VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INFORMAČNÍCH SYSTÉMŮ
DEPARTMENT OF INFORMATION SYSTEMS

STANDARDNÍ KNIHOVNY A PREPROCESOR JAZYKA C PRO WWW PROHLÍŽEČ

STANDARD LIBRARIES AND C PREPROCESSOR FOR WEB BROWSER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **LUDĚK BURDA**
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 bakalářské práce

Student: **Burda Luděk**
Program: Informační technologie
Název: **Standardní knihovny a preprocesor jazyka C pro WWW prohlížeč**
Standard C Language Preprocessor and Libraries for WWW Browser
Kategorie: Překladače
Zadání:

1. Studujte jazyky pro programování pro WWW prohlížeč (Javascript, Typescript, ...). Studujte podrobně standardní knihovny a preprocessing jazyka C.
2. Navrhněte nejvhodnější a největší podmnožinu standardních knihoven jazyka C, co má smysl podpořit v prohlížeči.
3. Tuto podporu, podle návrhu a dle doporučení a po konzultaci s vedoucím, implementujte, jak na úrovni C pro překladač v prohlížeči, tak na úrovni prohlížeče. Stejně tak implementujte, po dohodě s vedoucím, preprocesor pro jazyk C.
4. Výsledek otestujte v několika prohlížečích jak na rychlost, tak na spotřebu paměti, atd. Zaměřte se na zpracování výrazů v preprocesoru a na rychlost vybraných knihovních funkcí.
5. Zhodnoťte přínos své práce a případné možnosti rozšíření.

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
- Dále dle doporučení vedoucího

Pro udělení zápočtu za první semestr je požadováno:

- První 2 body a část bodu 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: 11. května 2022
Datum schválení: 18. října 2021

Abstrakt

Tato technická zpráva popisuje návrh i implementaci preprocessoru pro jazyk C v jazyce javascript bez využití vzdáleného serveru pro zpracování zdrojového kódu. Dále se zabývá implementací podmnožiny funkcí ze standardních knihoven jazyka C v jazyce javascript a jejich paměťovou a časovou efektivitou. Také je implementován paměťový model jazyka C založený na datových typech jako nezbytný prvek pro funkčnost standardních knihoven.

Abstract

This technical report describes design and implementation of a C preprocessor in javascript without using a remote server for processing of the source code. Further it deals with implementation of a subset of standard libraries functions of the C language in javascript and its memory and time efficiency. In addition, a type based C memory model is implemented as necessary component for the standard functions.

Klíčová slova

C preprocessor, C preprocessing, standardní knihovny C, offline překladač C prohlížeč, javascript

Keywords

C preprocessor, C preprocessing, standard libraries C, offline compiler C web browser, javascript

Citace

BURDA, Luděk. *Standardní knihovny a preprocesor jazyka C pro WWW prohlížeč.* Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Dr. Ing. Dušan Kolář

Standardní knihovny a preprocesor jazyka C pro WWW prohlížeč

Prohlášení

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

. . . .
Luděk Burda
18. května 2022

Poděkování

Děkuji svému vedoucímu práce, panu doc. Dr. Ing. Dušanu Kolářovi, za pozitivní přístup k tématu, nadšení pro věc a odborné rady při překonávání problémů.

Obsah

1 Úvod **2**

2 Fáze překladu **3**
- 2.1 Fáze 1 . . . 3
- 2.2 Fáze 2 . . . 4
- 2.3 Fáze 3 . . . 4
- 2.4 Fáze 4-8 . . . 6

3 Preprocesor **8**
- 3.1 Direktivy pro kontrolu maker . . . 8
- 3.2 Direktiva pro správu hlavičkových souborů . . . 10
- 3.3 Podmínkové direktivy . . . 10
- 3.4 Diagnostické direktivy . . . 11
- 3.5 Informativní direktiva . . . 11

4 Analýza technologií a návrh řešení **12**

5 Implementace **14**
- 5.1 Preprocesor . . . 14
- 5.2 Podpora standardních knihoven . . . 21

6 Testování **29**
- 6.1 Zátěžové testy . . . 29
- 6.2 Jednotkové testy . . . 34
- 6.3 Agregační testy . . . 34

7 Závěr **36**

Literatura **37**

A Obsah přiloženého média **38**

B Návod pro zprovoznění aplikace **39**

C Pravidla gramatiky **40**

Kapitola 1

Úvod

Zpracování zdrojového kódu napsaného v jazyce C probíhá ve dvou hlavních fázích – preprocesování a samotný překlad. C preprocesor je procesor maker využívaný překladačem jazyka C pro transformaci programu před samotnou kompilací. Preprocesor zpracuje veškeré direktivy definované standardem ISO/IEC 9899 [1], a řádky s direktivami (začínající znakem mřížky) ze zdrojového kódu odstraní. Výstup preprocesoru je zároveň vstupem překladače.[5]

Cílem této bakalářské práce (dále jenom práce) je návrh a implementace zmíněného preprocesoru v prostředí webového prohlížeče. Preprocesor se řídí pravidly gramatiky preprocesoru C a v implementaci této práce vykonává první 4 fáze překladu.

Tato práce je určena ke kooperaci s prací dalšího studenta, který implementuje zmíněný překladač přijímající výstup preprocesoru. Cílem celého díla je implementace interpretu jazyka C fungující ve webovém prohlížeči bez nutnosti komunikace se vzdáleným serverem, zpravidla nazývaným jazykový server, který zpracuje zdrojový kód a prohlížeč pouze zobrazí výsledek. Motivace je tedy vytvoření nástroje použitelného při edukaci studentů, kteří budou mít možnost na rozpracovaném zdrojovém kódu pracovat i cestou domů vlakem bez připojení k internetu.

Součástí práce je také implementace podmnožiny funkcí ze standardních knihoven jazyka C v jazyce použitém pro implementaci celého překladače, aby bylo možné jednodušeji a rychleji zpracovávat tyto funkce volané ve zdrojovém kódu a nebylo nutné celé knihovny kompilovat. Pro tyto knihovny je implementován také zjednodušený paměťový model jazyka C založený na typovém přístupu do paměti. Dále jsou implementované funkce testovány na rychlost a spotřebu paměti.

Část práce, preprocesor, je veřejně dostupná k vyzkoušení na webových stránkách školy `https://www.stud.fit.vutbr.cz/~xburda13/cpp/`.

[1] `https://www.iso.org/standard/29237.html`

Kapitola 2

Fáze překladu

Tato kapitola je výtahem informací ze zdrojů [3] a [5]. Zdrojový kód jazyka C je zpracován v následujících fázích daných specifikovaným pořadím. V samotné implementaci však může docházet k prolínání akcí jednotlivých fází, za předpokladu že zůstanou zachovány definované funkce a vlastnosti celého programu.

2.1 Fáze 1

1. Soubor se zdrojovým kódem, obecně vzato textový soubor s vícebytovým kódováním jako například UTF-8, je zpracován po jednotlivých bytech, jenž jsou následně podle konkrétní implementace namapovány na samostatné znaky zdrojové sady znaků. Zejména dle rozdílných operačních systémů různě definované symboly konce řádků jsou nahrazeny jednotným znakem konce řádku. Zdrojová sada znaků je vícebytová sada znaků, jenž zahrnuje základní sadu znaků jakožto jednobytovou podmnožinu, skládající se z následujících 96 znaků:

   (a) 5 symbolů bílých znaků (mezera, horizontální tabulátor, vertikální tabulátor, `form feed`, znak nového řádku)

   (b) 10 znaků číslic od 0 do 9

   (c) 52 písmen abecedy, velkých i malých

   (d) 29 znaků interpunkce _ [ ] ( ) < > % : ; . ? * + - / ^ | ! = , \“’

2. Trigrafické sekvence 2.1 jsou nahrazeny korespondujícími jednoznakovými symboly.

Trigrafické sekvence

Zdrojový kód jazyka C může být napsán v jakékoliv 8bitové sadě znaků, jenž zahrnují invariantní znakovou sadu definovanou standardem ISO 646:1983[1] (včetně znaků, které nezahrnuje tabulka ASCII). Nicméně, některé operátory a interpunkční znaky jazyka C vyžadují znaky, které ISO 646 znaková sada neobsahuje, konkrétně: {, }, [,], #, \, ^, |, ~. Aby bylo možné využívat kódování znaků, v nichž zmíněné symboly neexistují, je využito alternativní reprezentace těchto symbolů pomocí kombinací tří znaků kombatibilních se

[1] `https://www.iso.org/standard/4776.html`

standardem ISO 646, které jsou následně interpretovány jako jednotlivé znaky nezahrnuty ve znakové sadě daného standardu. Následující tříznakové skupiny 2.1 (trigrafické sekvence) jsou zpracovány před tím, než překladač rozpoznává řetězcové konstanty či komentáře, a každý výskyt trigrafické sekvence je nahrazen korespondujícím primárním znakem:

Primární znak	Trigrafická sekvence
{	??<
}	??>
[	??(
]	??)
#	??=
\	??/
^	??'
|	??!
~	??-

Tabulka 2.1: Trigrafické sekvence a jejich znaková reprezentace

Vzhledem k tomu, že trigrafické sekvence jsou ze zdrojového kódu zpracovány jako první, může dojít k neočekávanému chování vzhledem k transformacím kódu popsané v podkapitolách 2.2 a 2.3, a je tedy dobré tuto možnost brát na vědomí a předejít tím chybám z nepozornosti.

2.2 Fáze 2

1. U každého výskytu znaku zpětného lomítka na konci řádku (přímo následovaného znakem konce řádku) jsou oba tyto znaky - zpětné lomítko i znak konce řádku - smazány, čímž jsou tyto dva fyzicky oddělené řádky zkombinovány do jednoho logického řádku. Tato akce je jednoprůchodová, dvě zpětná lomítka následována dvěma znaky konce řádku (dohromady 3 fyzické řádky) nejsou tedy zkombinovány do jednoho logického řádku.

2. Pokud je po provedení kroku 1 neprázdný soubor se zdrojovým kódem ukončený jinak než znakem konce řádku, není chování překladače definováno.

2.3 Fáze 3

1. Soubor zdrojového kódu je rozložen na komentáře 3, sekvence bílých znaků a následující tokeny preprocesoru:

   (a) názvy hlavičkových souborů ve formátu <stdio.h> nebo "muj_soubor.h"

   (b) identifikátory 3

   (c) preprocesingová čísla 3

   (d) znakové a řetězcové konstanty 3

   (e) víceznakové operátory 3 a interpunkční znaky 1d

   (f) individuální nebílé znaky, které nespadají do žádné z předchozích kategorií

2. Každý komentář je nahrazen znakem mezery.
3. Znaky nového řádku jsou zachovány, a dle konkrétní implementace mohou být sekvence bílých znaků nahrazeny vždy jedním znakem mezery.

   Každý token preprocesoru je obecně vzato brán jako nejdelší možná sekvence znaků, jenž může představovat validní token preprocesoru, i přestože tento přístup může zapříčinit selhání následné analýzy. Takový přístup se běžně nazývá v angličtině 'maximal munch'. Jedinou výjimkou je zpracování tokenu preprocesoru typu název hlavičkového souboru, ke kterému dochází pouze za direktivou include, zde je preferována možnost validního názvu hlavičkového souboru před maximální možnou délkou tokenu.

