České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra počítačů
Obor: Softwarové inženýrství

Automatická kontrola stylu zápisu programu

Automatic check of the program coding style

DIPLOMOVÁ PRÁCE

Vypracoval:	Adam Novák
Vedoucí práce:	RNDr. Ingrid Nagyová, Ph.D.
Rok:	2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Novák** Jméno: **Adam** Osobní číslo: **492188**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Automatická kontrola stylu zápisu programu

Název diplomové práce anglicky:

Automatic check of the code style

Pokyny pro vypracování:

Cílem projektu je analyzovat pravidla pro zápis zdrojového kódu počítačového programu v jazyce C a vytvořit nástroj, který by dokázal zkontrolovat čitelnost a udržitelnost programu vytvořeného začínajícím programátorem.
1. Seznamte se s pravidly, která jsou doporučována pro psaní čitelného a udržitelného programového kódu v jazyce C.
2. Analyzujte, která z pravidel lze automaticky (pomocí počítače) vyhodnotit. Speciálně se zaměřte na pravidla, která souvisí s individuálním stylem programátora.
3. Seznamte se s možnostmi, jak jednotliví vyučující hodnotí kódovací styl programů v jazyce C v systému BRUTE.
4. Navrhněte a implementujte systém, který dokáže zkontrolovat čitelnost zdrojového kódu zapsaného v jazyce C a poskytnout jeho autorovi zpětnou vazbu.
5. Systém otestujte na příkladech z BRUTE a výsledky srovnejte s výsledky manuálního hodnocení prováděného vyučujícími.

Seznam doporučené literatury:

1. www.misra.org.uk
2. Stallman, R. „Making the best use of C.“ GNU Coding Standards. 2021. https://www.gnu.org/prep/standards/html_node/index.html#SEC_Contents
3. The Linux Kernel Development Community. „Linux kernel coding style.“ https://www.kernel.org/doc/html/v4.10/process/coding-style.html#allocating-memory
4. Kernighan, Brian W.; Ritchie, Dennis M. „The C Programming Language.“ 2nd. Prentice Hall Professional Technical Reference, 1988. isbn 0131103709.
5. ISO. „ISO C Standard 1999.“ 1999. Tech. zpr. Dostupné také z: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf.

Jméno a pracoviště vedoucí(ho) diplomové práce:

RNDr. Ingrid Nagyová, Ph.D. kabinet výuky informatiky FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **15.02.2024** Termín odevzdání diplomové práce: **24.05.2024**

Platnost zadání diplomové práce: **21.09.2025**

RNDr. Ingrid Nagyová, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

 © ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Prohlášení

Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze dne ....

....
Adam Novák

Poděkování

Děkuji RNDr. Ingrid Nagyové, Ph.D. za vedení mé diplomové práce a za podnětné návrhy, které ji obohatily.

Adam Novák

Název práce:
Automatická kontrola stylu zápisu programu

Autor: Adam Novák

Studijní program: Otevřená informatika
Obor: Softwarové inženýrství
Druh práce: Diplomová práce

Vedoucí práce: RNDr. Ingrid Nagyová, Ph.D.

Abstrakt: Hlavním cílem diplomové práce bylo provést analýzu základních pravidel pro psaní čitelného zdrojového kódu v programovacím jazyce C a na základě této analýzy navrhnout, implementovat a funkčně otestovat řešení, které by tato pravidla aplikovala. Program byl navržen tak, aby byl schopen generovat přehledný výstup pro studenty i vyučující. Součástí práce bylo také vyhodnocení korektnosti aplikace formou manuální kontroly výstupů na anonymizovaných zdrojových kódech studentů z předmětu Procedurální programování. Výstupem práce se stal funkční nástroj pro kontrolu stylu zápisu programu, který pomáhá programátorům zlepšit čitelnost jejich kódu v jazyce C.
Klíčová slova: Programovací jazyk C, Pravidla pro psaní zdrojového kódu, Analýza kódovacích stylů

Title:
Automatic check of the program coding style

Author: Adam Novák

Abstract: The main goal of the thesis was to analyze the basic rules for writing readable source code in the C programming language and based on this analysis to design, implement and functionally test an application that would apply these rules. The program was designed to be able to generate clear output for students and teachers. The work also included evaluating the correctness of the application by manually checking the output on the anonymized source code of students in the Procedural Programming course. The output of the work was a functional tool for checking the writing style of a program to help programmers improve the readability of their C code.
Key words: Programming language C, Rules for writing source code, Analysis of coding styles

Obsah

Seznam obrázků xiii

Úvod 1

1 Terminologie 3
1.1 Kódovací styl . . . 3
1.2 Konvence pojmenování . . . 4
1.2.1 Notace camel case . . . 4
1.2.2 Notace snake case . . . 5
1.2.3 Maďarská notace . . . 5
1.3 Bílé znaky . . . 7
1.4 Styly odsazení . . . 8
1.4.1 K&R . . . 8
1.4.2 Allman . . . 9
1.4.3 GNU . . . 9
1.4.4 Linux . . . 10

2 Abstraktní syntaktický strom 11

3 Základní pravidla pro psaní čitelného kódu 13
3.1 Konzistentní názvy . . . 15
3.2 Formátování . . . 16
3.3 Komentáře . . . 18
3.4 Makra . . . 18
3.5 Struktura kódu . . . 19
3.6 Lokálnost . . . 20
3.7 Práce s pamětí . . . 20
3.8 Shrnutí . . . 21

4 Technické požadavky 23
4.1 Vstup . . . 23
4.2 Výstup . . . 24
4.3 Kompatibilita . . . 25
4.4 Konfigurace . . . 25
4.5 Vstupní data . . . 25

5 Konceptuální návrh 27
5.1 Myšlenka projektu . . . 27
5.2 Proces aplikace . . . 27
5.2.1 Struktura pravidla . . . 28
5.2.2 Průběh hlavní funkce . . . 29
5.2.3 Struktura konfigurace . . . 29

6 Implementace 31
6.1 Výběr technologií . . . 31
6.1.1 Vývojové prostředí . . . 31
6.1.2 Programovací jazyk . . . 32
6.1.3 Libclang v jazyce C++ . . . 37
6.1.4 Konfigurační soubor . . . 39
6.2 Práce se soubory . . . 41
6.3 Kompabilita . . . 42
6.4 Průběh programu . . . 42
6.5 Kontrolní výstupy . . . 44
6.6 Pravidla pro psaní čitelného kódu . . . 44
6.6.1 Konzistentní názvy . . . 46
6.6.2 Formátování . . . 48
6.6.3 Komentáře . . . 51
6.6.4 Makra . . . 52
6.6.5 Struktura kódu . . . 53
6.6.6 Lokálnost . . . 56
6.6.7 Práce s pamětí . . . 56
6.7 Adresářová struktura projektu . . . 57
6.8 Spuštění programu . . . 58
6.9 Výstup programu . . . 59
6.9.1 Výstup pro studenty . . . 59
6.9.2 Výstup pro vyučující . . . 61
6.9.3 Výstup pro skript . . . 61
6.10 Shrnutí . . . 62

7 Testování 63
7.1 Ověření v praxi . . . 65

Závěr 67

Bibliografie 69

Přílohy 75
A Implementované funkce . . . 75
B Instalace a spuštění . . . 76
B.1 Závislosti . . . 76
B.2 Kompilace . . . 76
B.3 Spuštění . . . 76
B.4 Výstup . . . 77
B.5 Testování . . . 77
C Úlohy použité při testování . . . 78
D Externí přílohy . . . 79

Seznam zdrojových kódů

1.1 Ukázka rozdílných zápisů stejné funkcionality . . . 3
1.2 Ukázka upper camel case a lower camel case . . . 4
1.3 Ukázka snake case a screaming snake case . . . 5
1.4 Ukázka maďarské notace . . . 6
1.5 Ukázka rozšířené maďarské notace v jazyce C . . . 7
1.6 Ukázka bílých znaků ve zdrojovém kódu . . . 7
1.7 Ukázka K&R stylu . . . 8
1.8 Ukázka Allmanova stylu . . . 9
1.9 Ukázka GNU stylu . . . 10
1.10 Ukázka Linux stylu . . . 10
2.1 Zdrojový kód k abstraktnímu syntaktickému stromu 2.1 . . . 12
3.1 Ukázka použití screaming snake case . . . 15
3.2 Ukázka použití více jazyků v kódu . . . 16
3.3 Implementace funkce pro výpočet počtu vteřin ve dnech . . . 18
4.1 Funkce definovaná makrem . . . 26
5.1 Pseudokód hlavní funkce . . . 29
5.2 Pseudokód konfigurace . . . 30
6.1 Ukázka jednoduchého zdrojového kódu určeného k tokenizaci . . . 32
6.2 Ukázka vytvoření AST v pycparseru a jeho výpis . . . 33
6.3 Nevhodný zdrojový kód s direktivami pro pycparser . . . 33
6.4 Zkrácený zdrojový kód po zpracování preprocesorem . . . 34
6.5 Ukázka vytvoření AST v libclang a jeho výpis . . . 35
6.6 Modifikovaná funkce pro výpis uzlu AST . . . 36
6.7 Ukázka vytvoření AST v C++ . . . 37
6.8 Datová struktura pro C++ . . . 38
6.9 Ukázka rekurze v datové struktuře . . . 38
6.10 Ukázka rekurze v datové struktuře . . . 39
6.11 Výsledek zpracování konfiguračního souboru. . . . 40
6.12 Zjednodušený kód pro načítání konfigurace . . . 40
6.13 Datová struktura pro načítané soubory . . . 41
6.14 Kontrolní výpis získaných hodnot . . . 44
6.15 Podoba hlavičky funkce pro pravidla . . . 45
6.16 Obecná struktura implementovaného pravidla . . . 45
6.17 Ukázka vstupu pro funkci print_stats . . . 46
6.18 Ukázka konvencí pojmenování pro vyhodnocení pravidlem . . . 47
6.19 Ukázka implementace for cyklu s makry . . . 48
6.20 Ukázka z implementace pravidla pro kontrolu ASCII znaků . . . 49
6.21 Ilustrace významu mezer v kódu . . . 50
6.22 Různé styly zápisu složených závorek . . . 50

6.23 Program pro výpočet obvodu kruhu 52
6.24 Ukázka dvou podobných bloků kódu 54
6.25 Ukázka dvou podobných bloků kódu, které se liší 54
6.26 Ukázka formátu komentáře pro zpracování 61
7.1 Ukázka jednotkového testu 64

Seznam obrázků

2.1 Ukázka abstraktního syntaktického stromu [27] 11

3.1 IBM karta [36] 17

4.1 Diagram výstupu v aplikaci. 24

5.1 Proces navrhované aplikace 28

6.1 Diagram průběhu programu 43
6.2 Diagram pravidla scanf 55
6.3 Diagram pravidel pro práci s pamětí a soubory 56

Úvod

Problematika stylu zápisu zdrojového kódu je velmi diskutovanou a stále otevřenou otázkou, na kterou prozatím neexistuje jednoznačná odpověď. Původem této otázky je přirozený vývoj jednotlivých programovacích jazyků a rozvoj softwarových technologií jako takových. Příkladem takového rozvoje může být projekt Linux, na kterém dnes stojí nejen většina operačních systémů, ale i celá internetová infrastruktura. S tím, jak rostl význam Linuxu, rostl i jeho zdrojový kód. Dnes má přibližně 27 milionů řádků kódu, na kterém se podílejí tisíce vývojářů z celého světa. Aby bylo možné dlouhodobě takové množství kódu udržovat, je nezbytné dodržovat základní pravidla, jakými má být napsán. V tomto konkrétním případě se tato pravidla nazývají *Linux kernel coding style* a kód, který není napsaný těmito konvencemi se pravděpodobně do linuxového jádra nepodaří prosadit.

