí C++ interpretu JSCPP v prohlížeči Chrome a délka provádění na lokálním počítači v aplikaci typu WPF.

7.3.2 Spotřeba paměti

	Chrome	Firefox	Nightly	Edge	Opera	JSCPP	WPF
FIB1	638 MB	979 MB	799 MB	673 MB	678 MB	375 MB	57,7 MB
FIB2	651 MB	994 MB	825 MB	664 MB	667 MB	error	61,6 MB
SEL1	655 MB	882 MB	837 MB	674 MB	668 MB	337 MB	66,9 MB
BIN1	670 MB	924 MB	846 MB	667 MB	689 MB	340 MB	58,4 MB
KNAP1	667 MB	926 MB	860 MB	680 MB	701 MB	355 MB	61,1 MB

Tabulka 7.2: Tabulka výsledků výkonnostních testů. Uvádí spotřebu paměti napříč různými prohlížeči. Ve dvou posledních sloupcích je zobrazena spotřeba paměti C++ interpretu JSCPP v prohlížeči Chrome a na lokálním počítači v aplikaci typu WPF. Paměť byla měřena jako celková alokace paměti využitá pro prohlížeč.

7.3.3 Využití procesoru

	Chrome	Firefox	Nightly	Edge	Opera	JSCPP	WPF
FIB1	16 %	16 %	16 %	16 %	16 %	18 %	4 %
FIB2	5 %	8 %	9 %	5 %	5 %	error	1,2 %
SEL1	16 %	16 %	17 %	17 %	16 %	15 %	4 %
BIN1	4 %	7 %	6 %	4 %	4 %	8 %	1 %
KNAP1	16 %	17 %	17 %	16 %	16 %	16 %	8 %

Tabulka 7.3: Tabulka uvádí využití procesoru napříč různými prohlížeči. Ve dvou posledních sloupcích je zobrazena využití procesoru C++ interpretu JSCPP v prohlížeči Chrome a na lokálním počítači v aplikaci typu WPF. Využití bylo měřeno jako celkové využití procesoru prohlížečem.

7.4 Zhodnocení výsledků testování

Uvedené výsledky v tabulkách 7.1, 7.2 a 7.3 byly uvedeny již po optimalizacích. Tato podkapitola diskutuje nalezené výkonnostní problémy a jejich možné řešení. Věnuje se také rozdílům mezi jednotlivými prohlížeči.

7.4.1 Rychlost jádra interpretu

Jako problematické body v implementaci jádra, které by mohly být ještě zlepšeny, byly identifikovány následující body:

- Přístup k modelu paměti – při každém přístupu do paměti se vytváří kopie objektu z paměti, která zajišťuje, že původní objekt v paměti zůstane nemodifikován. Při rekurzivních voláních to může znamenat velké množství takových volání, které by teoreticky mohly být eliminovány.
- Asynchronní operace – asynchronní operace mají režijní náklady. Asynchronním operacím se však nelze vyhnout, jelikož Blazor prozatím nepodporuje více vláken. Výpočet by tak zablokoval UI vlákno a nebylo by možné přijmout uživatelský vstup.
- Reflexe a model mapy paměti – během prvních testů byly objeveny také problémy s nadměrným využíváním reflexe a ve vyhledávání adres v paměti, kde byla použita pomalejší kolekce. Tyto problémy byly zredukovány v průběhu analýzy. Navíc byly provedeny optimalizace počtu vytvořených instancí objektů.

7.4.2 Rychlost zkompilovaného WebAssembly řešení

Rychlost řešení ve zkompilovaném Blazor WebAssembly řešení je výrazně nižší, než při spuštění lokální aplikace ve WPF. Zpomalení oproti WPF se výrazně liší mezi prohlížeči na jádru Chromium a Firefox. Interpretace v prohlížeči je přibližně 15-30x pomalejší, v závislosti na prohlížeči a vykonávané úloze. Při analýze bylo nalezeno několik bodů, které toto zpomalení způsobují. Pro hledání problémů byly využity vývojářské nástroje prohlížeče Chrome a diskuze na téma problémů s WebAssembly a Blazorem.

- Rekurze, zanořené cykly – největší zpomalení nastává při provádění operací, které vytváří rámce v paměti a volají se s vysokým počtem opakování. Při prozkoumání

kódu a dostupných informací o WebAssembly jsem došel k závěru, že je to způsobeno právě vysokým počtem volání jednotlivých WebAssembly funkcí, které mohou využívat interpretovaný režim .NET v případě nepodporované funkce.