   Příklad situace kdy preprocesor označí zdrojový kód jako chybný: `0xE+var`. Tuto sekvenci znaků preprocesor vyhodnotí jako jeden token proprocesorového čísla 3, neboť `E+` je jedna z validních kombinací znaků vyskytující se v sekvenci znaků preprocesorového čísla a ihned za ním následující název proměnné. Celé preprocesorové číslo v této podobě však není validním tokenem pro překladač jazyka C, který následně hlásí chybu. Předejít této situaci lze například oddělením operátoru sčítání mezerami.

Komentáře

Komentáře slouží svým způsobem jako dokumentace integrovaná přímo ve zdrojovém kódu. Veškeré komentáře jsou překladačem ignorovány a nemají vliv na běh programu, jsou využívány výhradně jako poznámky pro lidské osoby, které čtou zdrojový kód. V jazyce C existují dva druhy komentářů, každý s vlastním pojmenováním.

Syntaxe	Označení
/* komentář */	styl C, případně "víceřádkový"
// komentář	styl C++, případně "jednořádkový"

Tabulka 2.2: Typy komentářů jazyka C a jejich syntaxe

Víceřádkové

Víceřádkové komentáře jsou zpravidla využívány k zakomentování větších bloků textu ve zdrojovém kódu, případně menších fragmentů samotného zdrojového kódu. Je s nimi samozřejmě možné vytvořit i komentář na jednom řádku. Chceme-li zakomentovat text ve zdrojovém kódu, stačí daný text obalit pomocí /* a */. Komentáře stylu C zpraví překladač o tom, že veškerý text mezi /* a */ má být kompletně ignorován. Je také zvykem označovat dokumentační bloky ve zdrojovém kódu vložením textu mezi /** a */, kde druhá hvězdička v pořadí je vnímána jako normální součást komentáře. Jediná výjimka, kdy dvojice znaků /* neoznačuje začátek komentáře, je případ řetězcových a znakových konstant 3. Obsah komentářů je zpracován pouze za účelem nalezení dvojice znaků */ ukončující daný komentář. Víceřádkové komentáře nemohou být vnořené do sebe, první dvojice znaků */ vždy komentář ukončí.

Jednořádkové

Jednořádkové komentáře jsou obvykle používány pro zakomentování jednoho řádku zdrojového kódu, je ale přirozeně možné poskládat z několika jednořádkových komentářů komentář víceřádkový. Zakomentování začíná vždy od dvojice znaků // a končí znakem nového řádku, veškerý text uvnitř komentáře slouží stejnému účelu jako víceřádkové komentáře a překladačem je vyjma hledání znaku konce řádku zcela ignorován. Stejně jako u předchozí situace dvojice znaků // indikuje začátek komentáře vždy, až na případ, kdy se nachází v řetězcové či znakové konstantě. Ze svojí podstaty, jednořádkové komentáře mohou být zanořovány.

Identifikátory

Identifikátor je libovolně dlouhá sekvence znaků, kde přípustnými znaky jsou myšleny číslice, podtržítka, malá písmena a velká písmena základní abecedy. Překladač při rozlišování idenifikátorů rozeznává velká a malá písmena. Indentifikátor může začínat libovolným přípustným znakem s výjimkou číslice.

Preprocesingová čísla

Tato kategorie tokenů má od běžných čísel významné odlišnosti. Jednotlivá preprocesingová čísla začínají číslicí, která může být předcházena tečkou. Poté následuje libovolně dlouhá sekvence písmen, číslic, podtržítek, teček a tzv exponentů (dvojznakové sekvence ‘e+’, ‘e-’, ‘E+’, ‘E-’, ‘p+’, ‘p-’, ‘P+’, a ‘P-’).

Řetězcové literály

Řetězcovými literály jsou označeny sekvence libovolných znaků ohraničené uvozovkami. Zatímco řetězcové konstanty jsou ohraničené dvojitými uvozovkami, konstanty znakové jsou ohraničeny uvozovkami jednoduchými. Pro zahrnutí hraničního znaku do konkrétní konstanty, je potřeba provést escape sekvenci tohoto znaku (\' případně \"). Znakové konstanty delší než právě jeden znak jsou překladačem interpretovány dle konkrétní implementace kompilátoru, standardem toto chování definované není.

Víceznakové operátory

Varianty základních víceznakových operátorů jsou v tabulce 2.3. Jejich kompletní seznam lze nalézt zde[2].

2.4 Fáze 4-8

2.4.1 Fáze 4

1. Je spuštěn samotný preprocesor. 3

[2] https://www.tutorialspoint.com/cprogramming/c$_{o}perators.htm$

Operátor	Název
++	inkrementace
--	dekrementace
==	je rovno
!=	není rovno
>=	větší nebo rovno
<=	menší nebo rovno
&&	logický AND operátor
||	logický OR operátor
<<	binární posun vlevo
>>	binární posun vpravo

Tabulka 2.3: Hlavní víceznakové operátory složené ze znaků interpunkce 1d.

2. Každý hlavičkový soubor zahrnutý do zdrojového kódu direktivou #include 3.2 prochází rekurzivně fázemi překladu 1-4 2.
3. Na konci této fáze jsou odstraněny všechny preprocesorové direktivy ze zdrojového kódu.

2.4.2 Fáze 5

Veškeré znaky escape sekvence v řetězcových literálech jsou převedeny ze zdrojové sady znaků do sady znaků určené pro spuštění programu. Není-li znak reprezentovaný escape sekvencí součástí zmíněné sady znaků, není výsledek definován konkrétně, pouze je zaručeno, že jím bude znak nenulové délky.

2.4.3 Fáze 6

Sousedící řetězcové literály (stejného typu) jsou zřetězeny do jednoho řetězcového literálu (daného typu).

2.4.4 Fáze 7

Je spuštěna samotná kompilace zdrojového kódu. Tokeny jsou analyzovány syntakticky a sémanticky a následně přeloženy jako překladové jednotky.

2.4.5 Fáze 8

Vytvořené překladové jednotky a knihovní komponenty potřebné pro splnění požadovaných externích referencí jsou propojeny a reprezentovány jako obraz programu, jenž obsahuje informace vyžadované pro spuštění programu v daném prostředí operačního systému.

Kapitola 3

Preprocesor

V této kapitole byly informace čerpány a parafrázovány ze zdrojů [3] a [2]. Direktivy preprocesoru řídí chování preprocesoru. Každá direktiva zabírá právě jeden řádek s následujícím formátem:

- znak mřížky - #
- instrukce pro preprocesor (define, undef, include, if, ifdef, ifndef, else, elif, endif, line, error, warning, pragma)
- volitelné argumenty instrukce
- konec řádku

Je povolen také speciální případ nulové direktivy, kdy je znak mřížky následován koncem řádku. Tato direktiva nemá žádný efekt ve zdrojovém kódu.

Preprocesor zdrojový kód skenuje na direktivy sekvenčně, každá z nich tedy začíná platit až od řádku, na němž se direktiva nachází.

3.1 Direktivy pro kontrolu maker

Makrem označujeme fragment kódu, kterému byl programátorem přiřazen název. Název musí být validním identifikátorem 3. Pro lepší přehlednost byla však zavedena nepovinná konvence psaní názvů maker výhradně velkými písmeny. Při jakémkoliv výskytu tohoto názvu je poté ve zdrojovém kódu samotný název nahrazen obsahem makra identifikovaného tímto názvem, toto chování nazýváme expanze makra. Existují dva druhy maker, objektová 3.1.1 a funkční makra 3.1.1. Makra jsou vytvářena pomocí direktivy #define 3.1.1. Pokud se v těle makra nachází název sebe sama, není tento název při expanzi makra a následném skenování na výskyt maker nahrazen vlastním tělem. Tímto chováním je zabráněno nekonečné rekurzi. V těle funkčního makra je také umožněno používat dva specifické operátory: ořetězení # 3.1.1 a zřetězení ## 3.1.1.

3.1.1 #define

Objektová makra

Nejčastěji jsou taková makra využívána pro pojmenování různých programátorem definovaných konstant. Při vytváření objektového makra je direktiva #define následována vlastním

názvem makra a poté sekvencí validních tokenů 2.3, již označujeme jako tělo makra, nepodléhající žádným syntaktickým či sémantickým pravidlům zdrojového kódu C. Vyskytne-li se po definici makra ve zdrojovém kódu zmíněný název makra jako samostatný token, je tento název nahrazen tělem daného makra.

Funkční makra

Makra lze definovat také s vlastními parametry. V takových makrech ihned za názvem makra následují klasické závorky, v nichž se nacházejí identifikátory jednotlivých parametrů makra oddělené čárkami. Funkční makra pro expanzi vyžadují výskyt názvu makra následovaný závorkami s argumenty. V opačném případě nedochází k nahrazení názvu makra jeho tělem. Při expanzi jsou nahrazeny výskyty názvů parametrů v těle makra odpovídajícími argumenty. Speciálním případem funkčního makra je makro variadické 3.1.1. Toto makro přijímá předem nedefinovaný počet argumentů.

Ořetězení

Argument funkčního makra může být převeden do podoby řetězcové konstanty pomocí operátoru ořetězení # (anglicky stringize). Pokud se v těle makra nachází parametr, jemuž předchází operátor #, preprocesor vytvoří textovou podobu samotného argumentu ve zdrojovém kódu a nahradí parametr řetězcovou konstantou obsahující tuto textovou podobu. Na rozdíl od běžného nahrazování parametrů v tomhle případě není provedena expanze případného makra v argumentu.

Zřetězení

Operátor zřetězení je využíván v situaci, kdy je žádoucí lexikální zřetězení dvou tokenů. Výsledek zřetězení musí být opět validním preprocesingovým tokenem. V takovém případě při expanzi makra jsou výsledkem operace zřetězení nahrazeny tyto dva tokeny, i mezi nimi nacházející se operátor ##.

Variadická makra

Stejně jako u běžných funkcí, funkční makro může být definováno s volitelným počtem přijímaných argumentů. Při invokaci makra, všechny tokeny v seznamu argumentů následující za posledním pojmenovaným argumentem se stávají variabilním argumentem. Sekvence těchto tokenů je nahrazena za všechny výskyty identifikátoru __VA_ARGS__ v těle makra.

Standardní předdefinovaná makra

V jazyce C existuje několik předdefinovaných objektových maker, lze je invokovat bez poskytnutí jejich definice. Je možné provést redefinici těchto maker, tím je však ztracena jejich původní funkcionalita.

- `__FILE__` tělo makra obsahuje řetězcovou konstantu nesoucí název aktuálně zpracovávaného souboru
- `__LINE__` tělo makra obsahuje celočíselnou konstantu informující o aktuálním řádku, na němž se invokace makra nachází

- `__DATE__` tělo makra obsahuje řetězcovou konstantu, která popisuje aktuální datum při běhu preprocesoru ve formátu "Dec 24 1998"a vždy má délku 11 znaků
- `__TIME__` tělo makra obsahuje řetězcovou konstantu, která popisuje aktuální čas při kterém bylo toto makro invokováno ve formátu "13:14:15"a vždy má délku 8 znaků
- `__STDC__` toto makro expanduje na celočíselnou konstantu 1, čímž značí, že překladač odpovídá standardu ISO C TODO_REF
- `__STDC_VERSION__` tělo makra obsahuje dlouhou celočíselnou konstantu popisující verzi standardu C ve formátu yyyymmL, kde yyyy a mm jsou rokem a měsícem vydání verze standardu
- `__STDC_HOSTED__` tělo makra obsahuje celočíselnou konstantu 1 za předpokladu, že cílem překladače je hostované prostředí (obsahující kompletní sadu dostupných standardních knihoven jazyka C)
- `__cplusplus` toto makro je definováno, pokud je používán kompilátor jazyka C++
- `__OBJC__` toto makro je definováno, pokud je používán kompilátor objektově orientovaného jazyku Objective-C (rozšíření jazyka C)
- `__ASSEMBLER__` toto makro obsahuje celočíselnou konstantu 1 za předpokladu, že preprocesor zpracovává jazyk symbolických instrukcí

3.1.2 #undef

Libovolná makra mohou být vymazána direktivou #undef. Za instrukcí direktivy se nachází identifikátor makra, jenž má být odstraněno ze seznamu definovaných maker. Po odstranění makra může být identifikátor opět redefinován jako makro použitím direktivy define 3.1.1. Nová definice nemusí mít žádnou souvislost se starou definicí.