Linux je pouze ukázkou toho, proč jsou taková pravidla nezbytná. Ve skutečnosti existuje mnoho standardů, které určují, jakým způsobem má být zdrojový kód napsán. Například pro programovací jazyk `Python` existuje *Průvodce stylem pro kód Python*, také známý jako `PEP8`. Není nutné se držet v mezích programovacího jazyka, dokonce i některé společnosti mají jako součást vlastní kultury specifická pravidla. Obecně známým příkladem je například společnost Google, která po svých vývojářích v rámci jazyka `Java` striktně vyžaduje dodržování specifikace pojmenované *Google Java Style Guide*.

Z výše uvedených příkladů je patrné, proč je nezbytné taková pravidla mít. Zároveň je patrné, že existuje mnoho pravidel pro mnoho jazyků. I přes tuto pluralitu existuje určitý průnik obecně dodržovaných základních pravidel. Tato práce se vymezuje pouze na programovací jazyk C a předpokládá, že hlavními uživateli budou začínající programátoři. Cílem tohoto projektu je nabídnout těmto začínajícím programátorům nástroj, který by umožnil snáze konfrontovat jejich styl zápisu programu s obecně platnými pravidly. Získáním zpětné vazby mohou zlepšovat své schopnosti a naučit se psát čitelné a přehledné zdrojové kódy.

Vzhledem k povaze této práce bude často využíváno obecně známé anglické názvosloví, jehož české ekvivalenty buď neexistují, anebo by jeho překlady nebyly schopny dostatečně vystihnout problematiku, kterou reprezentují. Zároveň se tato relaxace bude vztahovat i na další pojmy a názvy (například název programovacího jazyka `Python`), jejichž úpravy dle pravidel českého jazyka by mohly způsobit, že text nebude plynule čitelný nebo srozumitelný. Autor věří, že text touto změnou neutrpí na své kvalitě a čtenář toto rozhodnutí přijme.

Nejprve bude definována a popsána odborná terminologie, která se vztahuje k této diplomové práci, a na kterou budou následující kapitoly odkazovat. Dále budou stručně popsána základní pravidla uplatňovaná v rámci předmětu Procedurální programování. Na základě jejich výčtu dojde k rešerši odborné literatury, která naskytne podrobný vhled do problematiky jednotlivých pravidel stylu zápisu zdro-

jového kódu. Popsáním jejich vlastností a důvodů existence v širším kontextu bude možné navrhnout a implementovat - v souladu se zadáním a technickými požadavky - aplikaci, která bude tato pravidla kontrolovat ve zdrojových kódech zadaných jako vstup. Současně v průběhu implementace budou vytvářeny jednotkové testy, jejichž cílem bude reprezentovat modelové situace a kontrolovat správné chování implementace. V závěru práce dojde k otestování v praxi, kdy bude pro kontrolu předložena testovací sada různorodých zdrojových kódů napsaných v programovacím jazyce C. Princip testování bude založen na manuální kontrole výstupů generovaných výslednou aplikací.

Výsledkem této diplomové práce bude tedy nástroj pro kontrolu stylu zápisu programů napsaných v programovacím jazyce C, který bude generovat přehledné výstupy pro uživatele.

Kapitola 1

Terminologie

Zpočátku je nezbytné vysvětlit definice a významy pojmů, které budou používány v následujících kapitolách, a to s cílem dosáhnout srozumitelnosti a přesnosti jak v odborné, tak i v praktické části. V této kapitole budou za užití odborné literatury popsány a vysvětleny klíčové pojmy, které jsou základními stavebními kameny celkového porozumění diplomové práce. Důraz bude kladen především na formální definice a objasnění jednotlivých pojmů.

1.1 Kódovací styl

Kódovací styl, anglicky *coding style*, je textová reprezentace zdrojového kódu, která neovlivňuje chování při provádění programu [1]. Ve studii zkoumající kódovací styl jako indikátor k identifikaci kvalitních programátorů je definovaný jako estetická volba, která se projevuje ve zdrojovém kódu a odráží individuální zvyky programátorů [2]. Na základě definic lze tedy říci, že kódovací styl je soubor konvencí, které se aplikují při psaní zdrojového kódu programu. Styl zápisu zdrojového kódu má vliv na čitelnosti, srozumitelnost a různé aspekty softwarového inženýrství, včetně údržby softwaru [3] a rychlosti jeho vývoje [4]. V programování existuje celá řada stylů, které se projevují na různých místech zdrojového kódu. Některá vývojová prostředí umožňují nastavit až několik set parametrů v konfiguraci stylu kódování.

```
/* function to multiply two numbers */
int multiply(int a, int b) {
    return a*b;
}

// function to multiply two numbers
int multiply(int a, int b)
{
    return a * b;
}
```

Zdrojový kód 1.1: Ukázka rozdílných zápisů stejné funkcionality

Zdrojový kód 1.1 znázorňuje dva různé styly zápisu funkce `multiply`. I přesto, že jsou rozdílné, na výslednou funkcionalitu to nebude mít vliv. Existuje mnoho různých konvencí pro styl zápisu zdrojového kódu, přičemž každá z nich může být považována za správnou. V následujících kapitolách budou jednotlivá pravidla a styly popsány podrobněji.

1.2 Konvence pojmenování

Konvence pojmenování se vztahují k identifikátorům. Identifikátor je unikátní řetězec znaků, který značí entitu ve zdrojovém kódu [5]. Entitami jsou například proměnné, funkce, typy nebo konstanty. Styly zápisů názvů nejsou striktně definovány, ale jsou běžně přijímaným standardem v oblasti programovacích jazyků.

1.2.1 Notace camel case

Je-li název identifikátoru složen z několika slov, jako například *hello world* nebo *interest rate*, je zvykem, že každé slovo začíná velkým písmenem, s výjimkou prvního slova; tento způsob je nazýván camel case, jelikož velká písmena uprostřed slova mají připomínat hrboly na zádech velblouda. [6] V tomto případě by tedy název *hello world* byl zapsán jako `helloWorld` a *interest rate* jako `interestRate`.

Existuje podrobnější dělení camel case na varianty *lower camel case*, což reprezentuje malé počáteční písmeno prvního slova. Opakem je *upper camel case* (viz. zdrojový kód 1.2), kde je první písmeno každého slova vždy velké. Tato varianta je také známá jako *pascal case*, neboť je odvozena od konvence psaní identifikátorů v programovacím jazyce Turbo Pascal.

```
int main(void) {

    int camelCase
    int lowerCamelCase;
    int UpperCamelCase, PascalCase;

    return 0;

}
```

Zdrojový kód 1.2: Ukázka upper camel case a lower camel case

Studie popsaná v práci *To camelcase or under-score* ukazuje [7], že i když osoby bez tréninku potřebují více času na rozpoznání identifikátorů ve stylu camel case, všechny subjekty byly přesnější když rozpoznávaly camel case identifikátory. Kromě toho subjekty, které byly trénované v používání camel case, potřebovaly méně času na rozpoznání identifikátoru v camel case než identifikátoru ve snake case notaci. Tento fakt by mohl být jedním z mnoha důvodů, proč právě tato notace převládá a je stále populární.

1.2.2 Notace snake case

Snake case, underscore case či lower case with underscores je konvence pojmenovávání v programování, často používaná v jazycích C nebo Python. V tomto systému je každé písmeno ponecháno malé a každé slovo je odděleno podtržítkem [8]. Dokument *Google C++ Style Guides* uvádí obdobnou definici, tedy názvy jsou psány malými písmeny a mezi slovy jsou umístěna podtržítka [9]. Názvy *hello world* a *interest rate* by byly dle pravidel zapsány jako `hello_world` a `interest_rate`.

Screaming snake case, také známý jako constant case, je speciální varianta snake case. Jedná se o koncept pojmenovávání, který kombinuje velká písmena s vlastnostmi snake case. Všechna písmena ve slovech jsou napsána velkými písmeny a slova jsou oddělena podtržítky (viz. zdrojový kód 1.3). [10] Tato varianta mívá speciální význam. Používá se například pro pojmenování konstant, tj. hodnot, které se během běhu programu nemění.

```
int main(void) {

    const int SCREAMING_SNAKE_CASE = 1;
    const int CONSTANT_CASE = 2;
    int snake_case;
    int underscore_case;
    int lower_case_with_underscores;

    return 0;

}
```

Zdrojový kód 1.3: Ukázka snake case a screaming snake case

1.2.3 Maďarská notace

Maďarská notace je sada doporučených konvencí pro předpony názvů proměnných několika písmeny (viz. tabulka 1.1), která označují jejich typ [11]. Autorem tohoto konceptu, používaného od 70. let minulého století, je programátor Charles Simonyi, který se později stal významnou osobou ve společnosti Microsoft. Název *maďarská* odkazuje na původ svého autora a také na fakt, že některé názvy identifikátorů mohou vypadat, jako by byly napsány v nějakém čtenáři neznámém jazyce. Například proměnná *owner*, která by byla datového typu *ukazatel* na zero-terminated řetězec, by se jmenovala `pszOwner`.

Výhodou je, že maďarská notace může pomoci usnadnit pochopení programů, zvláště, pokud máme mnoho proměnných různých datových typů, které jsou argumenty funkcí Windows API či funkcí obecně. [12] V současné době převážná většina moderních integrovaných vývojových prostředí automaticky zobrazuje informace o datových typech proměnných a rovněž upozorňuje na operace, které používají nekompatibilní datové typy. Tím pádem je zápis touto notací do značné míry zastaralý a nadbytečný.