- Reflexe – v interpretu se pro vytváření objektů za běhu a typovou kontrolu využívá reflexe. Podle průzkumu není reflexe ještě plně podporována v Blazor pro AOT kompilaci do WebAssembly a používá se tak pro tyto operace interpretovaný režim Blazoru, který je pomalejší.
- UI vlákno – Blazor WebAssembly prozatím nepodporuje více vláken (očekává se, že budou součástí .NET 7). To znamená, že veškeré operace běží na UI vlákně, které musí zajišťovat také překreslování obsahu. To vede k velkým rozdílům při použití interpretace včetně IO funkcí a bez nich.

Dalším logický krok pro zrychlení interpretace je vyčkání na implementaci vláken a podporu reflexe v rámci .NET 7. Pro podporu běhu mimo UI vlákno je program již připraven, funguje tak WPF aplikace. Pokud tato aktualizace proběhne, je teoreticky možné odstranit také asynchronní volání metod a tím opět zrychlit vykonávání.

7.4.3 Porovnání s JSCPP

Při porovnávání s JavaScriptovou implementací JSCPP je pro zpracování rekurzivních operací WebAssembly implementace rychlejší. JavaScriptová implementace má naopak lepší výsledky pro iterativní algoritmy a také má menší spotřebu paměti. Iterativní fibonacci algoritmus se nepodařilo spustit kvůli chybě přetečení číselné hodnoty.

Tyto dvě technologie ale nelze na základě výsledků interpretace přímo porovnávat, jelikož interní implementace interpretů je jiná a rozdíly mohou být způsobeny implementačními detaily.

7.4.4 Porovnání prohlížečů a závěr

Podle očekávání prohlížeče využívající stejné jádro (Opera, Chrome, Edge) mají velmi podobné výsledky a to co se týká času tak i spotřeby systémových prostředků. Z této trojice má nejlepší výsledky prohlížeč Opera, který byl ve 4/5 případů nejrychlejší. Naopak nejpomalejší byl prohlížeč Edge.

Neočekávaně byl prohlížeč Firefox a i jeho o něco rychlejší verze Nightly, téměř dvakrát pomalejší než prohlížeče na jádru Chromium. Podle průzkumu a seznamu funkcionalit[1] jednotlivých prohlížečů by měl tento prohlížeč být nejpokročilejší v počtu implementovaných funkcionalit. Přesto z testů vyšel jako nejpomalejší.

Během uživatelského testování nebyl nalezen kód, jehož provedení by trvalo nepřiměřeně dlouho. Považuji tak výsledný stav za použitelný pro práci na projektech v rámci IZP, ačkoliv pro výpočetně náročné operace nemusí být rychlost uspokojivá.

Z výsledných časů lze také odvodit, že interpret je řádově pomalejší než kompilátory. Zpomalení oproti kompilátoru bylo očekávané, přesto byly provedeny kroky pro redukci času a analýza problémových míst. K určení problémů byly použity profilovací nástroje z prostředí Visual Studio.

[1] https://webassembly.org/roadmap/

Kapitola 8

Závěr

Cílem práce bylo vytvořit nástroj pro zpracování jazyka C v prohlížeči s využitím WebAssembly v kombinaci s platformou .NET. Takový nástroj je výsledkem této práce a je dostupný na adrese https://xkuzel06-learninc.cz/.

Po úvodním studiu související teorie, existujících řešení a dostupných nástrojů na platformě .NET jsem vyhodnotil, že nejlepší variantou bude řešení pojmout jako interpret. S vedoucím práce jsme si stanovili jako cílovou skupinu studenty předmětu IZP, kteří teprve začínají s programováním a tak pro ně rychlost řešení není stěžejním faktorem. Cesta interpretu může studentům nabídnout i další nástroje pro práci s kódem a lepší vyhodnocení chyb než v kompilovaném kódu.

Po prostudování související teorie byl vytvořen návrh implementace s ohledem na vlastnosti jazyka C. Tento návrh při samotné implementaci prošel pouze drobnými změnami. Tyto změny se týkaly především rozšíření funkcionality a přidání částí, které umožnily rychlejší zpracování kódu. Cílem bylo navrhnout architekturu tak, aby byla dlouhodobě udržovatelná a modulární, tedy jednoduše rozšiřitelná v případě, že by se projekt dále rozvíjel.