3.2 Direktiva pro správu hlavičkových souborů

Hlavičkový soubor obsahuje deklarace v jazyce C a definice maker určené pro sdílení napříč soubory se zdrojovým kódem. Do zdrojového kódu je zahrnut pomocí direktivy #include. Zkopírování obsahu hlavičkového souboru do zdrojového kódu přinese stejný výsledek jako zahrnutí tohoto hlavičkového souboru pomocí direktivy #include. Za instrukcí direktivy se může nacházet pouze validní název hlavičkového souboru 2.3.

3.3 Podmínkové direktivy

Podmínkové direktivy dávají preprocesoru informaci o tom, které kusy zdrojového kódu budou odeslány kompilátoru ke zpracování a které ne. Preprocesorové podmínky vyhodnocují aritmetické výrazy, případně zda existuje definice makra podle identifikátoru. Může vyhodnocovat kombinaci obojího pomocí speciálního operátoru `defined` 3.3. Logika těchto direktiv funguje obdobně jako u běžných podmínek ve zdrojovém kódu. Podmínkové direktivy se mohou libovolně zanořovat, je ale nutné, aby každá vrstva, včetně první, byla ukončena direktivou #endif.

#if, #elif, #ifdef, #ifndef a #else

Direktiva #if testuje hodnotu aritmetický operací, podmínka je vyhodnocena jako pravdivá, pokud je výsledkem nenulová hodnota. Speciální operátor `defined`, jenž je možné používat pouze v podmínkových direktivách, přijímá právě jeden argument, jímž je identifikátor, a výsledkem operace je celočíselná hodnota 1 pokud existuje makro definované s daným identifikátorem a 0 pokud neexistuje. Direktiva #ifdef se používá výhradně pro kontrolu existence makra a podmínka je vyhodnocena taktéž za předpokladu, že makro existuje. Její obdoba #ifndef funguje stejným způsobem, pouze s opačnou pravdivostní hodnotou. Direktiva #elif se může objevit pouze v posloupnosti za jednou ze dvou zmíněných předchozích direktiv, případně za další direktivou #elif. Mezi nimi však nesmí dojít k ukončení podmínkového bloku pomocí direktivy #endif. K vyhodnocení podmínky direktivy #elif dochází pouze za předpokladu, že předchozí podmínka podmínkové direktivy byla vyhodnocena jako nepravda. To stejné platí pro direktivu #else, s rozdílem, že zde není přítomna podmínka a zdrojový kód následovaný za touto direktivou je ve zmíněném případě rovnou interpretován jako kód, který bude odeslán překladači pro další zpracování.

3.4 Diagnostické direktivy

Direktiva #error způsobí vyvolání fatální chyby preprocesorem. Tokeny následující za touto direktivou jsou poté využity jako text chybové hlášky. Nejčastější použití této direktivy je ve spojení s podmínkovými direktivami 3.3. Druhou diagnostickou direktivou je #warning, jež funguje obdobným způsobem jako direktiva #error, s rozdílem, že místo vyvolání fatální chyby je vyvoláno pouze varování nebránící následnému zpracovávání zdrojového kódu.

3.5 Informativní direktiva

Preprocesor uchovává informace o poloze každého tokenu zdrojového kódu, sestávající z názvu souboru a řádku na kterém se nachází. Tyto informace však mohou být změněny pomocí direktivy #line. Za touto direktivou se musí nacházet celočíselná konstanta. Tuto konstantu preprocesor bude chápat jako aktuální řádek ve zdrojovém kódu a následující řádky se budou od této konstanty odvíjet také. Je-li následujícím tokenem řetězcová konstanta, direktiva změní informaci o názvu souboru zdrojového kódu na obsah dané konstanty. Místo konstant se mohou vyskytovat identifikátory definovaných maker, ta však musí být expandována na zmíněnýmé konstanty. Tato direktiva také upravuje výsledky maker __LINE__ a __FILE__ 3.1.1.

Kapitola 4

Analýza technologií a návrh řešení

V této kapitole jsou popsány technologie použitelné k implementaci nástroje a abstraktní návrh celého řešení. Podrobnější popis samotného využití daných nástrojů a samotné implementace se nachází v následující kapitole 5. Celý nástroj je implementován v programovacím jazyce javascript s využitím balíčkového systému Node.js[1]. Pro zprovoznění nástroje čistě v prohlížeči je poté využit balík modulů Webpack[2], který sjednotí větší počet modulů samotného projektu do jednoho souboru zdrojového kódu.

4.0.1 Preprocesor

Základní logika preprocesoru vychází z gramatiky 4.0.2 vytvořené podle definice vycházející ze standardu C. V první fázi návrhu byl pro analýzu programu uvažován nástroj ANTLR, jenž je schopný vygenerovat lexer a parser jazyka podle zadané gramatiky. Následně však vyšlo najevo, že pro potřeby preprocesoru jazyka C je tento nástroj zbytečně komplikovaný a je jednodušší provést implementaci komponent manuálně.

Samotný preprocesor přijímá zdrojový kód v textové podobě a následně proběhnou tři průchody celého programu. Nejprve dochází k lexikální analýze, logickému spojení řádků končících zpětným lomítkem s následujícím řádkem a následné tokenizaci lexémů, tato tokenizace však není finální. Ve druhém průchodu jsou tyto tokeny tokenizovány opětovně a vznikají již tokeny pro preprocesor. V posledním průchodu je analyzována sémantika již zmíněných tokenů a zároveň dochází k jejich interpretaci a dále k transformaci zdrojového kódu podle pravidel 4. fáze překladu 2.4.1. Transformace však probíhá na úrovni tokenů a tyto výsledné tokeny jsou nakonec předány překladači.

4.0.2 Gramatika a LL tabulka

Gramatika C pracuje s již definovanými pojmy využívanými jako terminály. Těmito terminály jsou:

- id - identifikátory 3
- cppnumber - preprocesingové číslo 3
- string - řetězcová konstanta 3
- char - znaková konstanta 3

[1] `https://nodejs.org/en/`
[2] `https://webpack.js.org/`

- op - víceznakové operátory 2.3
- punct - ostatní znaky interpunkce 1d
- other - nebílé znaky bez syntaktického významu v preprocesoru 2.3
- hdrname - název hlavičkového souboru 2.3
- expr - výraz v preprocesoru (pozn.: zahrnujíc define operátor) 4.0.2
- eps - epsilon
- eol - znak konce řádku
- direktivy preprocesoru (define, undef, include, if, ifdef, ifndef, else, elif, endif, line, error, warning, pragma)

Dále jsou v gramatice následující definované neterminály:

- BASE - startovní neterminál
- DIRECTIVE - neterminál popisující syntaxi direktiv
- FILE - pomocný neterminál pro volitelnost názvu souboru v direktivě
- NONDIRECTIVE a SRCNONTERMS - dvojice neterminálů zobecňující libovolnou posloupnost zbylých tokenů pro preprocesor

Výrazy

Výrazy v preprocesoru se mohou skládat z následujících prvků:

- Celočíselné konstanty
- Znakové konstanty, jenž jsou interpretovány stejně jako v interpretu jazyka C
- Aritmetické operace sčítání, odčítání, násobení, dělení, bitové operace, posuny, porovnávací operace a logické operace
- Makra
- Speciální operátor `defined`
- Identifikátory

4.0.3 Standardní knihovny

Obecný návrh podpory těchto knihoven spočívá v implementaci samostatné třídy pro každou knihovnu zvlášť s následnou integrací knihovních maker a knihovních funkcí do překladače v případě požadavku na zahrnutí dané knihovny do zdrojového kódu.

Pro jednotlivé knihovny lze často nalézt balíčky v systému Node.js kopírující návrh i funkčnost některých konkrétních funkcí ze standardních knihoven jazyka C. Zbylé knihovní funkce a jejich režijní náklady jsou manuálně naprogramovány v holém javascriptu.

Většina těchto funkcí využívá paměťový model C společně s ukazateli. Kvůli podpoře těchto funkcí byla implementována simulace tohoto modelu popsaná v podsekci 5.2.1.

Kapitola 5

Implementace

V této kapitole je detailně popsána logika implementace celého nástroje a jsou zde zmíněny veškeré konkrétní využité externí knihovny a balíčky.

Pro zprovoznění nástroje je nezbytně nutný balíček Webpack sjednocující všechny použité balíčky systému Node.js do jediného souboru zdrojového kódu, který je následně spouštěn v prohlížeči.

5.1 Preprocesor

V této sekci je popsána implementace procesů jednotlivých fází překladu a funkce preprocesoru. Pro spuštění transformací kódu před preprocesováním i samotného preprocesingu je volána hlavní funkce programu `preprocess()` přijímající jediný argument - řetězcovou konstantu obsahující zdrojový kód programu. Tato funkce je volána překladačem po deklaraci potřebných datových struktur.

Tato hlavní funkce následně volá metody postupně transformující zdrojový kód, které mohou při kritické chybě zastavit celý proces překladu a vypsat chybovou hlášku na simulovanou konzoli.

První fáze překladu (přepis trigrafických sekvencí 2.1) není implementována z důvodu malé využitelnosti.

5.1.1 Fáze 2

Ve fázi 2 probíhá první průchod zdrojového kódu reprezentovaného řetězcovou konstantou a prvotní tokenizace metodou tokenize() přijímající řetězcovou konstantu jako jediný argument.

Samotný token je objektem javascriptu s následující definovanou strukturou, předpokládanými datovými typy s nimiž je token vytvářen a významem jednotlivých atributů:

```
Token {
    // samotný textový obsah tokenu
    string content,

    // typ tokenu pro preprocesor
    string type, // může nabývat hodnot 'identifier','headername',
                 // 'cppnumber','character','string','punctuator',
                 // 'special_punctuators','concatenation',
```

```
                    // 'unknown symbols','eol','eof'

        // délka textového obsahu tokenu
        int length,

        // řádek v souboru zdrojového kódu, v němž se token nachází
        int line,

        // pořadí znaku, kterým začíná textová část tokenu na svém řádku
        int index,

        // označení spec. typu tokenu popsaného v implementaci preprocesoru
        bool special,

        // využito pouze v případě tokenu pocházejícího z expanze makra,
        // je v něm v pořadí od nejhlubšího zanoření uložen seznam Tokenů
        // expandovaných maker
        [Token] parentSymbol,

        // název souboru zdrojového kódu
        string module
    }
```

Před samotným průchodem je vytvořeno počítadlo řádků a dva seznamy znaků rozlišující interpunkční znaménka, jenž nemají žádný význam v sémantice preprocesoru a ta, která jsou v dalším průchodu analyzována důsledněji (dále jen speciální interpunkční znaky).