Předpona	**Význam**
b	Byte (8 bitů)
c	Charakter
dbl	Double
f	Float
i	Integer
p	Pointer
s	Short
str	Řetězec
sz	Null-terminated řetězec

Tabulka 1.1: Ukázka předpon a jejich významů v Maďarské notaci

Maďarská notace se dále rozděluje do dvou rozdílných přístupů: Systems Hungarian a Apps Hungarian. Systems Hungarian se zaměřuje na zahrnutí informací o datovém typu proměnné do jejího názvu (viz. zdrojový kód 1.4). Jde tedy o původní význam. Apps Hungarian se soustředí na popis významu proměnné přidáním doprovodné informace do názvu identifikátoru. Ukázkou rozdílných přístupů může být například proměnná *vertical*, která značí délku vertikální strany objektu. Zatímco v Systems Hungarian by název byl `iVertical`, tak v Apps Hungarian by byl `lenVertical`.

```
int main(void) {

    /* Hungarian notation */
    int iVar; // integer variable
    double dbVar; // double variable

    /* Systems Hungarian */
    char *usName; // unsafe string of name
    long lAccountNum; // account number of long type

    /* Apps Hungarian */
    int cRefs; // count of references
    int noStudents // number of students

    return 0;

}
```

Zdrojový kód 1.4: Ukázka maďarské notace

V některých případech se maďarská notace rozšiřuje v jazyce C o zahrnutí rozsahu platnosti proměnné, který je volitelně oddělen podtržítkem. Tato rozšířená verze se často používá i bez použití jakékoliv konvence zápisu názvů a slouží jak programátorům, tak i čtenářům jako upozornění na specifickou vlastnost (viz. zdrojový kód 1.5).

```
// prefix g as global
int g_nCars;

int main(void) {

    // prefix s as static
    static int s_nDrivers;

    return 0;

}
```

Zdrojový kód 1.5: Ukázka rozšířené maďarské notace v jazyce C

1.3 Bílé znaky

Bílé znaky jsou znaky, které textový editor při prohlížení a editaci textového souboru reprezentujícího program nezobrazuje, ale interpretuje je například jako prázdná místa, přechod na další řádek, nebo tabulátory [13]. Bílé znaky slouží jako oddělovače jednotlivých tokenů a umožňují programátorovi získat kontrolu nad vizuálním uspořádáním zdrojového kódu [14]. Při analýze zdrojového kódu jazyka C hraje počet bílých znaků mezi tokeny zanedbatelnou roli. Vzhledem k tomu, že slouží pouze k ohraničení tokenů, jejich množství se může značně lišit.

Zdrojový kód 1.6 za užití pomocných symbolů znázorňuje použití bílých znaků. Znak *plus* reprezentuje nový řádek, *tečka* mezeru a *prázdné místo* tabulátor. Ačkoliv je funkce `main` pokaždé zapsána s různým počtem oddělovačů identifikátorů, tak funkcionalita bude identická.

```
// first main function+
int.main(void).{+
....printf("Hello␣␣world");+
....return.0;+
}+
+
+
// second main function+
int.main(void).{+
....printf(."Hello␣␣world"...);+
....return.0;+
}+
```

Zdrojový kód 1.6: Ukázka bílých znaků ve zdrojovém kódu

1.4 Styly odsazení

Odsazování je způsob, jak reprezentovat vztah mezi konstrukty řízení toku, jako jsou podmínkové výroky nebo smyčky, a kódem, který je uvnitř nebo vně nich [15]. Dodržováním této konvence se vizuálně formátuje zdrojový kód a tím se zlepšuje jeho čitelnost [16]. Studie provedená na University of Waterloo taktéž ukazuje, že používání pravidel odsazení vede k menšímu počtu komentářů ve zdrojovém kódu [17]. Dříve se k formátování zdrojového kódu používaly znaky tabulátoru, což mělo vliv na velikost souboru. Dnes se namísto tabulátoru lze setkat se dvěma, čtyřmi nebo osmi mezerami. Historicky existuje nemalé množství stylů odsazení [18] a některá jsou přímo nedoporučovaná [19]. V následujících podkapitolách budou vzhledem k povaze této práce rozebrány pouze ty konvence, které jsou nejvíce používané v programovacím jazyce C.

1.4.1 K&R

Tento styl je pojmenován po Brianu Kernighanovi a Dennisu Ritchiem. Jedná se o jeden z nejrozšířenějších stylů odsazování v programovacím jazyce C a je používán v knize *The C Programming Language*, kterou napsali právě Kernighan a Ritchie. [20]

V K&R stylu je každá funkce strukturována následovně - otevírací závorka je umístěna na novém řádku s odsazením odpovídajícím záhlaví funkce. Kód uvnitř funkce je odsazen, a uzavírací závorka na konci je umístěna na samostatném řádku na stejné úrovni odsazení jako záhlaví funkce. Počáteční složené závorky se v ostatních případech zapisují na stejný řádek a mezi klíčovými slovy jazyka je vždy mezera (viz. zdrojový kód 1.7). Varianta, kdy klíčové slovo `else` je zapisováno na samostatný řádek, se jmenuje Stroustrupova, podle stejnojmenného autora knihy *(The) C++ programming language*, kde byl tento styl použit [21].

```
int main(void)
{

   int c = getchar();

   while (c != EOF) {
      if (some_error) {
         break;
      } else {
         continue;
      }
   }

   return 0;

}
```

Zdrojový kód 1.7: Ukázka K&R stylu

1.4.2 Allman

Tento druh stylu je pojmenován po Ericu Allmanovi, který v minulém století napsal nemalé množství zdrojového kódu pro Berkeley Software Distribution[1] (BSD). Proto se také nazývá jako BSD styl. V tomto případě jsou vnořené bloky kódu odsazeny a mezi klíčovými slovy programovacího jazyka jsou mezery. Hlavním rozdílem v porovnání s jinými styly jsou otevírací i zavírací závorky bloků, které jsou umístěny na samostatném řádku s odsazením odpovídajícím záhlaví funkce nebo příkazu, který blok otevírá (viz. zdrojový kód 1.8).

```
int main(void)
{

    int c = getchar();

    while (c != EOF)
    {
        if (some_error)
        {
            break;
        }
        else
        {
            continue;
        }
    }

    return 0;

}
```

Zdrojový kód 1.8: Ukázka Allmanova stylu

1.4.3 GNU

GNU's Not Unix jsou svobodné a otevřené softwarové projekty, které byly započaty v roce 1983 Richardem Stallmanem [22]. V GNU kódovacím standardu se dodržuje praxe vkládání složených závorek na samostatné řádky, které jsou odsazeny dvěma mezerami. Tato konvence platí pro všechny bloky kódů s výjimkou otevření a uzavření definice funkce, kde složené závorky nejsou odsazeny. [2] Také kód uvnitř bloků, například větvení nebo smyčky, je odsazen dvěma mezerami, aby byla zachována jasná vizuální struktura kódu. Je striktně vyžadováno, aby návratový datový typ funkce byl umístěn na samostatný řádek, oddělený od názvu a seznamu parametrů (viz. zdrojový kód 1.9).

[1] viz. repozitář https://github.com/freebsd/freebsd-src

[2] viz. dokumentace https://www.gnu.org/prep/standards/

```
int
main (void)
{

  int c = getchar();

  while (c != EOF)
    {
      continue;
    }

  return 0;

}
```

Zdrojový kód 1.9: Ukázka GNU stylu

1.4.4 Linux

Linux kernel dodržuje styl psaní kódu nazývaný *Linux kernel coding style* nebo *Linux kernel indent style*, který je zdokumentován v repozitáři zdrojových kódů Linuxového jádra [23]. K odsazení jsou použity tabulátory reprezentující 8 znaků. Otevírací závorky funkce jsou na samostatném řádku za záhlavím funkce s odpovídajícím odsazením. Jakékoli další otevírací složené závorky jsou na stejném řádku, jako odpovídající příkaz, oddělené mezerou. Pouze v případě příkazu `switch` existuje pouze jedna úroveň odsazení.

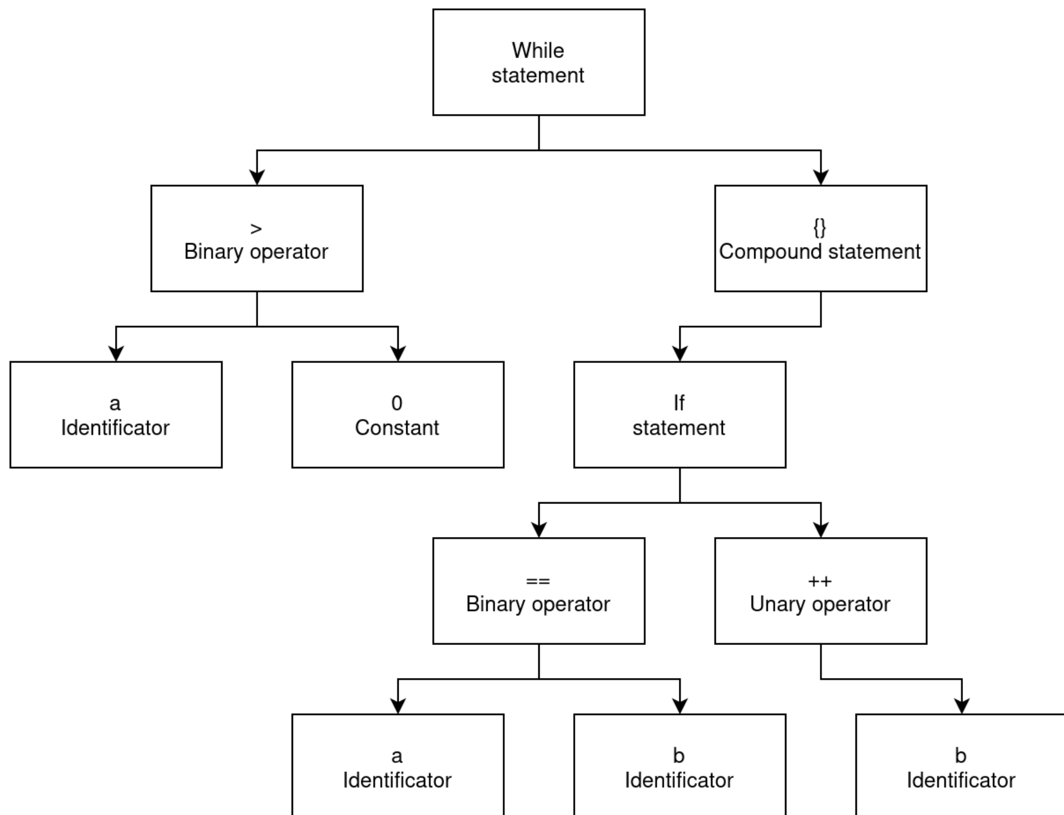
```
int main(void)
{

        int c = getchar();

        while (c != EOF) {
                if (some_error) {
                        break;
                }
        }

        return 0;

}
```

Zdrojový kód 1.10: Ukázka Linux stylu

Kapitola 2

Abstraktní syntaktický strom

Tuto kapitolu by bylo možné začlenit do terminologie. Avšak tento termín je natolik stěžejní pro tuto práci, že byl oddělen do samostatné kapitoly. Abstraktní syntaktický strom (AST) je jedna z nejběžnějších metod reprezentace zdrojového kódu ve formě stromové struktury [24]. V každém jeho uzlu je reprezentován konstrukt ze zdrojového kódu (viz. obrázek 2.1). Listy ve stromu představují elementární prvky, které jsou buď identifikátory nebo konstanty [25]. AST odstraňuje nepotřebné informace ze vstupního kódu, a tedy výsledná stromová struktura obsahuje pouze relevantní data. Současně je zachována hierarchická struktura a díky tomu je snadnější analyzovat kód a provádět různé operace, jako je kontrola, optimalizace nebo generování mezikódu [26].