Podařilo se implementovat interpret s využitím frameworku Blazor WebAssembly, který umožňuje kompilaci interpretu do WebAssembly. Kompilací interpretu do WebAssembly je dosaženo schopnosti práce offline včetně možného nainstalování do počítače. Implementovaný interpret nabízí uživatelům kromě zpracování zdrojového kódu také možnost vložení příkazu pro okamžité vyhodnocení a kopírování i vkládání *cut-n-paste* metodou. Poskytuje virtuální souborový systém pro přidávání kódových i textových souborů a práci s těmito soubory.

Nad vytvořeným interpretem proběhly tři fáze testování. První z nich byla realizována pomocí automatických testů již při vývoji. Po dokončení implementace proběhlo uživatelské testování bývalými studenty předmětu IZP. Testování ukázalo, že je interpret schopný pokrýt dva ze tří projektů, které studenti v průběhu roku tvoří. Funkce jenž nebyly pokryty interpretem ani po rozšíření na základě uživatelského testování, bylo možné ve většině případů nahradit alternativní implementací. Poslední fází bylo výkonnostní testování. Toto testování poukázalo na očekávaný fakt, že interpretace je pomalejší oproti kompilovanému kódu. Hlavně ale přineslo zajímavou informaci o rozdílech mezi výkonností stejného řešení ve WebAssembly a přímo na počítači ve frameworku WPF. Rozdíly byly detekovány také mezi jednotlivými prohlížeči, nejlépe si vedl prohlížeč Opera a nejpomalejší byl prohlížeč Firefox.

Architektura interpretu poskytuje možnost relativně jednoduché implementace rozšíření o ladění kódu. Po zvážení přínosů byl na toto téma učiněn krátký průzkum a implementace částečného ladění. Tato implementace spočívá v zavedení krokového režimu, který umožňuje uživateli vyhodnotit zdrojový kód po řádcích a zároveň kontrolovat, co se děje v paměti.

V budoucnosti je možné tuto implementaci rozšířit o možnosti krokování zpět, nastavování zarážek nebo uživatelské změny hodnot v paměti.

Z pohledu budoucího rozvoje je interpret připraven na rozšíření a rád bych se jeho vývoji dále věnoval, protože to pro mě byl velmi zajímavý projekt a myslím si, že by mohl být přínosný v kontextu vzdělávání studentů. Dokázal bych si představit, že by mohl být tento projekt také předmětem dalších studentských prací a to jak v ohledu rozšíření o další knihovny a funkce jazyka, tak především v možnosti vytvořit z tohoto nástroje celý ekosystém nástrojů pro vzdělávání studentů. Na mysli mám například rozšíření o plné ladění, našeptávání funkcí, živé zvýraznění chyb v kódu nebo interaktivní kurzy jazyka C.

Literatura

[1] Aho, A. V. a Aho, A. V., ed. *Compilers: principles, techniques, & tools*. 2nd ed. Pearson/Addison Wesley, 2006. ISBN 9780321486813.

[2] Albahari, J. a Johannsen, E. *C# 8.0 in a nutshell: the definitive reference*. First edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2020. ISBN 9781492051138.

[3] *ASP.NET Core Blazor* [online]. Microsoft, 17. dubna 2022 [cit. 2022-05-01]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/blazor/?view=aspnetcore-6.0.

[4] Běhálek, I. M. *Programovací jazyky a překladače: Úvod do překladačů* [online]. Katedra informatiky FEI VŠB-TUO [cit. 2022-04-08]. Dostupné z: http://www.cs.vsb.cz/behalek/vyuka/pjp/prednasky/prekladace-4.pdf.

[5] *Cleaning up unmanaged resources* [online]. Microsoft, 17. září 2021 [cit. 2022-05-01]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/unmanaged.

[6] *Code metrics values* [online]. Microsoft, 30. dubna 2022 [cit. 2022-05-06]. Dostupné z: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022.

[7] Cruysse, J. V. der. *Cs-wasm* [online]. [cit. 2022-02-10]. Dostupné z: https://github.com/jonathanvdc/cs-wasm.

[8] *Emscripten* [online]. Emscripten Contributors [cit. 2022-02-05]. Dostupné z: https://emscripten.org/.