V průchodu je kontrolován znak po znaku s případnou následnou aplikací regulárních výrazů pro rozpoznávání konkrétních typů tokenů. Tyto znaky jsou získávány v cyklu přístupem do řetězcové konstanty pomocí indexu začínajícího nulou. Tokeny jsou vytvářeny obalující funkcí `createToken()`, které jsou předány argumenty odpovídající struktuře tokenu.

Vytořené Tokeny jsou následně přidávány do výstupního pole Tokenů v pořadí, v jakém za sebou následují ve zdrojovém kódu. Na konci metody prvního průchodu je na konec výstupního pole přidán speciální token označující konec souboru, případně je před něj vložen Token reprezentující konec řádku, pokud se tam již nenchází. Nezáleží tedy na tom, zda je zdrojový kód ukončen znakem konce řádku, či nikoliv.

Pro každý analyzovaný znak probíhá posloupnost kontrol, přičemž při každém nalezeném Tokenu je zvýšena hodnota indexování zdrojového kódu a je provedena opět posloupnost kontrol pro znak přímo následující za daným Tokenem:

- Je provedena kontrola bílých znaků (s výjimkou znaků konce řádku) regexem `/^[^\S\r\n]+/`. Regulární výraz přijímá libovolně dlouhou nenulovou posloupnost bílých znaků a v interní reprezentaci je označuje pouze jako bílý znak ('whitespace') o délce této posloupnosti.

- Pokud se nachází znak ve skupině interpunkčních znaků mající sémantický význam pro preprocesor, jsou uloženy jako speciální interpunkční Token s typem ('special_punctuators').

- V případě, že je znakem zpětné lomítko, pokusíme se regulárním výrazem `/^\\\s*?\n/` zachytit shodu takovou, že mezi zpětným lomítkem a znakem konce řádku se nacházejí pouze bílé znaky. V takovém případě je jednoduše pouze zvýšeno indexování zdrojového kódu o délku této posloupnosti znaků. Tímto je logicky spojen řádek zdrojového kódu s řádkem následujícím. V případě, že regulární výraz nenalezne shodu je znak zpětného lomítka uložen jako speciální interpunkční znak.

- Je-li nalezen znak konce řádku, je po přidání adekvátního Tokenu ('eol') do výstupního pole v rámci optimalizace odříznut tento zpracovaný logický řádek zdrojového kódu z původní řetězcové konstanty reprezentující celý zdrojový kód a je restartováno indexování tohoto řetězce na nulu.

- Probíhá snaha o nalezení shody regulárním výrazem `/^[_A-Za-z][_A-Za-z\d]*/`, jenž je předpisem pro validní indentifikátor jazyka C. V běžném případě je pouze vytvořen Token a procedura pokračuje normálním způsobem. Speciálním případem je však identifikátor 'include', který je klíčovým slovem preprocesoru a indikuje možnost, že je tento řádek direktivou #include. V této situaci je potřeba označit následující Token adekvátním typem, tedy ('headername'). Je ověřeno, že předcházejícím Tokenem je znak mřížky a poté zkontrolována správnost formátu názvu hlavičkového souboru pomocí dvojice regulárních výrazů `/^<.*?>/` a `/^".*?"/`. Pokud však formát neodpovídá, je pokračováno v analýze běžným způsobem a Token s názvem hlavičkového souboru nevzniká.

- Je hledána shoda regexu `/^(\.)?\d([_A-Za-z\d\.]|([E|P|e|p][\+|\-]))*/` popisujícího formát preprocesingového čísla ('cppnumber').

- Je zkontrolován výskyt analyzovaného znaku v seznamu interpunkčních znamének bez sémantického významu pro preprocesor ('punctuator').

- Pokud znak neprošel žádnou z výše uvedených kontrol, nemá tento znak žádný sémantický význam ve zdrojovém kódu jazyka C ('unknown').

5.1.2 Fáze 3

Zde probíhá druhý průchod zdrojovým kódem již reprezentovaného polem Tokenů metodou `retokenize()` přijímající jako jediný argument zmíněné pole Tokenů.

Před začátkem průchodu je vytvořeno pole znaků interpunkce, které mohou být začínajícím znakem víceznakového operátoru jazyka C (dále speciální interpunkční znaky). Pro každý řádek zdrojového kódu je vytvořeno pole, jenž je následně plněno tokeny v pořadí, v jakém za sebou následují ve zdrojovém kódu. Každé takové pole Tokenů je při nalezení konce řádku přidáno do výstupního pole polí Tokenů reprezentujícího celý zdrojový kód.

Pro každý analyzovaný Token opět probíhá posloupnost kontrol řídících se typem Tokenu:

- V případě že se zpracovávaný token nachází v seznamu speciálních interpunkčních znaků, jsou analyzovány i následující tokeny a je rozpoznán případný víceznakový operátor nebo operátor tří teček využívaný v definici variadických funkcí a maker.

- Při nalezení tokenu konce řádku, je aktuální pole tokenů daného řádku přidáno do výsledného pole polí tokenů a následně je vyprázdněno, aby do něj mohly být opět přidávány tokeny nového řádku.

- Token znaku mřížky je analyzován z důvodu existence operátoru konkatenace 3.1.1 ('concatenation') nacházející se ve funkčních makrech.
- Rozpoznání řetězcových ('string') a znakových ('character') konstant. Jsou akceptovány i neukončené konstanty, označeny jako ('corrupted_string') a ('corrupted_character').
- Tokeny začínající lomítkem jsou kontrolovány na oba typy komentářů a v případně shody jsou nahrazeny jednou mezerou. Neukončené víceřádkové komentáře způsobí fatální chybu ukončující překlad programu.
- Pokud není zachycen ani jeden z těchto případů, je token beze změny přidán do pole tokenů k další analýze.

5.1.3 Fáze 4

Poslední průchod se již věnuje funkcionalitě samotného preprocesoru. Jsou rozlišovány dvě hlavní situace - zda se na začátku řádku nachází znak mřížky a je tedy tento řádek interpretován jako direktiva preprocesoru, či nenachází. Zdrojový kód je analyzován sekvenčně, a při analýze je zároveň generován transformovaný výstupní zdrojový kód ve stejném formátu jako vstupní zdrojový kód. Z důvodu existence podmínkových direktiv 3.3 je implementován přepínač (dále pouze přepínač) nesoucí informaci o tom, zda jsou analyzované řádky zdrojového kódu interpretovány běžným způsobem (výchozí hodnota přepínače), a nebo byl výraz poslední podmínkové direktivy vyhodnocen jako nepravda a zdrojový kód je analyzován pouze za účelem nalezení další podmínkové direktivy, která tento mód, v němž jsou ignorovány fragmenty zdrojového kódu, přepne opět na běžný způsob analýzy. Před započetím posledního průchodu jsou připravena pole pro výstupní zdrojový kód a pro uchovávání aktuálního stavu podmínkových direktiv.

Direktiva

```
// objekt objektových maker
defines {
    identifikátor: [Token],
    ...
}

// objekt funkčních a variadických maker
definesF/definesFV {
    identifikátor: {
        'parameters': [Token],
        'content': [Token]
    },
    ...
}
```

- define - Makra jsou interně reprezentována jako objekt dvou různých formátů 5.1.3. Je rozlišován případ funkčního makra a objektového makra dle prezence znaku otevřené závorky bezprostředně za identifikátorem makra.

- funkční makro - Nejdříve jsou zkoumány parametry definovaného makra, analýze pomáhá binární přepínač, pomocí něj preprocesor ví, zda posledním tokenem byl oddělovač parametrů, tedy čárka, nebo samotný parametr, a podle toho následně provádí kontroly přicházejících tokenů. Dále také hledá mezi parametry výskyt operátoru tří teček, díky němuž je poté zřejmé, že se jedná o variadické makro, a je s ním dále tímto způsobem nakládáno. Po načtení parametrů jsou již pouze postupně přidány tokeny těla daného makra do adekvátního objektu.
- objektové makro - Proběhne načtení tokenů následujících za identifikátorem makra a jejich uložení do adekvátního objektu.

- undef - Je smazán záznam z objektu, v němž se identifikátor makra, jenž je jediným argumentem této direktivy, nachází.

- ifdef/ifndef - Direktiva ifndef se liší od direktivy ifdef pouze zapnutím přepínače invertujícího pravdivostní hodnotu zpracovávaného výrazu. Direktiva je nejprve přidána do LIFO fronty uchovávající aktuální stav zanoření podmínkových direktiv. Dále je vyhodnoceno, zda argument direktivy, identifikátor makra, je definován, či není, a je podle této pravdivostní hodnoty rozhodnuto o způsobu následného analyzování zdrojového kódu pomocí přepínače zmíněného v podsekci5.1.3.

- if - Je podstoupen stejný proces jako u předchozí direktivy, pravdivostní hodnota se ale vyhodnocuje z výrazu popsaného v sekci 4.0.2.

- else - V interní reprezentaci invertuje pravdivostní hodnotu poslední zpracované podmínkové direktivy a nastaví přepínač určující způsob zpracovávání zdrojového kódu podle stavu výše zmíněné LIFO fronty.

- elif - Funguje stejně jako za sebou jdoucí direktiva else následovaná direktivou if.

- endif - Uzavírá aktuální vrstvu podmínkových direktiv. Vyjme z LIFO fronty nejnovější prvek a následně podle stavu této fronty vyhodnotí způsob, jakým bude analýza programu pokračovat. Toto vyhodnocení je prostá detekce nepravdivé hodnoty v LIFO frontě.

- error/warning - Obě direktivy fungují stejně, pouze direktiva error ukončí celý preprocessing, direktiva warning jen vypíše zprávu na výstup. Tokeny následující za direktivou jsou konvertovány do jedné řetězcové konstanty, která je následně odeslána na standardní výstup.

- line - Jsou analyzovány argumenty direktivy, nejprve jsou expandována veškerá makra vyskytující se za samotnou direktivou. Následně se musí jako první argument vyskytovat za direktivou celočíselná konstanta. Její hodnota je následně vnímána jako aktuální řádek zdrojového kódu, a je od té doby zdrojový kód řádkován od této hodnoty. Dále se volitelně může nacházet za celočíselnou hodnotou také řetězcová konstanta, jenž je od chvíle zavolání této direktivy považována za název souboru zdrojového kódu.

- include - Jediným argumentem je speciální typ tokenu 'hdrname', popisující název hlavičkového souboru, jenž má být zahrnut do zdrojového kódu. Pokud je argumentem název standardní knihovny, je zavolána inicializační funkce právě této knihovny a je dále možné využívat definovaná makra a standardní funkce této knihovny.

Ne-direktiva

V případě, že je přepínač upravující typ analýzy zdrojového kódu nastaven na pouhé vyhledávání podmínkových direktiv, je řádek přeskočen. V opačném případě je inicializováno prázdné pole, do nějž jsou postupně vkládány tokeny řádku analyzovaného zdrojového kódu. Pokud aktuálně analyzovaný token není typu 'identifikátor', je pouze přidán do zmíněného pole, jinak probíhá postupná kontrola výskytu ve 3 různých skupinách maker, jenž se mohou objevovat ve zdrojovém kódu. Pro expanzi libovolného typu je vždy volána tatáž funkce `replaceMacro()`5.1.3, přijímající jako první argument pole tokenů, u nichž má dojít k expanzi libovolného typu makra a jako druhý argument seznam rekurzivně již volaných maker. Její návratová hodnota je pole tokenů, tyto tokeny jsou konkatenovány do výstupního pole daného řádku zdrojového kódu.