Obrázek 2.1: Ukázka abstraktního syntaktického stromu [27]

```
while (a > 0) {
    if (a == b)
        b++;
}
```

Zdrojový kód 2.1: Zdrojový kód k abstraktnímu syntaktickému stromu 2.1

Kapitola 3

Základní pravidla pro psaní čitelného kódu

Pravidla zápisu zdrojového kódu zajišťují, že kód je napsán konzistentně a čitelně. Zároveň vytváří společný formát jazyka, který vývojáři mohou snadno pochopit. V konečném důsledku dodržováním stanovených pravidel lze minimalizovat riziko nových chyb, jednoduše se orientovat [28] a provádět změny v kódu rychleji a efektivněji. Toto tvrzení mimo jiné dokazuje i publikovaný seznam deseti největších příčin defektu softwaru, kde je přímo uvedeno, že disciplinované programovací praktiky mohou snížit míru výskytu defektů až o 75 procent [29].

Praktická část této diplomové práce bude koncipována jako podpůrný software pro studenty a vyučující v rámci předmětu Procedurální programování, který je vyučován v prvním semestru programu Otevřená informatika. Formálním požadavkem pro úspěšné absolvování kurzu je mimo jiné níže uvedený základní seznam pravidel pro psaní kvalitního zdrojového kódu. [1] V následujících částech budou tato pravidla podrobně popsána, aby bylo zřetelné, jak konkrétně přispívají k rozvoji dovedností v oblasti programování a jaký mají dopad na kvalitu a čitelnost výsledného kódu.

1. **Konzistentní názvy**

   (a) **Význam proměnné/funkce je popsaný jejich názvem:** Proměnné a funkce by měly mít názvy, které jasně vysvětlují, co reprezentují nebo provádějí.

   (b) **Formát názvů je sjednocený v rámci programu:** V rámci celého kódu je používán jednotný styl pojmenování.

2. **Formátování**

   (a) **V programu jsou použité pouze ASCII znaky:** Kód obsahuje pouze znaky ze standardní ASCII tabulky.

   (b) **Řádek programu má maximálně 80 znaků:** Každý řádek kódu je omezen na maximálně 80 znaků.

[1] Originální seznam: `https://cw.fel.cvut.cz/wiki/courses/b0b36prp/resources/tessun/start`

3. **Komentáře**
   (a) **Složitější části kódu jsou doplněny komentářem:** Komplexnější sekce kódu, jako jsou složité algoritmy nebo funkce s neintuitivní logikou, jsou doprovázeny komentáři, které vysvětlují jejich funkci a důvod jejich použití.
4. **Makra**
   (a) **Pro netriviální numerické literály jsou definovaná makra:** Místo používání *magických* čísel je v kódu využíváno maker, která tyto hodnoty pojmenovávají.
   (b) **V kontextu ukazatelů preferovat `NULL` nad `0`:** Pro inicializaci a kontrolu ukazatelů je používán nulový ukazatel `NULL` namísto numerické nuly (`0`).
5. **Struktura kódu**
   (a) **Program je rozdělen do krátkých a jednoduchých funkcí:** Každá funkce by měla mít jasně definovaný a omezený účel. To znamená, že funkce by neměla být příliš dlouhá nebo zahrnovat více než jeden primární účel nebo funkčnost.
   (b) **Je preferované nízké zanoření podmínek (IF) a cyklů (FOR / WHILE):** Kód by měl být strukturován tak, aby se minimalizovalo hluboké zanoření.
   (c) **Logika ukončení cyklů je přehledná:** Podmínky pro ukončení cyklu by měly být jasné a snadno pochopitelné.
   (d) **Program neobsahuje podobné bloky kódu:** Místo kopírování a vkládání podobných kódových bloků je vhodnější použít funkce, které tuto funkčnost abstrahují a znovu použijí.
   (e) **Vstupy programu/funkcí jsou ošetřeny:** Vstupní parametry funkcí a vstupy do programu by měly být kontrolovány na platnost a relevanci.
6. **Lokálnost**
   (a) **Proměnné jsou zavedeny co nejblíže použití:** Proměnné by měly být deklarovány v co nejmenší možné vzdálenosti od místa jejich prvního použití.
   (b) **Životnost proměnné je co nejvíc omezená:** Životnost proměnných by měla být co nejkratší a měly by existovat pouze po dobu, kdy jsou potřebné.
7. **Práce s pamětí**
   (a) **Dynamická alokace je kontrolovaná:** Při dynamické alokaci paměti je nezbytné ověřit, zda byla alokace úspěšná. Neúspěšná alokace by měla být náležitě ošetřena.
   (b) **Práce se soubory je kontrolovaná:** Při otevírání, čtení, zápisu a zavírání souborů je třeba pečlivě kontrolovat výsledky těchto operací. To zahrnuje ověření, že soubor byl úspěšně otevřen, a ujištění se o řádném zavření souborů po dokončení práce s nimi.

3.1 Konzistentní názvy

Správné pojmenování identifikátorů, jako jsou proměnné, konstanty, funkce či struktury, je klíčovým prvkem v programování, neboť jasné a výstižné názvy přispívají k čitelnosti a srozumitelnosti zdrojového kódu. Identifikátory by měly jednoznačně odhalit svůj účel a funkci bez nutnosti komentářů, a pokud tomu tak není, je třeba přehodnotit jejich pojmenování. Například Derek M. Jones ve své práci *Operand names influence operator precedence decisions* dává do kontextu proměnné a operátory. Hypotéza, že vývojáři využívají kontextové informace obsažené v názvech identifikátorů, když se rozhodují, na který operand se váže, byla podpořena výsledky experimentu. Vliv informací o názvech na rozhodování o prioritě byl významný. [30]

V předchozí kapitole 1.2.1 byly definovány odlišné notace pro zápis identifikátorů v programovacích jazycích. Každá notace nabízí specifické výhody a je proto nezbytné při výběru zohlednit různé faktory v rámci daného projektu. Zejména je důležité zvolit **jednotný styl pro pojmenování identifikátorů**, který bude dodržován napříč celým projektem [31]. V případě, že již existují ustálené konvence, je vhodné se jich držet. Symbolické konstanty, jak již bylo zmíněno v kapitole o notaci snake case (1.2.2), by se měly zapisovat velkými písmeny ve stylu screaming snake case, aby se odlišily od názvů proměnných a funkcí (viz zdrojový kód 3.1). Definice maker a jejich názvy by měly být taktéž psány stejnou konvencí pojmenování [32].

```
#define SQUARE(X) ((X) * (X))
#define PI_VALUE 3.141592

int main(void) {

    double radius = 5.0;
    double result = PI_VALUE * SQUARE(radius)

    return result;

}
```

Zdrojový kód 3.1: Ukázka použití screaming snake case

Kromě jednotného formátu názvů a jejich správné pojmenování je nezbytné, aby identifikátory byly vyslovitelné. Schopnost vyslovit jejich název je klíčová pro případnou diskuzi o nich. Tato schopnost je úzce spojena s jazykem, ve kterém je identifikátor pojmenován. Programovací jazyk C byl původně navržen a implementován s anglickými klíčovými slovy pro operační systém UNIX na počítači DEC PDP-11 Dennisem Ritchiem ve Spojených státech amerických v minulém století [20], a proto je **preferování anglických názvů** zcela opodstatněné.

```
#define SQUARE(X) ((X) * (X))
#define PI_HODNOTA 3.141592

int main(void) {

    double polomer = 5.0;
    double vysledek = 2 * PI_HODNOTA * SQUARE(radius)

    return vysledek;

}
```

Zdrojový kód 3.2: Ukázka použití více jazyků v kódu

Výše uvedený zdrojový kód 3.2, který vychází z ukázky 3.1, znázorňuje využití více jazyků v jednom programu. Zatímco klíčová slova jazyka jsou psána anglicky, některé proměnné a konstanty jsou psány česky. I přesto, že tato pojmenování nebudou mít vliv na funkčnost, **může být tento zápis pro čtenáře matoucí**.

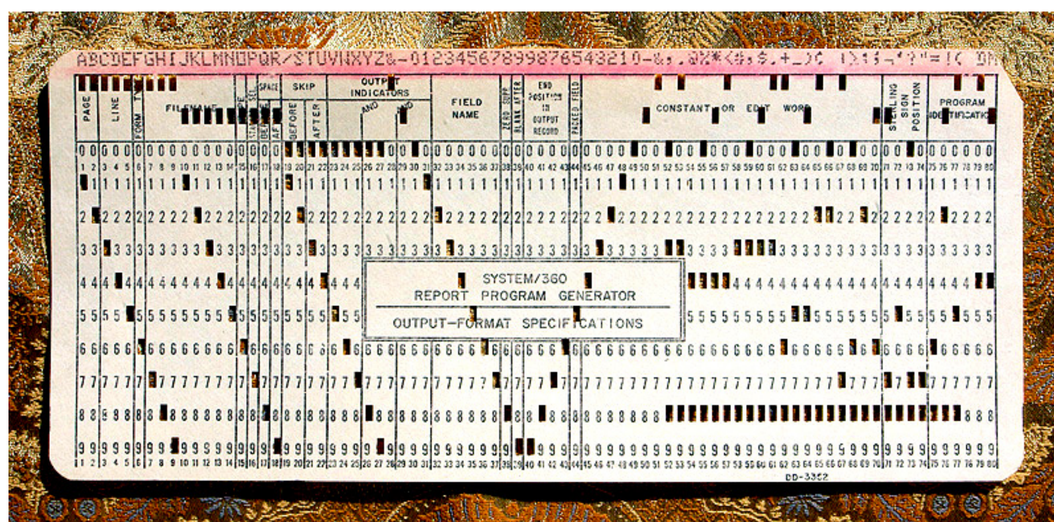
3.2 Formátování

ASCII (American Standard Code for Information Interchange) je **standardní znaková sada**, která přiřazuje jednoznačné číselné kódy různým znakům používaným v anglickém jazyce a dalším symbolům [33]. Naprostá většina platforem dnes používá pro kódování znaků typu `char` American Standard Code for Information Interchange. Není nutné vědět, jak konkrétní kódování uvnitř stroje funguje, pokud zůstaneme pouze v základní znakové sadě [34]. Standardní knihovny programovacího jazyka C toto kódování transparentně aplikují a pokusíme-li se použít speciální znaky, které nepochází ze standardní sady, může být chování programu na různých platformách odlišné. Lze tedy konstatovat, že omezení na znakovou sadu ASCII zajišťuje kompatibilitu a čitelnost napříč hardwarem.