[9] *Fetch API* [online]. MDN Contributors [cit. 2021-12-30]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.

[10] ISO. *ISO/IEC 9899:2017 Information technology — Programming languages — C*. International Organization for Standardization, 2017.

[11] Jenny Chen, Ruohao Guo. *Stack and Heap Memory* [online]. [cit. 2022-01-09]. Dostupné z: https://courses.engr.illinois.edu/cs225/fa2021/resources/stack-heap/.

[12] Lamansky, R. *Dotnet-webassembly* [online]. [cit. 2022-02-10]. Dostupné z: https://github.com/RyanLamansky/dotnet-webassembly.

[13] Linden, P. V. der. *Expert C Programming: Deep C secrets*. [1. ed.]. Mountain View: SunSoft Press, 1994. ISBN 9780131774292.

[14] *Loading and running* [online]. MDN Contributors [cit. 2021-12-30]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly/Loading_and_running.

[15] MARTINEK, D. *Moduly a knihovny* [online]. [cit. 2022-16-01]. Dostupné z: http://www.fit.vutbr.cz/~martinek/clang/modules.html.

[16] MCPEAK, S. a NECULA, G. C. Elkhound: A Fast, Practical GLR Parser Generator. In: DUESTERWALD, E., ed. *Compiler Construction*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, s. 73–88. ISBN 978-3-540-24723-4.

[17] MDN CONTRIBUTORS. *WebAssembly* [online]. Mozilla Corporation [cit. 2022-02-05]. Dostupné z: https://developer.mozilla.org/en-US/docs/WebAssembly.

[18] *Null-terminated byte strings* [online]. cppreference community [cit. 2022-16-01]. Dostupné z: https://en.cppreference.com/w/c/string/byte.

[19] *OnlineGDB* [online]. OnlineGDB team [cit. 2021-12-30]. Dostupné z: https://www.onlinegdb.com.

[20] PARR, T. *ANTLR: ANother Tool for Language Recognition* [online]. [cit. 2022-20-02]. Dostupné z: https://www.antlr.org/.

[21] PARR, T., HARWELL, S. a FISHER, K. Adaptive LL(*) Parsing: The Power of Dynamic Analysis. [online]. New York, NY, USA: Association for Computing Machinery. 2014, sv. 49, č. 10, s. 579–598. DOI: 10.1145/2714064.2660202. ISSN 0362-1340. Dostupné z: https://www.antlr.org/papers/allstar-techreport.pdf.

[22] REESE, R. *Understanding and Using C Pointers*. O'Reilly Media, Inc., 2013. ISBN 9781449344184.

[23] *Run-wasm* [online]. Slip [cit. 2021-12-29]. Dostupné z: https://www.runwasm.com/.

[24] STARICH, J. *How to compile code in the browser with WebAssembly* [online]. [cit. 2022-01-18]. Dostupné z: https://blog.johnstarich.com/how-to-compile-code-in-the-browser-with-webassembly-b59ffd452c2b.

[25] TOUB, S. *Performance Improvements in .NET 6: Blazor and mono* [online]. Microsoft, 17. srpna 2021 [cit. 2022-02-05]. Dostupné z: https://devblogs.microsoft.com/dotnet/performance-improvements-in-net-6/#blazor-and-mono.

[26] *WebAssembly* [online]. WebAssembly Community Group [cit. 2022-02-05]. Dostupné z: https://webassembly.org/.

[27] *What is .NET?* [online]. Microsoft [cit. 2022-01-15]. Dostupné z: https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet.

Příloha A

Přidání nové knihovny

Tato sekce ukazuje jak řešení rozšířit o další funkcionality, tím však zároveň popisuje jakým způsobem je modularita zajištěna.

Pro registraci knihovny do ekosystému je vyžadováno několik pravidel:

- Knihovna musí být zaregistrována jako závislost v projektu `Interpreter.Core`, který má na starost načtení a vyhodnocení knihoven za běhu.
- U každé knihovny je doporučeno ustálené pojmenování `Library.NazevPuvodniKnihovny`, suffix `NazevPuvodniKnihovny` slouží pro přehled o obsahu knihovny a v ideálním případě by měl odpovídat knihovně jazyka C, kterou implementuje, například `Library.StdIO`.
- Knihovna musí definovat atribut, který obsahuje základní informace o knihovně a určuje také její název využívaný v rámci příkazu `include` ve zdrojovém kódu jazyka C. Vyplněný atribut lze vidět na výpise A.1. Původně se mělo jednat o atribut sestavení, ale kvůli nedostatečné podpoře reflexe ve WebAssembly byl tento přístup zrušen. Uvedené informace se při načítání vypisují na výstup ladicí konzole.
- Pro korektní registraci a načtení knihovny je potřeba přidat třídu `LibraryStartup.cs`, která dědí abstraktní třídu `LibraryBase`, potažmo implementuje rozhraní `ILibraryStartup` viz. výpis A.1. Doporučeno je využít dědičnosti.
- Nakonec stačí knihovnu zaregistrovat do seznamu referencí v souboru `LibraryResolver.cs`. Vzhledem k použití Blazoru totiž nelze využít DependencyContext, který by nám umožnil knihovny vyhledat dynamicky podle daných pravidel a skrze ReferencedAssemblies tyto knihovny nedohledáme, jelikož nemají využití přímo z kódu hlavní knihovny.

Takto registrované knihovny jsou poté zpracovány automaticky knihovnou Interpreter.Core, obsahující třídu `LibraryResolver`, která má metody na registraci knihoven a načtení funkcí jednotlivých knihoven.

```
namespace Library.StdIO
{
    public class LibraryStartup : LibraryBase
    {
        // citelny nazev, nazev implementovane knihovny, popis funkce, autor
        public override LibraryInfoAttribute LibraryInfo { get => new
            LibraryInfoAttribute("Library.StdIO", "stdio.h", "Standard IO lib", "Michal
            Kuzela"); }
        // lze doplnit pri registraci custom funkcionalitu
        public override void OnStartupRegistration(MemoryContext memoryContext)
        {
            base.OnStartupRegistration(memoryContext);
        }

        // moznost doplnit vlastni makra
        public override List<MacroEntry> OnRegisterCustomMacros(string[]
            registeredMacros)
        {
            return base.OnRegisterCustomMacros(registeredMacros);
        }

        // pri zpracovavani kodu a nacteni #include se jmenem knihovny se zavola tahle
            metoda
        public override void OnLoadFunctions()
        {
            // registrace funkce printf pro vyuziti pri provadeni kodu
            // STD IO
            RegisterFunction(new Printf());
            // File IO
            RegisterFunction(new FPrintf());

            // registrace custom typu File
            RegisterCustomTypeResolver("FILE", typeof(FileType), (parameters) => new
                FileType()
            {
                Name = parameters.Name,
            }, 1);
        }
    }
}
```

Výpis A.1: Registrace knihovny a jejich funkcí v rámci LibraryStartup.cs.

A.1 Přidání nové funkce

Implementace nové funkce je z pohledu zaregistrování do systému o něco jednodušší, než registrace knihovny, má však podobné rysy. U funkci však může být náročná samotná implementace - záležet bude na tom, co se snažíme implementovat. Předdefinované rozhraní nabízí možnosti mimo implementaci logiky také přepsat provádění sémantické kontroly, pro možné krajní případy, to se však nyní nevyužívá. Pro registraci funkce je potřeba:

- Vytvořit třídu, ideálně dodržovat adresářovou strukturu podle existujících knihoven, a dodat jí třídní atribut, který určuje její jméno, vstupy, výstupy. Viz. výpis A.2.
- Pro implementace funkcí je opět připravena abstraktní třída, v tomhle případě *LibraryFunctionBase*, kterou třída s implementací musí dědit.
- Posledním krokem je registrace funkce, jak je naznačeno ve výpise A.1.

```
1  namespace Library.StdIO.Functions
2  {
3      // nazev funkce, navratovy typ, pole typu parametru funkce, pole nazvu jednotlivych
           parametru, boolean urcujici zda funkce obsahuje variadicky parametr
4      [LibraryFunction("printf", typeof(int), new Type[] { typeof(Pointer<char>) }, new
           string[] { "format" }, true)]
5      public class Printf : LibraryFunctionBase
6      {
7          // metoda pro zpracovani prikazu, na vstupu dostava aktualni kontext pameti, se
               kterym muze pracovat, napr. nacist parametry, tisknout na stdout
8          protected async override Task<IValue> ExecuteInternal(MemoryContext
               memoryContext)
9          {
10             // nacteni parametru
11             string? format = await PointerHelper.GetString(memoryContext, MemoryHelper.
                   GetPointer<char>(memoryContext, "format"));
12             object[] variadicArgs = (await MemoryHelper.GetStackVariadicVariableValue<
                   object>(memoryContext))
13                 .Select(p => p is char[] chArr ? string.Join("", chArr) : p)
14                 .ToArray();
15
16             // vykonani logiky
17             var print = await PrintfImplementation.sprintf(format, variadicArgs);
18             // tisk na vystup
19             await memoryContext.PrintToOutput(print);
20             // vraceni hodnoty
21             return new Constant() { Value = print.Length };
22         }
23     }
24 }
```