- Speciální makro - Speciálními makry jsou myšlena předdefinovaná makra popsána v podsekci 3.1.1. Při jejich výskytu je funkci funkci `replaceMacro()` předán pouze první argument ve formě jednoprvkového pole obsahující identifikátor daného makra, neboť u těchto maker nemůže dojít k nekonečné rekurzi.

- Objektové makro - Objektová makra se liší od speciálních pouze předáním i druhého argumentu, v tomto případě taktéž jednoprvkové pole obsahující identifikátor daného makra.

- Funkční/variadické makro - Pokud se za identifikátorem nenachází znak otevřené závorky, je token identifikátoru přidán do výstupního pole řádku a probíhá analýza dalšího tokenu. Pokud se zde znak otevřené závorky nachází, je inicializováno prázdné pole, jenž je postupně plněno přicházejícími tokeny. Při invokaci těchto typů maker je možné argumenty oddělovat i znaky konce řádku, z tohoto důvodu byl implementován speciální cyklus ovlivňující i stav nejvíce vnějšího cyklu procházejícího celým zdrojovým kódem. V případě, že je započat nový řádek dříve, než dojde k načtení všech případných argumentů makra, může se vyskytovat na daném novém řádku opět libovolná direktiva preprocesoru. Tuto možnost algoritmus také podporuje. Je-li nalezen znak konce souboru, znamená to, že volání makra nebylo uzavřeno závorkou, je ohlášena kritická chyba a dochází k ukončení preprocesování. Při úspěšném uzavření seznamu argumentů makra závorkou je naplněné pole analyzovaných tokenů předáno zmíněné funkci `replaceMacro()` spolu s prázdným polem jako druhý argument - rekurze je ošetřena uvnitř samotné funkce. Je také přidán přepínač jako speciální třetí argument přetěžující tuto funkci, který zabezpečuje možnou expanzi maker vyskytujících se v argumentech makra před jejich nahrazením do těla makra. Chování těchto typů maker má jedno málo známé specifikum, při nahrazování je za danou expanzi uměle přidána mezera, pokud se tam sama nevyskytuje. Tato mezera je však speciální, neboť pokud by byla tato expanze s následujícími tokeny celá předána jako argument funkčnímu makru, který nad tímto argumentem provede operaci ořetězení3.1.1, nebude zmíněná mezera v této řetězcové konstantě obsažena. Tato situace je v algoritmu ošetřena přidáním speciálního atributu tokenu této uměle přidané mezery.

replaceMacro(input,recursionIds,prescan)

input - pole tokenů určené k analýze a expanzi vyskytujících se maker
recursionIds - pole již jednou expandovaných identifikátorů
prescan - přepínač ošetřující skenování argumentů funkčních maker na expanze maker

Funkce prochází vstupní pole a u každého tokenu typu 'identifikátor' kontroluje výskyt v různých skupinách definovaných maker. Před započetím průchodu vstupního pole tokenů je inicializováno prázdné pole, jenž je postupně plněno zpracovanými tokeny. U každé expanze makra je také uchováván stav historie expanzí z důvodu podpory přesných informativních hlášek uživateli v případě nalezení chyby. Historie tokenů je uložena v atributu 'parentSymbol'. Jsou rozlišovány stejné 3 typy maker k expanzi jako u popisu implementace direktivy define 5.1.3:

- Speciální makro 3.1.1 - identifikátor je předán jako jediný argument funkci `expandSpecialMacro()`, v němž jsou předem definována těla všech těchto maker. Funkce vrací pole tokenů, které je následně konkatenováno do výstupního pole.
- Objektové makro - Proběhne rekurzivní volání funkce `replaceMacro()` s argumentem pole tokenů těla adekvátního makra. Druhý argument je konkatenací doposud propagovaných rekurzivních identifikátorů a samotného identifikátoru makra, které je expandováno. Výstup funkce je přidán do výstupního pole.
- Funkční/variadické makro - Podle typu objektu, v němž se vyskytuje identifikátor makra, je nastaven přepínač indikující variadické makro. Dále jsou inicializována dvě prázdná pole, jedno pro pole argumentů, kde každý argument je polem tokenů. Druhé pole pro opakované načítání tokenů jednotlivých argumentů. V této části funkce je již garantován správný počet závorek a není třeba kontrolovat jejich ukončení. Je stále však třeba držet přehled o kontextu zanoření kvůli rozdělení jednotlivých argumentů makra. Jednotlivé tokeny jsou přidávány do pole pro tokeny argumentu, dokud nedojde k nalezení operátoru čárka v situaci, kdy je nulové zanoření závorek. Pokud je zapnutý přepínač indikující variadické makro, je kontrolován počet dosud načtených argumentů, a pokud dosáhne počtu předpokládaných argumentů, je operátor čárky ignorován a veškeré tokeny až po konec invokace variadického makra jsou interpretovány jako jediný, poslední argument. Po nalezení argumentů je zavolána funkce `replaceMacroF()`5.1.3 se seznamem argumentů jako první argument, tělem tohoto makra jako druhý argument, rekurzivními identifikátory jako třetí arugment a identifikátor samotného makra jako poslední argument. Výstup z této funkce je přidán do výstupního pole tokenů.

replaceMacroF(input,recursionIds,prescan)

argumentList - pole polí tokenů, každé reprezentující jeden argument makra
theMacroObject - objekt invokovaného makra nesoucí potřebná data pro expanzi 5.1.3
recursionIds - pole již jednou expandovaných identifikátorů
macroID - identifikátor expandovaného makra

Ve funkci je postupně procházeno tělo makra. Před samotným průchodem je inicializováno prázdné pole určené k ukládání výstupní sekvence tokenů. Není-li analyzovaný token typu 'identifikátor', je beze změny předán do výstupního pole. Pokud identifikátorem je, je zkontrolováno, zda následující token obsahuje operátor zřetězení 3.1.1. Tímto rozlišujeme dva významné případy.

Ne-li následující token operátorem zřetězení, začne funkce prohledávat seznam parametrů makra a hledá shodu se zkoumaným identifikátorem, přidávajíc tento identifikátor do výstupního pole tokenů pokud nalezena není. V opačném případě je testováno, zda předchozím tokenem není operátor ořetězení 3.1.1, který by převedl obsah odpovídajícího argumentu makra na řetězcovou konstantu. Při neúspěchu této kontroly je obsah odpovídajícího

argumentu předán funkci `replaceMacro()` 5.1.3, a až její výstup je nahrazen za původní identifikátor argumentu makra. Tímto krokem je ošetřeno předskenování argumentů makra.

Druhým významným případem je sitauce, kdy se nachází za identifikátorem operátor zřetězení. V takové chvíli jsou zkoumány dva tokeny obklopující tento operátor. Je-li první z těchto tokenů identifikátorem obsaženém v seznamu parametrů, dojde před zřetězením nejprve k nahrazení parametru argumentem, ne však k předskenování argumentu na výskyt maker. Poslední token argumentu je poté označen jako token ke zřetězení. U druhého z tokenů obklopujících operátor zřetězení dochází ke stejnému procesu, pouze případný první token argumentu je určen ke zřetězení. Zbylé tokeny argumentů jsou následně přidány do výstupního pole beze změny. Není-li některý z původních tokenů zároveň parametrem, je určen ke zřetězení tento samotný token. Zřetězení následně proběhne zřetězením samotných obsahů obou tokenů určených ke zřetězení. Takto zřetězená řetězcová konstanta je následně předána funkcím obsluhující první a druhý průchod zdrojového kódu. Pokud výstupem této funkce je právě jeden token, zřetězení bylo úspěšné. V opačném případě je oznámena chyba operátoru zřetězení.

Speciálním případem u operátoru zřetězení je u variadického makra. Pokud je požadováno zřetězení tokenu čárky na prvním místě a speciálního parametru `__VA_ARGS__`, je zkoumáno, zda existuje nějaký argument nahrazující tento název parametru. Neexistuje-li, nedochází při expanzi celého makra k vypsání této čárky. Pokud však existuje, je nahrazen normálním způsob a nedochází ke zřetězení se zmíněnou čárkou.

Na konci celého procesu zpracování tohoto typu makra je celé výstupní pole předáno funkci `replaceMacro()` pro možnost invokace například nově vzniklých volání dalších funkčních maker a až výstup této funkce je vrácen.

5.2 Podpora standardních knihoven

Standardní knihovny jazyka C nabízejí makra, definice datových typů a funkce pro práci s textovými řetězci, matematickými výrazy a výpočty, zpracovávání vstupu a výstupu, správu paměti a několik dalších služeb operačního systému. Rozhraní pro práci s těmito knihovny je zprostředkováno ve formě množiny hlavičkových souborů. Podmnožina knihoven, jenž dává smysl podporovat v této práci, byla zvolena podle účelu celého díla 1. Konkrétní knihovní funkce, jenž v této podmnožině nejsou zahrnuty, jsou společně s důvody pro jejich absenci popsány v sekci implementace 5.2.9. Definice maker a funkcí byly převzaty z [1]:

5.2.1 Paměťový model

Valná většina knihovních funkcí využívá paměťový model jazyka C na úrovni ukazatelů. Tento paměťový model byl tedy nezbytným prvkem pro možnost vytvoření podpory standardních knihoven.

Vzhledem k tomu, že precizní návrh paměťového modelu založeného na různé interpretaci jednotlivých bytů by byl velmi pracný a je nad rámec této práce, byl zvolen typový model paměti.

Princip tohoto modelu je založen na trojici objektů v jazyce javascript, kde každý reprezentuje určitou vrstvu paměti:

- nameSpaceTable - Tento objekt slouží k ukládání informací o adresách uživatelem definovaných proměnných alokovaných zásobníkovým způsobem. Každý záznam také nese informaci o tom, zda je konkrétní proměnná inicializovaná, či nikoliv.

```
nameSpaceTable[id] = {
    initialized: bool,
    address: string
}
```

- SaddressTable - Zde jsou uloženy informace o proměnných alokovaných zásobníkovým způsobem. Záznamy jsou rozlišovány unikátním klíčem, jímž je právě adresa v paměti konkrétní proměnné. Skladba atributů těchto záznamů může mít mírné odlišnosti odvíjející se od datového typu proměnné, jejíž obsah je zde zaznamenán. Základní struktura objektu vypadá následovně:

```
SaddressTable[Saddress] = {
    name: bool,          // název proměnné
    length: int,         // délka|indikátor inicializace
    type: string,        // datový typ proměnné
    size: int,           // velikost alokovaného prostoru
    value: string|array // hodnota proměnné
}
```

  Platí pro všechny základní datové typy včetně ukazatele. Staticky alokované pole je však odlišeno typem 'array'. V takovém případě je v záznamu také typ jednotlivých prvků pole, je využit atribut 'length', jenž nese v případě inicializovaného základního datového typu celočíselnou hodnotu 1, případně hodnotu 0, pokud proměnná inicializována není. Atribut 'value' je poté javascriptovým polem hodnot adekvátních datovému typu jednotlivých prvků. Nakonec jsou také přidány atributy 'offset' a 'dereferred', nezbytné pro korektní vyhodnocování ukazatelových výrazů předávaných standardním funkcím. Pokud je datovým typem ukazatel, atribut 'value' nese hodnotu paměti uložené v poslední ze tří tabulek. Pokud je tento ukazatel neinicializovaný (tzv. wild pointer), hodnotou je nulová celočíselná konstanta. Způsob ukládání datových typů definovaných specifickými standardními knihovnami jsou popsány v následujících podsekcích.