ASCII znaky nejsou vyžadovány pouze v rámci kurzu Procedurální programování. Podrobněji se pravidly zabývá mnoho rozšířených standardů jazyků C a C++. V oficiálním dokumentu pro vývojáře společnosti Google je přímo uvedeno, že non-ASCII znaky by měly být velice vzácné, nikoliv běžné. [2] Mnohem striktněji k problematice například přistupují autoři GNU Coding Standards. V komentářích ke zdrojovému kódu GNU, textových dokumentech a dalších kontextech je preferováno držet se znakové sady ASCII (prostý text, 7-bitové znaky), pokud neexistuje dobrý důvod udělat něco jiného kvůli doméně aplikace. [3] Další omezení zmiňuje například standard MISRA C (Motor Industry Software Reliability Association) a další.

[2] viz. stránka https://google.github.io/styleguide/cppguide.html
[3] viz. stránka `https://www.gnu.org/prep/standards/html_node/Writing-C.html`

Standardní **limit 80 znaků** pro délku kódu **na jednom řádku** úzce souvisí s historickým rozvojem technologií. V roce 1928 byla zveřejněna *IBM karta* s šířkou právě 80 sloupců a 10 řádků (viz. obrázek 3.1) [35]. Historicky byly vytvořeny i jiné formáty s méně sloupci či řádky, ale tato karta byla s ostatními kompatibilní a získala si větší popularitu. Později v 70. letech 20. století byl na trh uveden obrazovkový terminál VT100 od společnosti Digital Equipment Corporation, který disponoval displejem schopným zobrazit maximálně 80 znaků na řádek.



Obrázek 3.1: IBM karta [36]

Ve výchozím nastavení terminálu moderního operačního systému Windows 11 je nastaveno 80 znaků na řádek. Toto omezení odpovídá historickému vývoji standardů a technickým limitům terminálů ze starších verzí systémů Windows, čímž zajišťuje zpětnou kompatibilitu. Důležitým milníkem svobodného softwaru byl vznik linuxového jádra. Linux byl původně vyvinut Linusem Torvaldsem v roce 1991 jako operační systém pro IBM-kompatibilní osobní počítače založené na mikroprocesoru Intel 80386 [37]. V dokumentaci je přímo uvedena poznámka: *"Limit na délku řádků je 80 sloupců a to je silně preferovaný limit."* [4]

Jak již bylo řečeno, příčinou pro omezení byla schopnost tehdejších tiskáren a monitorů zobrazit limitované množství znaků na řádce. Nicméně, v současné době, kdy jsou širokoúhlé monitory běžnými zařízeními, se nabízí otázka, proč toto omezení stále přetrvává. Podle článku *Typography and readability*, jehož autorem je Markus Itkonen, je čtení textu lidmi prováděno sekvenčně a v každém skoku je oko schopné přečíst přibližně 10 znaků. Ve své práci uvádí, že maximální limit by měl být stanoven na 90 znaků na řádek, což by mělo přispět lepší čitelnosti, správné rychlosti čtení a přirozenému pohybu oka. [38]

Omezení délky řádky v kódovacích konvencích se částečně relaxuje s rozvojem monitorů a rozmanitostí znakových sad. Například konvence společnosti Google pro jazyk Java nyní stanovuje limit 100 znaků na řádek, kde se za znak považují všechny znaky Unicode. [5] Dále zachází například vývojové prostředí `CLion` pro jazyky C a C++ od společnosti JetBrains, které ve výchozím nastavení stanovuje maximální délku řádku na 120 znaků.

[4]viz. https://www.kernel.org/doc/html/v4.10/process/coding-style.html

[5]viz. stránka `https://google.github.io/styleguide/javaguide.html`

3.3 Komentáře

Pochopení zdrojového kódu v jakémkoliv programovacím jazyce může být velmi obtížné, zejména pokud sami nejsme jeho autorem, a proto by měli vývojáři dbát na dobrou dokumentaci implementovaného softwaru. Komentáře ke zdrojovému kódu jsou jednou z nejužitečnějších forem dokumentace a metadat pro pochopení implementace softwaru [39]. Komplexnější sekce kódu, jako jsou složité algoritmy nebo funkce s neintuitivní logikou, by tedy měly být doprovázeny komentáři, které jasně a stručně vysvětlují jejich funkci. Nekvalitní komentáře mohou mít negativní vliv na kvalitu softwaru nebo mohou být neúčinné při porozumění kódu [40].

Woodfield a kolektiv již v roce 1981 provedli experiment na 48 zkušených vývojářích, kterým ukázali různé typy modulárních kódů s komentáři a bez komentářů. Následný kvíz o 20 otázkách ukázal, že subjekty, jejichž kód obsahoval komentáře, dokázaly správně odpovědět na více otázek. Ve shodě s tímto zjištěním 66 % softwarových odborníků, kteří odpověděli na průzkum v této studii, souhlasilo s tím, že komentáře pomáhají lépe porozumět kódu. [41] Hartzman a Austin ve své případové studii na velkém softwarovém systému došli k závěru, že komentáře jsou klíčovým faktorem pro dlouhodobé udržování softwaru [42]. Většinu času a úsilí vynaloženého při činnostech souvisejících s údržbou představuje čtení kódu. Méně času stráveného čtením kódu následně vede ke snížení nákladů [43], což je z obchodního hlediska důležitá vlastnost.

3.4 Makra

Pro **netriviální numerické literály** v kódu by měla být definovaná makra. Pokud tomu tak není, jsou tyto hodnoty zpravidla nazývána jako *magická čísla*. Magická čísla jsou konstantní hodnoty, jejichž význam nemusí být čtenáři kódu zřejmý [44]. Jedná se pravděpodobně o jedno z nejstarších pravidel při vývoji softwaru, jelikož je možné jej nalézt v příručkách jazyků Cobol a Fortran z šedesátých let minulého století [45]. Pro ukázku vlivu magických čísel na čitelnost kódu byla implementována jednoduchá funkce `secondsInDays` za použití makra a bez makra (viz zdrojový kód 3.3). Pro čtenáře, který není seznámen s faktem, že číslo 86400 představuje počet vteřin v jednom dni, bude funkce `secondsInDaysWithoutMacro` nesrozumitelná.

```
#define SECONDS_IN_DAY 86400

long secondsInDaysWithMacro(int n) {
    return n * SECONDS_IN_DAY;
}

long secondsInDaysWithoutMacro(int n) {
    return n * 86400;
}
```

Zdrojový kód 3.3: Implementace funkce pro výpočet počtu vteřin ve dnech

V programovacím jazyce C se použití maker neprojeví na výkonu aplikace, jelikož jsou nahrazeny preprocesorem před kompilací. Existence magických čísel je nepříjemná i v případě, že se jedna hodnota vyskytuje v kódu opakovaně. Pokud by se totiž měla hodnota změnit, je nezbytné provést úpravu v několika částech programu. Přítomnost (nejen číselných) literálů ve zdrojovém kódu nemusí nutně znamenat porušení konvencí pro zápis programu, neboť v některých kontextech je zcela legitimní nebo nutné vložit hodnotu literálu na určité místo v kódu [46].

Ukazatel je proměnná, jejíž hodnota je adresa jiné proměnné [47]. Správné používání ukazatelů je obtížné i pro studenty, kteří již mají zkušenosti se základními řídícími strukturami a paměťovým modelem [48]. Proto je doporučeno **v kontextu nulových ukazatelů preferovat makro `NULL` nad numerickou `0`**. Makro `NULL` a hodnota 0 jsou ekvivalentní jako nulové konstanty ukazatele, ale `NULL` je srozumitelnější, protože představuje účel použití konstanty pro ukazatel. [6] Z hlediska typové bezpečnosti, když je `NULL` definován jako `(void*)0`, použití `NULL` místo numerické nuly může pomoci zabránit některým typům chyb a varování při kompilaci.

3.5 Struktura kódu

Počítačový program je utvářen reprezentací dat a příkazy. Ty definují strukturu programu. [49] Gerald Weinberg zdůrazňuje, že programátor je důležitějším čtenářem kódu než překladač a naznačuje, že je třeba kód i několikrát přepsat, než se z něj stane dobře strukturovaný a čitelný kód [50].

Strukturu kódu lze vyjádřit za pomocí odsazení. Slovo **odsazení** označuje přístup k uspořádání zdrojového kódu tak, že se na začátku řádku kódu použije určitá mezera, aby se zdůraznil specifický fragment kódu. Johannes Morzeck a kolektiv ukazují, že odsazení má - přinejmenším v kontextu řídících toků - pozitivní vliv na čtenáře ve srovnání s neodsazeným kódem. [51] Článek *Impact of Indentation in Programming* kromě jeho významu také popisuje několik typů odsazení [52], které byly částečně popsány v kapitole 1.

V dobře strukturovaném kódu by měly být **podmínky pro ukončení cyklů** snadno pochopitelné. Doporučení vychází například z empirických pozorování, které naznačují, že `FOR` smyčky jsou výrazně složitější na pochopení než `IF` příkazy [53]. Toto zjištění by bylo možné zobecnit i na další druhy smyček `WHILE` a `DO-WHILE`. Složitost ukončovacích podmínek smyček může přispět k zvýšení pravděpodobnosti chyb v programu včetně špatné čitelnosti. V praxi se používají jednoduché a přímé podmínky pro ukončení, které jsou často na začátku nebo na konci bloku kódu.

Dále by měl být kód v takovém formátu, aby neobsahoval **duplicitní bloky kódu**, čehož lze dosáhnout **rozdělením práce do krátkých a jednoduchých funkcí**. Duplicitní kód je definován jako identické nebo navzájem podobné sekvence příkazů ve zdrojovém kódu, které vznikají z různých důvodů, například při programování typu *copy-and-paste* [54]. Kim a kolektiv provedli empirickou studii na dvou softwarových systémech s otevřeným zdrojovým kódem a zjistili, že přibližně 38 % skupin duplicitního kódu bylo alespoň jednou změněno [55]. Toto relativně nízké číslo může být jednou z příčin, proč existuje i opačný názor, že klonování kódu je dobrou volbou pro návrh zdrojového kódu [56]. Zkoumání vzorků ze vzdělávacího prostředí umožnilo vyučujícím identifikovat několik častých oprav nečitelného kódu, které zahrnovaly i redukci duplicit [57].

[6] viz IBM dokumentace `https://www.ibm.com/docs/en/zos/2.4.0?topic=pointers-null`

S maximální délkou řádků, odsazením a rozdělením práce do jednoduchých funkcí souvisí i **nízké zanoření podmínek a cyklů**. Myšlenku, že přílišné zanoření kódu snižuje čitelnost, se nepodařilo přímo potvrdit. Některé práce poukazují na fakt, že v tomto případě je srozumitelnost kódu závislá na specifických situacích [58]. Další literatura bez širšího kontextu pouze uvádí, že vnořené řídící konstrukce jsou dost obtížné na pochopení [19]. I tak je toto pravidlo v rámci kódovacího stylu vyžadováno. Například *PEP 20 – The Zen of Python* zmiňuje pro strukturu kódu větu: *"Flat is better than nested."* Neodborná literatura používá termín *Arrow Anti Pattern* pro vnořené příkazy `IF`, jelikož generují tvar šipky.