Výpis A.2: Nastavení atributu třídy pro funkci a ukázka možné implementace funkce printf, v příkladu se využívá externí kód. viz sekce 6.10

Příloha B

Tabulka mapování datových typů

Typ C	**Typ C#**	**Velikost (B)**	**Přesnost na číslice**
char	char	2	
signed char	char	2	
unsigned char	char	2	
short	Int16	2	
signed short	Int16	2	
unsigned short	UInt16	2	
int	Int32	4	
unsigned int	UInt32	4	
long	Int64	8	
unsigned long	UInt64	8	
long long	Int64	8	
unsigned long long	UInt64	8	
float	float	4	∼6-9
double	double	8	∼15-17
long double	decimal	16	∼28-29
bool	bool	1	
size_t	UInt64	8	

Tabulka B.1: Mapování datových typů jazyka C na typy jazyka C#. Cílem bylo zachovat odpovídající hierarchie rozsahů s využitím dostupných typů.

Příloha C

Testovací sada

```
int fibbonacci(int n) {
   if(n == 0){
      return 0;
   }
   if(n == 1) {
      return 1;
   }
  return (fibbonacci(n-1) + fibbonacci(n-2));
}

int main() {
   int n = 10;

   for(int i = 0;i<n;i++) {
      fibbonacci(20);
   }

   return 0;
}
```

Výpis C.1: FIB1: Implementace rekurzivního výpočtu fibonacci v jazyce C, použito i pro C++

```
int main() {
  int n = 10000;

  int t1 = 0, t2 = 1;
  int nextTerm = t1 + t2;

  for (int i = 3; i <= n; ++i)
  {
  t1 = t2;
  t2 = nextTerm;
  nextTerm = t1 + t2;
  }

  return 0;
}
```

Výpis C.2: FIB2: Implementace iterativního výpočtu fibonacci v jazyce C, použito i pro C++

```c
void swap(int *a, int *b)
{
  int temp = *a;
  *a = *b;
  *b = temp;
}
void selectionSort(int arr[], int n)
{
  int i, j, min_element;
  for (i = 0; i < n-1; i++)
  {
    min_element = i;
    for (j = i+1; j < n; j++) { if (arr[j] < arr[min_element]) { min_element = j; } }
  swap(&arr[min_element], &arr[i]);
  }
}
int main()
{
  int arr[] = { 499, 498, 497, 496, 495, 494, 493, 492, 491, 490, 489, 488, 487, 486,
      485, 484, 483, 482, 481, 480, 479, 478, 477, 476, 475, 474, 473, 472, 471, 470,
      469, 468, 467, 466, 465, 464, 463, 462, 461, 460, 459, 458, 457, 456, 455, 454,
      453, 452, 451, 450, 449, 448, 447, 446, 445, 444, 443, 442, 441, 440, 439, 438,
      437, 436, 435, 434, 433, 432, 431, 430, 429, 428, 427, 426, 425, 424, 423, 422,
      421, 420, 419, 418, 417, 416, 415, 414, 413, 412, 411, 410, 409, 408, 407, 406,
      405, 404, 403, 402, 401, 400, 399, 398, 397, 396, 395, 394, 393, 392, 391, 390,
      389, 388, 387, 386, 385, 384, 383, 382, 381, 380, 379, 378, 377, 376, 375, 374,
      373, 372, 371, 370, 369, 368, 367, 366, 365, 364, 363, 362, 361, 360, 359, 358,
      357, 356, 355, 354, 353, 352, 351, 350, 349, 348, 347, 346, 345, 344, 343, 342,
      341, 340, 339, 338, 337, 336, 335, 334, 333, 332, 331, 330, 329, 328, 327, 326,
      325, 324, 323, 322, 321, 320, 319, 318, 317, 316, 315, 314, 313, 312, 311, 310,
      309, 308, 307, 306, 305, 304, 303, 302, 301, 300, 299, 298, 297, 296, 295, 294,
      293, 292, 291, 290, 289, 288, 287, 286, 285, 284, 283, 282, 281, 280, 279, 278,
      277, 276, 275, 274, 273, 272, 271, 270, 269, 268, 267, 266, 265, 264, 263, 262,
      261, 260, 259, 258, 257, 256, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246,
      245, 244, 243, 242, 241, 240, 239, 238, 237, 236, 235, 234, 233, 232, 231, 230,
      229, 228, 227, 226, 225, 224, 223, 222, 221, 220, 219, 218, 217, 216, 215, 214,
      213, 212, 211, 210, 209, 208, 207, 206, 205, 204, 203, 202, 201, 200, 199, 198,
      197, 196, 195, 194, 193, 192, 191, 190, 189, 188, 187, 186, 185, 184, 183, 182,
      181, 180, 179, 178, 177, 176, 175, 174, 173, 172, 171, 170, 169, 168, 167, 166,
      165, 164, 163, 162, 161, 160, 159, 158, 157, 156, 155, 154, 153, 152, 151, 150,
      149, 148, 147, 146, 145, 144, 143, 142, 141, 140, 139, 138, 137, 136, 135, 134,
      133, 132, 131, 130, 129, 128, 127, 126, 125, 124, 123, 122, 121, 120, 119, 118,
      117, 116, 115, 114, 113, 112, 111, 110, 109, 108, 107, 106, 105, 104, 103, 102,
      101, 100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82,
       81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62,
      61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42,
      41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22,
      21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
  int size = sizeof(arr)/sizeof(arr[0]);
  selectionSort(arr, size);
  return 0;
}
```