- HaddressTable - V tomto objektu jsou uloženy dynamicky alokované proměnné. Základní struktura těchto objektu je podobná jako u minulé tabulky, má však několik základních odlišností:

```
HaddressTable[Haddress] = {
    own: string(Haddress), // vlastní adresa záznamu
    source: string,      // adr. ukazatele na tento blok paměti
    dereferred: bool,    // informace pro vnitřní reprezentaci
                            nutná pro správnou interpretaci při
                            vyhodnocování ukazatelových výrazů
    length: int,         // množství alokovaných jednotek
                            konkrétního datového typu
    type: string,        // konkrétní datový typ, jenž se v
                            tomto bloku paměti nachází
    size: int,           // celková alokovaná velikost
```

```
        value: [],        // hodnota je vždy polem konkrétních
                             hodnot uložených v alokovaném
                             prostoru
        offset: 0         // ukazatel nemusí ukazovat na začátek
                             této alokované paměti
    }
```

Je-li typem opět ukazatel, jsou přidány dva další atributy. Atribut 'on' popisující datový typ hodnoty, na kterou tento ukazatel ukazuje a atribut 'addresses', jenž je paralelním nosičem informace o tom, na jakých adresách jsou uloženy hodnoty tohoto záznamu. Samotnými hodnotami jsou poté objekty tabulky HaddressTable.

Paměťový model využívá aplikační rámec sestávající z množiny funkcí okomentovaných ve zdrojovém kódu v sekci 'STDLIB OVERHEAD'. Nejvýznamnější z nich je funkce `getArguments()`, využívaná v každé knihovní funkci přijímající argumenty. Tato funkce přijímá pole dvojic, každá pro jeden požadovaný argument. Dvojice sestává z vlastní hodnoty předaného argumentu a požadovaného datového typu. Tímto je ošetřena kontrola datových typů argumentů předávaných funkci. Návratovou hodnotou je pole se stejnou velikostí jako pole předané této funkci argumentem. Nachází se zde dereferencované hodnoty ukazatelů, případně samotné objekty z jedné ze tří tabulek, je-li požadován datový typ ukazatel.

Celý proces zpracování výrazu argumentu probíhá rozlišením, zda se v hodnotě argumentu nachází identifikátor, či nikoliv. Přímé předávání hodnot je možné pouze pomocí nevýrazových konstant základních datových typů. Je-li však v hodnotě obsažen identifikátor, je spuštěn proces analýzy ukazatelového výrazu. Ten začíná vyhledáním identifikátoru, nahrazením znakem 'x', uložením indexu kde se toto nahrazení nachází a následnému postupnému vynořování, při němž je zpracováván ukazatelový výraz se zohledněním precedence[1]. Při každé zpracované operaci je v analyzovaném řetězci nahrazen tento podřetězec obsahující znaky celé operace opět univerzálním znakem 'x' a probíhá opět stejný proces analýzy. Celý algoritmus podporuje výskyt právě jednoho identifikátoru. Přičítání a odečítání ukazatele a indexace pole je možná pouze pomocí celočíselných konstant. Podporovanými operacemi jsou:

- `++ --` - prefixová/postfixová inkrementace/dekremenace
- `[]` - indexace pole
- `*` - dereference ukazatele
- `&` - reference ukazatele
- `+ -` - odečítání/sčítání celočíselných konstant od/s ukazatele/m

5.2.2 stdio.h

Pro simulaci souborového systému byla pro tuto standardní knihovnu implementována interní reprezentace souborového systému spočívající v objektu, kde se název souboru váže k obsahu souboru:

```
fileSystemTable[filename] = fileContent
```

[1] `https://en.cppreference.com/w/c/language/operator_precedence`

Dále byla implementována pomocná tabulka, jenž asociuje určitý simulovaný soubor k jednotlivým proudům dat typu 'FILE' z důvodu zachování správného kontextu při úpravách souborů, k nimž je nějaký datový typ 'FILE' navázán.

```
filStreamTable[filename] = [addresses]
```

Samotný datový typ 'FILE' má následovnou strukturu v objektu paměti (zobrazeny pouze odlišnosti od struktury standardního základního datového typu):

```
FILE = {
    type: 'FILE',       //
    size: 216,          //
    read: bool,         // přepínač povolující čtení zdrojového souboru
    write: bool,        // přepínač povolující zápis do zdroj. souboru
    offsetLock: bool,   // přepínač pro kontrolu, zda je možné při
                           zápisu přepisovat soubor, či pouze přidávat
                           znaky na konec souboru
    eof: bool,          // přepínač indikující, zda daný proud narazil
                           na konec souboru
    offset: int,        // aktuální poloha jezdce v obsahu souboru
    filename: string    // název souboru, k němuž se datový proud váže
}
```

Pro implementaci knihovních funkcí `printf()`, `sprintf()` a `fprintf()` byl využit node.js balíček `printf`[2], jenž v kombinaci s obalující funkcí `printfWrap()` kopíruje funkcionalitu zmíněných knihovních funkcí. Při čerpání hodnot z proměnných však jsou podporovány pouze formátovací řetězce `%u` pro získání adresy dané proměnné, `%d` pro celočíselnou konstantu, `%s` pro řetězcovou konstantu a `%f` pro konstantu desetinného čísla. Dále pro získání dat ze standardního vstupu pro funkce `getchar() a gets()` byl využit node.js balíček `prompt-sync`[3]. Ostatní funkce byly implementovány manuálně.

Seznam implementovaných prvků knihoven:

- Knihovní makra
  - NULL
    Makro reprezentující nulovou celočíselnou konstantu otypovanou na ukazatel typu `void`.

  - EOF
    Makro indikující konec vstupního proudu dat. Je reprezentován zápornou celočíselnou konstanou -1 uzavřenou v závorkách.

  - FOPEN_MAX
    Maximální počet souborů, jenž nástroj garantuje, že mohou být v jednu chvíli souběžně otevřeny.

[2] https://www.npmjs.com/package/printf
[3] https://www.npmjs.com/package/prompt-sync

  - SEEK_SET, SEEK_CUR, SEEK_END
    Makra využívaná ve funkci `fseek()` pro lokalizování konkrétní pozice v souboru. Nesou celočíslené hodnoty `0, 1 a 2`.

- Knihovní funkce

  - implementovány všechny s výjimkou níže zmíněných

Následující prvky knihovny nebyly implementovány vůbec:

- `fread(), fwrite()` - bajtová práce s pamětí - paměťový model nepodporuje
- `setbuf(), setvbuf(), fflush()` - vyrovnávací paměť datových proudů není implementována
- `vsprintf(), vprintf()` - hlavičkový soubor `stdarg.h` není implementován, tudíž ani datový typ `va_list`
- `fscanf(), scanf(), sscanf()` - náročná práce s rozborem jednotlivých argumentů daných funkcí
- `clearerr(), ferror(), perror()` - chyby jsou z edukativního důvodu hlášeny přímo při jejich výskytu, není tedy potřeba uchovávat tyto informace dále
- V poslední řadě nejsou podporována makra `stdio,stdout,stderr` z důvodu komplikované integrace do systému funkcí standardních knihoven a makra využívaná výhradně funkcemi bez podpory v tomto nástroji.

5.2.3 stdlib.h

Z této knihovny jsou všechny funkce implementovány manuálně, některé konkrétní však mají kvůli chybějící implementaci interpretu jazyka C upravenou funkcionalitu. Konkrétně funkce `malloc() a calloc()` přijímají kromě definovaných argumentů také dva argumenty navíc, specifikující datový typ a název proměnné, do nichž je vytvořený ukazatel rovnou také uložen.

Seznam implementovaných prvků knihoven:

- Knihovní makra

  - NULL
    Makro reprezentující nulovou celočíselnou konstantu otypovanou na ukazatel typu `void`.

  - RAND_MAX
    Makro nesoucí maximální možnou hodnotu, jenž může vrátit funkce `rand()`.

- Knihovní funkce

  - implementovány všechny s výjimkou níže zmíněných

Nepodporovanými prvky knihovny jsou:

- `wctomb(), wcstombs(), mbtowc(), mbstowcs()` - funkce zhodnoceny jako příliš málo významné pro edukativní účely nástroje
- `bsearch(), qsort()` - bajtová práce s pamětí - paměťový model nepodporuje
- `system(), getenv()` - systémové funkce nelze podporovat v prohlížeči
- `atexit()` - paměťový model nepodporuje datový typ ukazatele na funkci

5.2.4 string.h

V případě funkcí `memchr,strchr,strrchr,strpbrk,strstr` není vracen ukazatel na paměť předanou argumentem, nýbrž pouze řetězcová konstanta reprezentující tento úsek paměti. Důvodem je chyba v návrhu paměťového modelu. Dále funkce `strcoll()` pouze volá unkci `strcmp`, neboť hlavičkový soubor `locale.h` není podporován.

Seznam implementovaných prvků knihoven:

- Knihovní makra
  - NULL
    Makro reprezentující nulovou celočíselnou konstantu otypovanou na ukazatel typu `void`.
- Knihovní funkce
  - implementovány všechny s výjimkou níže zmíněných

Nepodporovanými prvky knihovny jsou:

- `strerror()` - hlavičkový soubor `errno.h` není podporován
- `strtok()` - příliš komplexní přístup pro uchovávání různých stavů při volání dané funkce

5.2.5 ctype.h

Návrh této knihovny se mírně liší od ostatních knihoven. Vyjma dvou konverzních funkcí se všechny funkce generují vytvořením instance třídy, při čemž je konstruktoru předán název konkrétního pole, v němž je vyhledán výskyt znaku předaného argumentem funkce vytvořené při instanciaci třídy. Podporovány jsou všechny funkce této knihovny.

5.2.6 math.h

Pro implementaci velké části knihovních funkcí byl využit node.js balíček `mathjs`[4]. Konkrétně také pro implementaci funkce `frexp` byl využit balíček `locutus`[5] a pro funkci `modf()` je to balíček `@stdlib/math-base-special-modf` [6]. Podporovány jsou veškeré složky této knihovny s výjimkou makra `HUGE_VAL`.

[4]https://www.npmjs.com/package/mathjs
[5]https://www.npmjs.com/package/locutus
[6]https://www.npmjs.com/package/@stdlib/math-base-special-modf

5.2.7 time.h

Speciálně pro tuto knihovnu byla přidána podpora datového typu 'struct' do aplikačního rámce simulace paměťového modelu5.2.1. Pro definice struktur je implementován objekt `structDefsTable` s následující strukturou:

```
structDefsTable[label] = {
    size: int,          // velikost celé struktury
    varNames: [string], // názvy proměnných v pevně daném pořadí
    types: [string]     // datové typy proměnných v pořadí
                           odpovídajícím předchozímu poli
}
```

Deklaraci struktury zprostředkovává funkce `createStruct()`. Tato funkce přijímá jako argumenty identifikátor proměnné, s níž bude struktura deklarována, štítek struktury, podle níž má být strukura vytvořena a přepínač, zda má být alokována staticky, či dynamicky.

Implementace většiny knihovních funkcí využívá node.js balíček `date-format-ms`[7]. Dále pro implementaci funkce `strftime` byl využit balíček `strftime`[8]. Z knihovny není podporována pouze funkce `clock()`, která potřebuje znát počet časových tiků centrální procesorové jednotky počítače a samotné makro `CLOCKS_PER_SEC` reprezentující tuto hodnotu.