Poslední bod této sekce se věnuje **ošetření vstupů**. Ověřování vstupů je považováno za správnou programátorskou praxi při psaní spolehlivého softwaru [59]. Tento postup je univerzální a snižuje pravděpodobnost, že se do softwaru dostanou zranitelnosti spojené se vstupem [60]. Programovací jazyk C provádí všechny operace čtení prostřednictvím knihovních funkcí, jako jsou `getchar()` a `scanf()` [61]. Aby se předešlo chybám, dochází často k ověření bezpečnosti aplikace již ve fázi revize kódu, a proto je nezbytné tuto kontrolu provádět v blízkosti načítání vstupu a v přehledné formě.

3.6 Lokálnost

Dle konvencí by měly být **proměnné zavedeny co nejblíže použití**, a tudíž jejich **životnost co nejvíce omezená**. Životnost proměnné je omezena na blok, ve kterém byla definována [62]. Pokud je proměnná definována globálně, znamená to, že je dostupná v celém programu, a pokud je proměnná definována v lokálním rozsahu, tak je omezená pouze na daný blok kódu nebo funkci. Rozsah proměnné může ovlivnit schopnost porozumět kódu [63]. Nakamura a kolektiv zmiňují, že je obtížné pochopit roli a hodnotu proměnné, jejíž definice je oddělena od reference na ni [64]. Změna pořadí příkazů, která zkrátí vzdálenost mezi definicí proměnné a prvním použití, bude mít pozitivní vliv na čitelnost zdrojového kódu [65].

3.7 Práce s pamětí

Při dynamické alokaci paměti je její přidělení prováděno za běhu programu. Většinou se jedná o explicitní volání funkcí pro správu haldy. Programovací jazyk C podporuje funkce `malloc()`, `calloc()` a `realloc()`, které slouží k přidělení paměti. Jazyk C také nabízí funkci `free()` pro dealokace nepoužívané paměti. [66] Obdobné zacházení lze nalézt při práci se soubory. Soubory je nezbytné jak otevřít (funkce `fopen()`), tak zavřít (funkce `fclose()`) [67]. Pokud si například program alokuje celou operační paměť počítače, tak mu ji nelze odebrat. Musí být uvolněna voláním konkrétního příkazu, v opačném případě se uvolní až při ukončení programu, což má negativní dopad na výkon ostatních aplikací. Špatné zacházení s prostředky nakonec může vést k řadě bezpečnostních problémů [68]. Za správné použití prostředků zodpovídá vývojář.

3.8 Shrnutí

V předešlé kapitole 1 pojednávající o odborné terminologii byl představen náhled do dané problematiky. Bylo ukázáno, že neexistuje standardizovaný způsob zápisu programu. Specifická pravidla lze nalézt pro pojmenování identifikátorů, odsazení kódu a další. Velké softwarové projekty navíc vynucují vlastní styl. Zmíněny byly například *GNU Coding Standards*, *Linux Kernel Coding Style* nebo *MISRA C*.

Jelikož existuje velké množství široce používaných stylů vhodných k rozboru, tak tato kapitola vycházela ze seznamu základních pravidel pro psaní čitelného zdrojového kódu z kurzu Procedurální programování. Pravidla byla uvedena především v kontextu odborné literatury, která poskytla podrobný vhled do příčin jejich vzniku.

Celkově lze říci, že kódovací styl má velký vliv na čitelnost a udržitelnost zdrojových kódů. Tento vliv navíc roste přímo úměrně s velikostí projektu. Pokud by nebyla dodržována žádná pravidla, tak by byly programy pro čtenáře těžko srozumitelné. To by v konečném důsledku vedlo ke ztíženému vývoji a vzniku vážných chyb v aplikacích.

Kapitola 4

Technické požadavky

Předmětem této kapitoly je soubor technických požadavků na aplikaci, jejíž návrh a implementace budou popsány v dalších částech. Požadavky byly definovány na základě zadání a sérií konzultací se zadavatelkou práce. Úkolem aplikace je zpracovat soubory se zdrojovými kódy v programovacím jazyce C a následně v nich ověřit předem definovaná pravidla stylu zápisu programu. V případě porušení pravidla dojde k uložení záznamu, jež bude obsahovat jaké pravidlo, jakým způsobem a v jaké části bylo porušeno. Výstup programu bude uložen do souboru ve specifikované složce a bude obsahovat nalezené záznamy v přehledné a srozumitelné formě.

4.1 Vstup

Jak již bylo zmíněno v úvodní části této kapitoly, vstupem jsou zdrojové kódy napsané v programovacím jazyce C. Přesněji řečeno, aplikace dostane seznam cest, ať už relativních či absolutních, k těmto zdrojovým kódům. Seznam může obsahovat cesty k souborům různých formátů, přičemž standardně se jedná o soubory s příponami `c` nebo `h`. Příkladem takového seznamu může být:

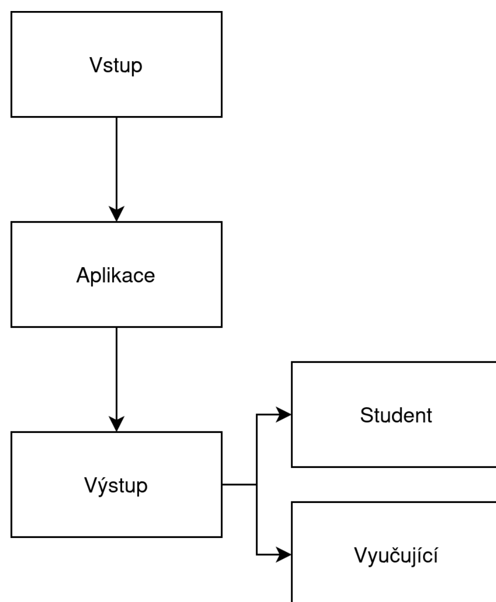
```
/student/novak/util.h
/student/novak/output.c
/student/novak/main.c
/student/novak/algorithm.c
/student/novak/memory.h
```

Dále může být vstupem cesta k dekomprimovanému souboru TGZ ve formě adresáře, který je produktem exportu ze systému BRUTE. Tento systém poskytuje funkci ke stažení všech zdrojových kódů studentů pro určité zadání. Adresář obsahuje složky jednotlivých studentů, a uvnitř těchto složek se nacházejí příslušné zdrojové kódy. Příkladem takového uspořádání může být následující stromová struktura:

```
HW04/
    dolezape/
        main.c
    novaka/
        main.c
```

4.2 Výstup

Výstupní soubory generované aplikací budou uloženy do složky `output`, která se bude nacházet v aktuálním adresáři, ze kterého je program spuštěn. V případě, že tato složka neexistuje, dojde k jejímu automatickému vytvoření. Pokud složka `output` již existuje, budou data uvnitř přepsána nově generovanými výstupy. Výstupní soubory budou připraveny ve dvou verzích - jedna bude určena pro studenta a druhá pro vyučujícího (viz obrázek 4.1). Každý výstupní soubor bude v takovém formátu, který bude snadno čitelný na různých platformách.



Obrázek 4.1: Diagram výstupu v aplikaci.

U každého porušení pravidla bude ve výstupním souboru uveden krátký popis specifikující podstatu problému a bude vyznačeno místo na řádku, kde došlo k jeho porušení. Struktura výstupních souborů bude přímo odvozena od druhu vstupních dat. V případě, že byly zpracovávány soubory s příponami `c` nebo `h`, výstupní složka `output` bude obsahovat pouze dva soubory – jeden určený pro studenta a druhý pro vyučujícího. Struktura složky může vypadat například takto:

```
output/
    student_file
    teacher_file
```

Na druhou stranu, pokud byla zpracována data ze systému BRUTE, struktura výstupní složky bude reflektovat jejich stromovou strukturu. V tomto případě bude pro každého studenta vytvořen samostatný adresář a v každém takovém adresáři budou umístěny dva výstupní soubory – jeden pro studenta a jeden pro vyučujícího. Příklad takové struktury může vypadat následovně:

```
output/
    dolezape/
        student_file
        teacher_file
    ...
```

4.3 Kompatibilita

V rámci kompatibility je vyžadováno, aby aplikace byla plně funkční v linuxovém prostředí. Tento požadavek je motivován faktem, že počítače, které jsou k dispozici ve fakultních učebnách, typicky používají operační systém Ubuntu. Linuxový operační systém je navíc doporučován v rámci předmětu Procedurální programování. Potenciální platformou, na které může být aplikace použita je fakultní systém BRUTE, který je taktéž schopen spolehlivě operovat s linuxovým prostředím.

4.4 Konfigurace

Na aplikaci je kladen požadavek, aby její konfigurace byla oddělena od zdrojového kódu. Cílem je umožnit modifikaci chování programu bez potřeby znalosti její implementace. Konfigurace musí obsahovat možnost deaktivovat libovolné pravidlo definované v aplikaci. Zároveň by měla existovat možnost úpravy míry penalizace stupňovitým systémem.

4.5 Vstupní data

Z úvodu kapitoly vyplývá, že aplikace bude zpracovávat zdrojové kódy programovacího jazyka C. Před samotným návrhem a implementací je vhodné zmínit, že existuje více specifikací tohoto jazyka. V roce 1989 byl vytvořen standard ANSI C, schválený jako *ANSI X3.159-1989*, a v roce 1990 byl tento standard adoptován Mezinárodní organizací pro normalizaci jako *ISO 9899*. [69] ANSI C bylo navrženo jako nadmnožina původního jazyka C s několika přidanými vlastnostmi a je široce podporováno většinou dnešních překladačů.

V pozdějších letech byl vydán standard C99 (*ISO 9899:1999*), který byl přijat také jako ANSI standard v roce 2000. Tento standard je momentálně nejvíce uplatňovaným. Pro účely této práce bude tedy standard C99 považován za výchozí.

C99 přinesl několik významných inovací a vylepšení, které rozšiřují možnosti programování v C a zlepšují interoperabilitu s programovacím jazykem C++ [70]. Následující seznam nabízí přehled některých klíčových vlastností standardu C99, které bude nezbytné brát v potaz.

- Inline funkce
- Deklarace proměnných kdekoli v kódu
- Nové datové typy
- Podpora pro jednořádkové komentáře
- Variadická makra
- Klíčové slovo `restrict`
- další ...

Vzhledem k existenci variadických maker (a maker obecně) je nutné při procesu zpracování zdrojových kódů a kontroly jejich stylu zápisu dbát zvýšené pozornosti kódům těmito makry generovanými. Zvláště je nezbytné zajistit, aby tyto kódy byly v aplikaci správně zpracovány. Pokud by se tak nedělo, výstup programu by nesprávně označoval čísla řádků a sloupců těch částí kódu, které jsou v rozporu s pravidly.