Výpis C.3: SEL1: Implementace algoritmu selection sort v jazyce C, použito i pro C++.

```c
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x) { return mid; }
        if (arr[mid] > x) { return binarySearch(arr, l, mid - 1, x); }
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}

int main()
{
    int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
        20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
        40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
        60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
        80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
        100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115,
        116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131,
        132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147,
        148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163,
        164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179,
        180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195,
        196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211,
        212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227,
        228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243,
        244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259,
        260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275,
        276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291,
        292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307,
        308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323,
        324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339,
        340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355,
        356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371,
        372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387,
        388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403,
        404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419,
        420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435,
        436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451,
        452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467,
        468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483,
        484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499};
    int n = sizeof(arr) / sizeof(arr[0]);
    for (int i = 0; i < n; i++){
        binarySearch(arr, 0, n - 1, i);
    }
    return 0;
}
```

Výpis C.4: BIN1: Implementace rekurzivního algoritmu binary search v jazyce C, použito i pro C++. Hledání pro každý prvek pole.

```
int max(int a, int b) { return (a > b) ? a : b; }
int knapSack(int W, int wt[], int val[], int n)
{
    if (n == 0 || W == 0) { return 0; }
    if (wt[n - 1] > W) { return knapSack(W, wt, val, n - 1); }
    else {
        return max(
            val[n - 1]
                + knapSack(W - wt[n - 1],
                           wt, val, n - 1),
            knapSack(W, wt, val, n - 1));
    }
}

int main()
{
    int val[] = { 60, 100, 120, 49, 34, 67, 140, 43, 67, 3, 89, 54, 21, 80, 134, 56};
    int wt[] = { 10, 20, 30, 2, 12, 4, 49, 7, 17, 30, 23, 45, 34, 12, 34, 24};
    int W = 1000;
    int n = sizeof(val) / sizeof(val[0]);
    knapSack(W, wt, val, n);
    return 0;
}
```

Výpis C.5: KNAP1: Implementace rekurzivního algoritmu knapsack v jazyce C, použito i pro C++.

Příloha D

Obsah paměťového média

Zde je popsán obsah přiloženého paměťového média.

- Technická zpráva ve formátu PDF a zdrojové soubory pro její přeložení ve formátu LaTeX.
- Implementace, součástí je projekt spustitelný ve Visual Studio 2022, instrukce k přeložení WebAssembly aplikace a spuštění Blazor a WPF aplikace
- Testy využívané pro výkonnostní testování
- Příklady spustitelné v interpretu
- Přeložená WebAssembly aplikace
- Obecný readme soubor s přehledem média a projektu