5.2.8 Ostatní podporované knihovny

limits.h, float.h

Hlavičkové soubory obsahují pouze množinu maker, v implementaci přidaných do objektu reprezentující definovaná makra.

assert.h

Jediným makrem je zde makro `assert`, jeho tělo se odvíjí od existence definice makra `NDEBUG`. V případě že existuje, jsou argumenty makra `assert` ignorovány a expanduje pouze na nulovou celočíselnou konstantu. Pokud neexistuje, je vyhodnocen výraz předaný argumentem který při neúspěchu způsobí volání funkce `printf()` ze standardní knihovny `stdio.h` 5.2.2. Z tohoto důvodu dochází k zahrnutí této knihovny zároveň při zahrnutí knihovny `assert.h`.

5.2.9 Nepodporované knihovny

errno.h

Tato knihovna zahrnuje pouze deklaraci globální proměnné celočíselného typu s identifikátorem 'errno', jež je v různých případech chybových stavů libovolných knihovních funkcí nastavována na určitou hodnotu odpovídající konkrétnímu typu chyby. Vzhledem k motivaci této práce jsou chybové hlášky oznamovány uživateli vždy při výskytu libovolné chyby, není tedy `errno` pro tuto práci potřebné.

[7] https://www.npmjs.com/package/date-format-ms

[8] https://www.npmjs.com/package/strftime

locale.h

Knihovna definující lokalitou specifická nastavení systému byla shledána pro cíle této práce nepotřebnou.

setjmp.h

Záznamy o prostředí interpretovaného zdrojového kódu nejsou k dispozici.

signal.h

Chybějící podpora datového typu ukazatele na funkci znemožňuje implementaci hlavní knihovní funkce `signal()` a generovat signály pomocí druhé knihovní funkce `raise()` také není možné.

stdarg.h

Pro implementaci této knihovny, konkrétně systému maker zabezpečující aplikační rámec pro práci s variadickými funkcemi, potažmo jejich variadickými argumenty, by byla nutná spolupráce s interpretem jazyka C, který není aktuálně implementován.

Implementace této knihovny by v případě existence interpretu mohla být založena na uložení seznamu variadických argumentů do proměnné typu heterogenní struktury. Vzhledem k potřebě argumentu popisujícímu počet těchto argumentů by bylo možné přidávat prvky do struktury iterativně.

Kapitola 6

Testování

Testování nástroje bylo prováděno několika způsoby. V první fázi vývoje preprocesoru byla souběžně s přidáváním funkčních prvků také uchovávana vstupní data, jenž byla později využita pro agregační testy preprocesoru 6.3.

Dále při tvorbě aplikačního rámce pro standardní knihovny byl vytvořen ve zdrojovém kódu uměle přidaný obsah v simulovaném paměťovém modelu 5.2.1, jenž byl následně využíván při jednotkovém testování 6.2 jednotlivých knihovních funkcí.

V konečné fázi probíhalo testování preprocesoru i funkcí standardních knihoven zaměřené na detekci úniku paměti a na časovou efektivitu obou nástrojů 6.1.

6.1 Zátěžové testy

6.1.1 Paměť

Obsah této podsekce byl inspirován zdrojem [4]. Správa paměti v javascriptu funguje na principu "garbage collectingu", je tedy zabezpečován automaticky samotným interpretem javascriptu. Z podstaty aplikace založené na balíčkovém systému Node.js[1], neexistuje žádná jednoduchá a spolehlivá metoda na odhalení úniků paměti, neboť v množství zahrnovaných balíčků existují právě i takové, kterým samotným již nějaká paměť uniká, a je tedy složité odhalit úniky paměti ze zdrojového kódu vlastní aplikace. Nejlepším způsobem vyhnutí se paměťovým únikům je prevence prostřednictvím dobrých programovacích návyků[2], které byly při psaní zdrojového kódu dodržovány.

I s tímto přístup je však potřeba nějakým způsobem ověřit samotnou míru úniků paměti aplikace, aby ji bylo možno v praxi využívat. Za tímto účelem byly vytvořeny speciální extrémní situace ve zdrojovém kódu, kterým byla následně vystavena aplikace, zatímco na pozadí byl sledován a analyzován stav paměti.

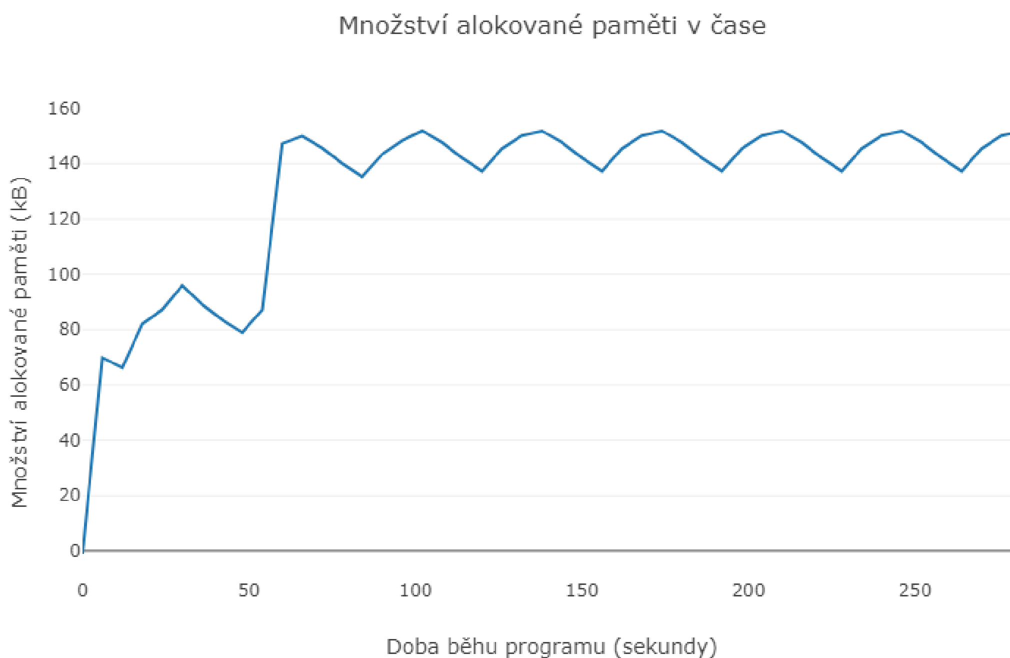
Nejprve probíhalo jednodušší testování konzolové verze aplikace s průběžným měřením alokované paměti. K tomuto byl využit node.js balíček[3]. Bylo spuštěno 54 iterací preprocesingu se sadou 6 různých jednotlivých vstupních dat. Po každé iteraci bylo zkontrolováno množství aktuálně alokované paměti. Z výsledku testů na obrázku 6.1 se zdá být spotřeba paměti od určité iterace stabilní, okolo hodnoty 150kB. Výkyvy v grafu jsou zapříčiněny

[1] https://nodejs.org/en/

[2] https://sematext.com/blog/nodejs-memory-leaks/#toc-how-to-avoid-memory-leaks-in-nodejs-applications-prevention-best-practices-11

[3] https://www.npmjs.com/package/node-memwatch-new

algoritmem správy paměti javascriptu, který vždy uvolní již nepotřebné bloky paměti až po určité době běhu programu.



Obrázek 6.1: Zátěžové testy v konzoli 1

Toto detailní měření paměti je však velmi časově náročné a není možné tímto způsobem provádět extrémní testovací scénáře.

Z tohoto důvodu byl zvolen odlišný přístup testování a byla napsána funkce `stressTest()`. V této funkci byla využita tatáž sada vstupních dat pro preprocesor, byl však měřen pouze stav alokované paměti na konci běhu programu. Opakovaně byla spouštěna aplikace pouze volající tuto funkci, vždy s větším počtem iterací programu. Výsledky těchto měření jsou zaznamenány v tabulce 6.1 a jsou zprůměrovanými hodnotami vždy z několika jednotek spuštění celého programu s danými parametry.

Počet iterací	Množství alokované paměti v kB	Doba běhu programu v s
10	187.37	0.03
100	303.6	0.13
1 000	337.88	0.31
1 000 000	340.09	174
100 000 000	340.58	16373

Tabulka 6.1: Množství paměti alokované preprocesorem

Tyto 6.1 testy vykazují konvergenci paměťové náročnosti programu v závislosti na množství iterací, tudíž se zdá že nedochází k únikům paměti. Je však potřeba provést i testy přímo v prohlížeči. Zde není podpora balíčku použitého při konzolových testech, je tedy nutné využít zabudovaný profilovač paměti v prohlížečích.

V prohlížečích byl spouštěn týž soubor vstupních dat pro preprocesor jako v konzoli. Záměrně je vynechán při profilování první okamžik načtení stránky, jenž zabírá vždy přibližně 10MB paměti a není relevantní pro testování samotné aplikace.

Počet iterací	Množství alokované paměti v kB	Doba běhu programu v s
10	207.5	0.01
100	258.8	0.095
1 000	348.6	0.38
1 000 000	348.36	173

Tabulka 6.2: Množství paměti alokované preprocesorem v prohlížeči Opera

Počet iterací	Množství alokované paměti v kB	Doba běhu programu v s
10	233.6	0.01
100	284.7	0.075
1 000	386	0.33
1 000 000	379.9	190

Tabulka 6.3: Množství paměti alokované preprocesorem v prohlížeči Chrome

Počet iterací	Množství alokované paměti v kB	Doba běhu programu v s
10	233.9	0.015
100	285	0.09
1 000	386.4	0.33
1 000 000	386.3	167.2

Tabulka 6.4: Množství paměti alokované preprocesorem v prohlížeči Edge

Testy v prohlížeči vykazují totožný trend jako testy konzolové. Lze tedy předpokládat, že v aplikaci nedochází k únikům paměti. Pro vizualizaci výsledku jsou všechna 4 měření spojena do jednoho grafu 6.2.

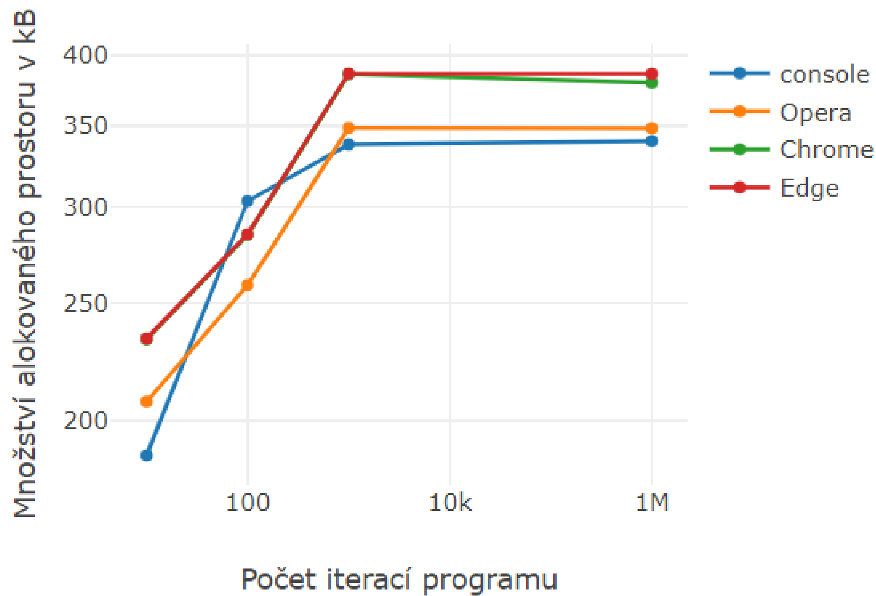
6.1.2 Čas

Preprocesor

Měření v předchozí podsekci bylo zaměřeno primárně na zátěž paměti, zároveň však byla zaznamenávána i doba trvání běhu programu. Z výsledků zobrazených v grafu 6.3 je zřejmé, že rozdíly v rychlosti mezi jednotlivými prohlížeči jsou zanedbatelné a při větším množství iterací jsou tyto rychlosti srovnatelné s rychlostí aplikace spuštěné z konzole.