Jako příklad lze uvést makro DYN_ARRAY_PRINT (viz zdrojový kód 4.1), které generuje funkci pro tisk dynamického pole určitého typu. Toto makro definuje funkci TYPE##_print, která přijímá ukazatel na dynamické pole a vypisuje jeho obsah ve formátu zadaném argumentem PRINTF. Tento přístup demonstruje, jak je možné efektivně využívat makra k redukci kódu. Nicméně, je důležité si uvědomit, že při zpracování kódu preprocesorem dochází k vytvoření nového kódu, který se může výrazně lišit od původního zápisu. To přináší specifické výzvy, zejména v kontextu zpracování kódu.

```
#define DYN_ARRAY_PRINT(TYPE, PRINTF)                \
    void TYPE##_print(TYPE* t) {                     \
        putchar('[');                                \
        if (t->length != 0) {                        \
            printf(PRINTF, t->arr[0]);               \
        }                                            \
                                                     \
        for (int i = 1; i < t->length; ++i) {        \
            printf(", " PRINTF, t->arr[i]);          \
        }                                            \
                                                     \
        putchar(']');                                \
        putchar('\n');                               \
    }                                                \
```

Zdrojový kód 4.1: Funkce definovaná makrem

Kapitola 5

Konceptuální návrh

Obsahem této kapitoly bude konceptuální návrh, který bude zahrnovat přehledná schémata ilustrující hlavní komponenty aplikace a jejich vzájemné vztahy. Zaměříme se primárně na splnění technických požadavků projektu a rozvedení neimplementačních aspektů programu. Výsledný návrh poslouží jako základní rámec pro další fáze vývoje, zvláště pak při výběru specifických komponent nezbytných pro implementaci.

5.1 Myšlenka projektu

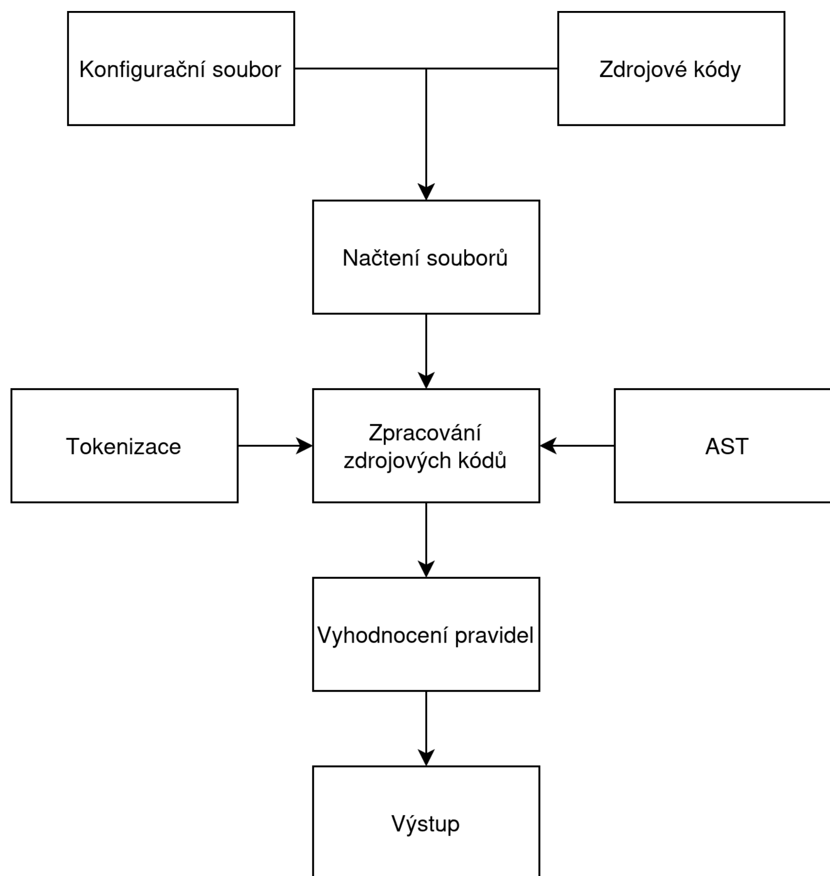
Autorovou myšlenkou je vytvořit aplikaci, která poskytne začínajícím programátorům kvalitní nástroj pro kontrolu zápisu zdrojového kódu v programovacím jazyce C s důrazem na intuitivitu použití.

Podstatnou součástí aplikace bude funkcionalita na generování výstupu. Tento výstup by měl být dostatečně detailní a informativní, poskytovat informace o nalezených chybách a ukázat přesnou lokaci těchto chyb ve zdrojovém kódu. Další důležitou vlastností projektu bude jeho modularita, aby bylo v budoucnu možné program rozšířit, a to zejména v možnostech generování výstupů v různých jazycích a přidávání nových pravidel.

Celkově bude kladen důraz na uživatelskou přívětivost, funkcionalitu a rozšiřitelnost. Autor věří, že tyto vlastnosti jsou nezbytné pro úspěšné použití softwaru vyučujícími a studenty v rámci předmětu Procedurální programování.

5.2 Proces aplikace

Prvním krokem aplikace bude načtení konfiguračního souboru a zdrojových kódů do paměti. Cesty k těmto souborům budou programu předány ve formě argumentů. Následovat bude proces zpracování zdrojových kódů do strojově čitelné podoby. Zpracování se bude skládat z tokenizace jednotlivých souborů, aby bylo možné iterovat přes jednotlivé identifikátory. Také budou vytvořeny abstraktní syntaktické stromy, které přiřadí tokenům jejich logické významy. Toto zpracování vstupu umožní ověřit předem definovaná pravidla. Výsledkem tohoto vyhodnocení bude seznam míst ve zdrojovém kódu, kde došlo k jejich porušení. Posledním krokem bude vytvoření přehledného výstupu jak pro vyučujícího, tak i pro studenta. (viz diagram procesu aplikace 5.1)



Obrázek 5.1: Proces navrhované aplikace

5.2.1 Struktura pravidla

V rámci dlouhodobé udržitelnosti projektu je nezbytná uniformní struktura každého pravidla. Pravidla budou implementována ve formě funkcí, přičemž každá funkce bude přijímat identické parametry, které budou obsahovat data důležitá pro její správné fungování. Následuje seznam parametrů.

- **Konfigurace** - Jedná se o nastavení, které bylo načteno z konfiguračního souboru při spuštění programu. Konfigurace obsahuje různé preference, které ovlivňují chování pravidel a způsob jejich aplikace na zdrojový kód.
- **Zdrojové kódy** - Tento parametr obsahuje zdrojové kódy určené k analýze. Každý zdrojový kód je reprezentován několika položkami.
  - **Abstraktní syntaktický strom:** AST zdrojového kódu.
  - **Tokeny:** Zdrojový kód rozdělený na tokeny.
  - **Řádky:** Původní zdrojový kód rozdělený na řádky.
  - **Cesta:** Cesta k souboru z něhož byl zdrojový kód načten.

5.2.2 Průběh hlavní funkce

Pokud bude mezi argumenty aplikace specifikován přepínač `-dir`, bude se předpokládat jako vstup rozbalený TGZ archiv ze systému BRUTE. Pro jednotlivé adresáře studentů proběhnou následující akce. Každý soubor v adresáři bude zpracován způsobem popsaným na začátku sekce 5.2, a poté dojde ke kontrole definovaných pravidel. V případě, že bude přítomen přepínač `-files`, dojde k obdobnému zpracování obsahu souborů jako v předchozím případě. Ať už se jedná o zpracování dekomprimovaného TGZ archivu nebo jednotlivých souborů, program vytvoří výstupní adresář, do kterého se vygenerují výstupy určené pro studenty a vyučující (viz pseudokód 5.1).

```
function main():

    language = argv[1]
    config = load configuration at argv[2]

    if argv[3] is '-dir':

        For each student in argv[4]:

            For each file in the student:
                Parse the file to structure

            Check rules on the parsed files
            Generate output in defined language

    else if argv[3] is '-files':

        For each file in argv[4:]:
            Parse the file to structure

        Check rules on the parsed files
        Generate output in defined language

    else:
        exit
```

Zdrojový kód 5.1: Pseudokód hlavní funkce

5.2.3 Struktura konfigurace

Konfigurace bude obsahovat tři základní sekce. První sekce bude určena pro výchozí nastavení aplikace. Druhá sekce se bude skládat z jazykových překladů textů použitých ve výstupních souborech a poslední sekce bude určena pravidlům. (viz pseudokód 5.2) Každé pravidlo v konfiguraci bude mít následující strukturu.

- **Require** - Parametr určuje, zda má být pravidlo kontrolováno.
- **First penalty - second penalty - next penalty** - Parametr definuje systém bodové penalizace pro první, druhé a následné porušení pravidla.
- **Messages** - Podsekce obsahuje specifické textové hlášky pro různé jazyky, které popisují důvody pro penalizace nebo upozornění.
- **Description** - Podrobný popis pravidla, včetně jeho účelu.

```
# default settings
tab_size: 4

# translations
texts:
    cs:
        message: Toto je zprava.
        rule: Pravidlo
    en:
        message: This is message.
        rule: Rule

# rules
rules:
    rule_name:
        require: true
        first penalty: -3
        second penalty: -2
        next penalty: -1
        messages:
            cs:
                error: chyba
            en:
                error: error
        description:
            cs:
                name: Nazev pravidla
                content: Toto je popis pravidla.
            en:
                name: Rule name
                content: This is description of rule.
```

Zdrojový kód 5.2: Pseudokód konfigurace

Kapitola 6

Implementace

Obsah kapitoly bude zaměřen na proces vývoje aplikace dle návrhu a technických požadavků. Bude rozebrán výběr nástrojů a technologií. Dále v podkapitolách budou také podrobně rozebrány jednotlivé fáze vývoje, včetně přehledů kódů a zajímavých aspektů, které nastaly během implementace.

6.1 Výběr technologií

Tato sekce se věnuje výběru nástrojů a technologií, jež sehrály důležitou roli během vývoje aplikace. Bude vysvětleno, proč byla provedena volba specifického vývojového prostředí, programovacího jazyka a externí knihovny s cílem poskytnout hlubší vhled do technických rozhodnutí a jak tyto volby ovlivnily výkonnost a funkcionalitu konečného programu.

6.1.1 Vývojové prostředí

Integrovaná vývojová prostředí (IDE) nabízejí různé nástroje pro provádění různých úkolů s ohledem na vývoj, včetně průběžné kompilace, automatizovaného testování, integrovaného ladění a refaktorování kódu pro zvýšení produktivity. [71] Na počátku projektu bylo rozhodnuto využít pro vývoj **Visual Studio Code**. Visual Studio Code je multiplatformní textový editor. V tomto případě multiplatformní znamená, že je k dispozici verze pro spuštění na Windows, macOS a Linux [72].
Z definice vyplývá, že se jedná pouze o textový editor, nikoliv o plnohodnotné vývojové prostředí. V tomto případě je vhodné zmínit, že jeho zdrojové kódy jsou k dispozici ve formě open-source a nehledě na programovací jazyk je nejpoužívanějším prostředím pro psaní kódu na světě. [1] Hlavními důvody pro jeho použití byla právě multiplatformost, multijazyčnost a podpora komunity, neboť bylo předpokládáno, že vývoj aplikace bude v různých fázích veden na vícero strojích v různých programovacích jazycích.