Zpracování výrazů

Testování zpracování výrazů probíhalo v konzolovém režimu. Sto tisíc iterací zpracování výrazu `1 + 3 * efg << 12 || 8 == 3 + abc` bylo provedeno za 886 ms. Průměrně tedy trvalo zpracování tohoto výrazu zhruba 0.0009 ms.



Obrázek 6.2: Všechna měření alokovaného prostoru

Knihovní funkce

Dále bylo provedeno testování rychlosti několika vybraných knihovních funkcí. Jako nejpřesnější metoda měření času interpretovaného kódu byla využita funkce `performance.now()`[4]. Pokusy o srovnání s gcc[5] překladačem a spuštěním zkompilovaného kódu byly zmařeny optimalizacemi překladače. Optimalizace v mojí práci vzhledem k chybějícímu interpretu jazyka C nejsou k dispozici. Zatímco přeloženému kódu trvala miliarda vykonání jedné funkce 2ms, mému nástroji to zabralo zhruba 190 tisíc krát delší dobu. Tyto optimalizace nebylo možné vypnout ani pomocí přepínače -O0, je tedy pravděpodobné, že optimalizace probíhá na nižší úrovni než jazyka C. Srovnání tedy probíhalo pouze mezi konzolovou aplikací a verzí pro prohlížeč v několika prohlížečích. Měření času také komplikuje skutečnost, že doba běhu programu může být významně ovlivněna procesy systému nesouvisejícími se samotnou aplikací. Z tohoto důvodu bylo prováděno velké množství iterací každého testu a výsledná hodnota je průměrem ze všech iterací.

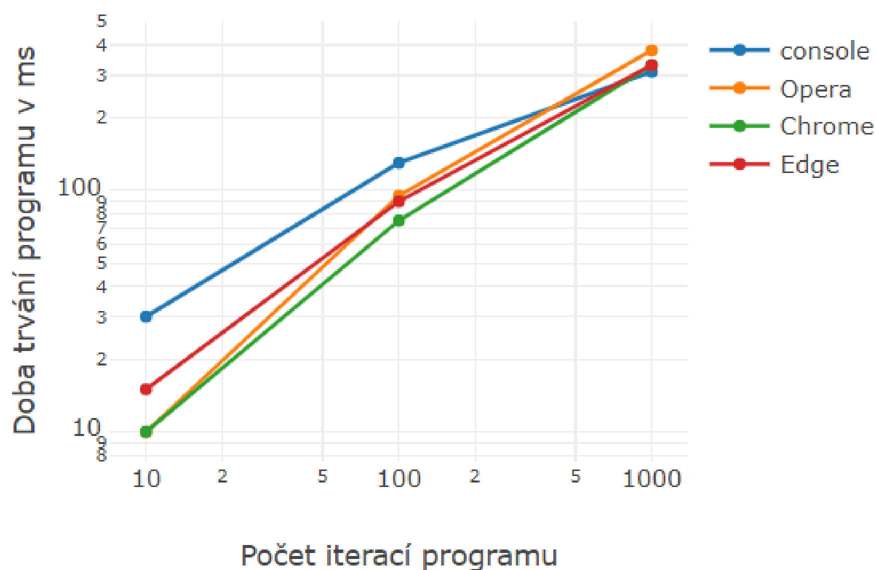
atoi()

- jednoduchá konverzní funkce pro převod řetězcové konstanty na celočíselnou konstantu

Varianta s řetězcovou konstantou jako argument (bez přístupu do paměťového modelu):
Varianta s předáním proměnné nesoucí řetězcovou konstantu (s přístupem do paměťového modelu):

Z těchto 6.5 6.6 měření vyplývá, že rychlost mezi jednotlivý prohlížeči se téměř neliší, je však téměř pětinásobně časově náročnější zároveň s voláním funkce přistupovat do paměťového modelu. Tento výkonnostní problém by řešila například optimalizace na úrovni interpretu. Vzhledem k časové náročnosti přístupu do paměťového modelu se při dalších

[4] https://developer.mozilla.org/en-US/docs/Web/API/Performance/now
[5] https://gcc.gnu.org/



Obrázek 6.3: Všechna měření doby trvání programu

Platforma	Průměrná doba 100 milionů iterací volání funkce v ms
Konzole	51842.6
Opera	50437.5
Chrome	51492.9
Edge	50213.8

Tabulka 6.5: Doba trvání vykonání funkce atoi()

měřeních již budeme zabývat pouze voláním funkcí s konstantami jako argumenty a nikoliv názvy proměnných. Samotná délka trvání jednoho volání funkce `atoi()` je zhruba 0,002 milisekundy.

strcmp()

- funkce pro lexikální porovnání dvou řetězcových konstant

Z těchto 6.7 měření vyplývá, že rychlost mezi jednotlivý prohlížeči se opět téměř neliší, je zde však patrná větší rychlost konzolové verze aplikace. Samotná délka trvání jednoho volání funkce `strcmp()` je zhruba 0,016 milisekundy.

pow()

- funkce pro umocnění čísla na libovolný exponent

Z měření 6.7 se jeví, že volání funkce z knihovny `math` má velmi rozdílnou dobu vykonání s ohledem na platformu, ze které je volána. Samotná délka trvání jednoho volání funkce `pow()` je zhruba 0,004 milisekundy.

Platforma	Průměrná doba 100 milionů iterací volání funkce v ms
Konzole	248846.1
Opera	257708.3
Chrome	257457.4
Edge	255735.2

Tabulka 6.6: Doba trvání vykonání funkce atoi()

Platforma	Průměrná doba 100 milionů iterací volání funkce v ms
Konzole	151372.6
Opera	168724.7
Chrome	169169.4
Edge	167959.9

Tabulka 6.7: Doba trvání vykonání funkce strcmp()

6.2 Jednotkové testy

Přímo ve zdrojovém kódu jsou napsány jednotkové testy knihovních funkcí. Pro jejich spuštění stačí odkomentovat blok kódu nacházející se na konci souboru zdrojového kódu těsně před sekvencí `try-catch` rozlišující spuštění z konzole od aplikace běžící v prohlížeči.

6.3 Agregační testy

Pro testování funkčnosti preprocesoru byl napsán testovací skript nacházející se ve složce `/tests`. Stačí jej pouze spustit, úspěšnost testů je poté zobrazena v konzoli.

Platforma	Průměrná doba 100 tisíců iterací volání funkce v ms
Konzole	1116.6
Opera	611.7
Chrome	343.1
Edge	508.4

Tabulka 6.8: Doba trvání vykonání funkce pow()

Kapitola 7

Závěr

Cíle této práce byly až na drobné výjimky naplněny. Při hledání řešení konkrétních problému jak při implementaci preprocesoru, tak při implementaci paměťového modelu C, byl dvakrát kompletně transformován základní návrh celého programu, neboť návrh původní se projevil jako nedostačující. Velkou komplikací byla samotná absence interpretu, jenž byla očekávaná, a bylo tudíž potřebné doimplementovat chybějící komponenty mimo původní plán, což v konečném důsledku vyústilo v absenci několika zmíněných knihovních funkcí.

Práci lze rozvíjet dalším směrem například implementací interpretu se kterým tahle práce má spolupracovat, případně úpravou paměťového modelu aby podporoval bajtový přístup nebo přidáním několika chybějících funkcí popsaných v podsekci 5.2.9.

Preprocesor lze otestovat v prohlížeči na veřejných stránkách univerzity `https://www.stud.fit.vutbr.cz/~xburda13/cpp/`.

Literatura

[1] Courouble, T. *C++ language documentation* [online]. [cit. 2022-15-05]. Dostupné z: http://www.cplusplus.com/.

[2] ISO. *Information technology — Programming languages — C*. ISO 9899:2018. Geneva, Switzerland: International Organization for Standardization, 2018 [cit. 2022-17-05].

[3] Network, T. C. R. *C language documentation* [online]. [cit. 2022-16-05]. Dostupné z: https://devdocs.io/c.

[4] Rahić, A. *Debugging Node.js Memory Leaks: How to Detect, Solve or Avoid Them in Applications* [online]. © Sematext Group, únor 2022 [cit. 2022-17-05]. Dostupné z: https://sematext.com/blog/nodejs-memory-leaks/#toc-how-to-avoid-memory-leaks-in-nodejs-applications-prevention-best-practices-11.

[5] Stallman, R. *GNU compiler collection documentation* [online]. [cit. 2022-15-05]. Dostupné z: https://gcc.gnu.org/.

Příloha A

Obsah přiloženého média

```
/
├── text - soubory technické zprávy
│   ├── xburda13 - technická zpráva ve formě PDF
│   └── tex - zdrojové soubory technické zprávy v LaTeXu
├── src
│   ├── fix - pomocný soubor pro správnou funkčnost webové aplikace
│   ├── index - html zdrojový kód frontendu webové aplikace
│   ├── script - hlavní zdrojový kód celé aplikace
│   ├── sw - zdrojový kód nástroje service worker
│   ├── webpack - konfigurační soubor nástroje webpack
│   └── package - seznam modulů potřebných pro spuštění aplikace
└── tests
    ├── cpp_test_files - testovací data pro preprocesor
    └── testFunctionalityMacros - automatický testovací skript
```

Příloha B

Návod pro zprovoznění aplikace

Ve složce `src` je potřeba zavolat nejprve příkaz `npm install`, jenž stáhne potřebné moduly.

V případě použití konzolové verze aplikace stačí spustit soubor `script.js`, jenž na vstupu očekává vstup pro preprocesor, na standardní výstup poté vytiskne výsledek preprocesingu.

Pro zprovoznění webové aplikace je nutné opět ve složce `src` zavolat příkaz `npx webpack`, který umožní spuštění aplikace přímo v prohlížeči i se všemi moduly. Následně stačí pouze spustit v této složce server obsluhující `GET` požadavky.

Příloha C

Pravidla gramatiky

BASE → # DIRECTIVE eol BASE
BASE → NONDIRECTIVE eol BASE
BASE → eps

DIRECTIVE → define id NONDIRECTIVE
DIRECTIVE → undef id
DIRECTIVE → include hdrname
DIRECTIVE → if expr
DIRECTIVE → ifdef id
DIRECTIVE → ifndef id
DIRECTIVE → else
DIRECTIVE → elif expr
DIRECTIVE → endif
DIRECTIVE → line integer FILE
DIRECTIVE → error NONDIRECTIVE
DIRECTIVE → pragma
DIRECTIVE → eps

FILE → string
FILE → eps

NONDIRECTIVE → SRCNONTERMS NONDIRECTIVE
NONDIRECTIVE → eps

SRCNONTERMS → id|cppnumber|string|char|op|punct|other

	#	eol	define	id	undef	include	hdrname	if	expr	ifdef	ifndef	else	elif	endif	line	integer	error	pragma	string	id|cppnumber|string|char|op|punct|other	$
BASE	1	2																		2	3
DIRECTIVE		16	4		5	6		7		8	9	10	11	12	13		14	15			
NONDIRECTIVE		20																		19	
FILE		18																	17		
SRCNONTERMS																				21	

Obrázek C.1: LL tabulka vygenerovaná z pravidel pomocí nástroje TODO_REF?.