[1] viz Developer Survey 2023 `https://survey.stackoverflow.co/2023`

6.1.2 Programovací jazyk

Samotný vývoj aplikace započal po splnění formálních požadavků v rámci předmětu *Softwarový nebo výzkumný projekt*, který je obvykle určen k řešení dílčího problému diplomové práce. Již na počátku bylo předpokládáno, že s výslednou aplikací budou v budoucnu pracovat nejen vyučující, ale i studenti. Na základě informací získaných diskuzí se zadavatelkou práce bylo rozhodnuto, že bude použit programovací jazyk **Python**. Python je univerzální, multiplatformní a objektově orientovaný skriptovací jazyk. Podstatnou vlastností je, že jde o interpretovaný jazyk na vysoké úrovni, kde k překladu kódu do strojového jazyka dochází za běhu [73]. Není tedy nutné zdrojové kódy v něm napsané ručně kompilovat. Interpret je dostupný pro všechny běžné operační systémy a spuštění skriptů napsaných v tomto jazyce je uživatelsky velmi přívětivé.

Další nespornou výhodou jazyka Python je komunita, která neustále vytváří nové knihovny poskytující, mimo jiné, nástroje pro zpracování textu. V tomto konkrétním případě bylo nutné vybrat takovou knihovnu, která z textového souboru obsahujícího zdrojový kód v programovacím jazyce C vytvoří abstraktní syntaktický strom a provede lexikální analýzu. Pro upřesnění, nástroj pro lexikální analýzu, obecně nazývaný pojmem lexer, přijme jako svůj vstup řetězec jednotlivých písmen a rozdělí tento řetězec do slovně podobných entit nazývaných tokeny [74]. Například výstupem pro zdrojový kód 6.1 bude následující seznam tokenů: `#, include, <, stdio, ., h, >, int, main, (, void, ), {, return, 0, ;, }`.

```c
#include <stdio.h>

int main(void) {
    return 0;
}
```

Zdrojový kód 6.1: Ukázka jednoduchého zdrojového kódu určeného k tokenizaci

Pravděpodobně nejznámější knihovnou pro práci se zdrojovým kódem v jazyce C je **pycparser**. Pycparser je parser pro jazyk C napsaný čistě v jazyce Python. [2] Slouží k analýze statického kódu, jako front-end pro specializované kompilátory nebo jako obfuskátor kódu. Autoři si kladou za cíl plně podporovat standard C99 podle normy ISO/IEC 9899, což je zároveň v souladu s technickými požadavky implementované aplikace. Instalace probíhá pro všechny operační systémy stejně přes správce balíčků PIP příkazem `pip install pycparser`. Samotné použití je velmi přímočaré. Například pro vytvoření abstraktního syntaktického stromu stačí importovat parser, vytvořit jeho instanci a jako vstup mu předat zdrojový kód napsaný v programovacím jazyce C (viz ukázka kódu 6.2).

[2] viz Github `https://github.com/eliben/pycparser`

```
from pycparser import c_parser

def print_ast(node, indent=0):
    print(' ' * indent + node.__class__.__name__)
    for child_name, child in node.children():
        print_ast(child, indent + 4)

if __name__ == "__main__":

    code = """
    int main() {
        return 0;
    }
    """

    parser = c_parser.CParser()
    ast = parser.parse(code)

    print_ast(ast)
```

Zdrojový kód 6.2: Ukázka vytvoření AST v pycparseru a jeho výpis

Při podrobnějším testování funkcionality knihovny se objevila zásadní překážka pro použití v tomto projektu. Přidáme-li do vstupního zdrojového kódu jakékoliv direktivy pro preprocesor kompileru (viz upravený kód 6.3), výsledkem bude chybové hlášení `pycparser.plyparser.ParseError: Directives not supported yet`.

```
code = """
    #include <stdio.h>

    int main() {
        return 0;
    }
"""
```

Zdrojový kód 6.3: Nevhodný zdrojový kód s direktivami pro pycparser

Dokumentace se k tomuto chování vyjadřuje následovně. Aby bylo možné kód v jazyce C zkompilovat, musí být předzpracován preprocesorem jazyka C. Ten zpracovává direktivy, jako jsou `#include` a `#define`, odstraňuje komentáře a provádí další drobné úkony, které připravují kód v jazyce C ke kompilaci. Aby **pycparser** mohl správně fungovat, musí, stejně jako kompilátor jazyka C, přijímat předzpracovaný kód jazyka C. [3] Pro účely této práce je nezbytné manipulovat s originálními neupravenými zdrojovými kódy. Důvodem je, že i zdánlivě jednoduchý vstup se po zpracování preprocesorem transformuje do výstupu o délce několika set řádků (viz ukázka 6.4, jež je výsledkem preprocessingu kódu 6.3). To znemožňuje identifikaci původního umístění kódu v souboru, což by vedlo k odkazování na nesprávné lokace ve výsledcích implementované aplikace.

```
// ...

typedef unsigned char __u_char;
typedef unsigned long int __u_long;
typedef signed char __int8_t;
typedef unsigned char __uint8_t;
typedef signed short int __int16_t;
typedef unsigned short int __uint16_t;
typedef signed int __int32_t;
typedef unsigned int __uint32_t;
typedef signed long int __int64_t;
typedef unsigned long int __uint64_t;
extern int getchar (void);
extern int getc_unlocked (FILE *__stream);
extern int getchar_unlocked (void);
extern int fgetc_unlocked (FILE *__stream);
extern int fputc (int __c, FILE *__stream);
extern int putc (int __c, FILE *__stream);
extern int putchar (int __c);
extern int fputc_unlocked (int __c, FILE *__stream);
extern int putc_unlocked (int __c, FILE *__stream);
extern int putchar_unlocked (int __c);
extern int getw (FILE *__stream);
extern int putw (int __w, FILE *__stream);

int main() {
    return 0;
}
```

Zdrojový kód 6.4: Zkrácený zdrojový kód po zpracování preprocesorem

[3] viz Github `https://github.com/eliben/pycparser`

Zpracování zdrojových kódů napsaných v jazyce C představuje složitou výzvu. Obzvláště náročné bylo najít knihovnu, která by byla schopna tuto úlohu zvládnout s formální přesností a bez předchozího preprocessingu. Výsledkem pátrání se stal **Clang**. Projekt Clang poskytuje jazykový front-end a nástroje pro jazyky rodiny C pro projekt LLVM. K dispozici je jak ovladač překladače kompatibilní s GCC, tak i ovladač překladače kompatibilní s MSVC. [4] Pro jazyk Python lze například použít `libclang`. Projekt kopíruje potřebné Python binding soubory z LLVM repozitáře, přidává skripty pro balíčkování a nahrává výsledný balíček na PyPI. Aby byl `libclang` dostupný bez nutnosti instalace sad nástrojů LLVM, poskytuje tento balíček sdílené knihovny libclang se statickou vazbou pro různé platformy, které by měly dobře fungovat na OSX, Windows i běžných linuxových distribucích. [5] Tento neoficiální balíček tak zpřístupňuje Clang pro Python a zahrnuje dodatečnou funkcionalitu pro lepší kompatibilitu na různých operačních systémech a v extrémních případech umožňuje ruční nastavení cesty k původní knihovně.

Instalace probíhá obdobně jako v případě `pycparser`, a to přes správce balíčků příkazem `pip install libclang`. Příklad použití znázorňuje jednoduchá ukázka 6.5. Zde je důležité upozornit na fakt, že argument `-nostdinc` zajistí, aby tokeny každého uzlu výsledného abstraktního syntaktického stromu odpovídaly vstupnímu souboru.

```
import clang.cindex

def print_ast(node, indent=0):
    print(' ' * indent, str(node.type.kind.spelling))
    for child in node.get_children():
        print_ast(child, indent + 4)

if __name__ == "__main__":

    code = """
    int main() {
        return 0;
    }
    """

    index = clang.cindex.Index.create()
    arguments = ['-x', 'c', '-std=c99', '-nostdinc']
    cursor = index.parse(
                'fakefile.c',
                unsaved_files=[('fakefile.c', code)],
                args=arguments
            ).cursor
```

Zdrojový kód 6.5: Ukázka vytvoření AST v libclang a jeho výpis

[4] viz Github `https://clang.llvm.org/`
[5] viz PyPI `https://pypi.org/project/libclang/`

Četné konzultace a rychlá elektronická komunikace se zadavatelkou diplomové práce umožnila již v rané fázi semestrálního projektu vytvořit rozsáhlou kolekci anonymizovaných zdrojových kódů z fakultního systému BRUTE. Tato sada dat poskytla možnost empiricky ověřit výsledky generování abstraktního syntaktického stromu včetně samotné tokenizace vstupů. Výstupy parseru z `libclang` odpovídaly očekávaným výsledkům. Pokud spustíme kód uvedený v 6.5 s modifikovanou funkcí `print_ast` z 6.6, zjistíme, že vygenerovaný AST je odvozen z kódu, který prošel preprocessingem. Přesto tokeny, které tvoří jednotlivé uzly ve stromu, pocházejí z původního zdrojového souboru. Díky tomu je možné přesně odkazovat na specifická místa v kódu.

```python
def print_ast(node, indent=0):

    print("indent", indent)
    print("kind", cursor.kind)
    print("spelling", cursor.spelling)
    print("display_name", cursor.displayname)
    print("type", cursor.type.spelling)
    print("location", cursor.location.line, cursor.location.column)

    for token in cursor.get_tokens():
        print("token", token.spelling)
        print("token_kind", token.kind)
        print("location", token.location.line, token.location.column)

    for child_name, child in node.get_children():
        print_ast(child, indent + 1)
```

Zdrojový kód 6.6: Modifikovaná funkce pro výpis uzlu AST

Ačkoli se podařilo objevit knihovnu nabízející požadovanou funkcionalitu, ukázalo se, že jazyk Python není ideální pro efektivní správu paměti. Součástí datasetu byly zdrojové kódy s délkou několika tisíc řádků. Při zpracování více takových souborů najednou a následné kontrole definovaných pravidel v jejich stromové struktuře narůstala doba zpracování i na několik minut a paměťové nároky dosahovaly několika set megabajtů. Protože nebylo možné v rozumném časovém horizontu najít optimalizaci, která by snížila pozorované hodnoty, **bylo autorem rozhodnuto migrovat již implementované části kódu do jazyka C++**. C++ je multiplatformní jazyk umožňující vývoj výkonných aplikací. Nabízí programátorům pokročilou kontrolu nad systémovými zdroji a pamětí. Jazyk prošel čtyřmi hlavními aktualizacemi v letech 2011, 2014, 2017 a 2020, které přinesly standardy C++11, C++14, C++17 a C++20. [75] Pro implementaci byl zvolen standard C++17, jelikož přinesl knihovnu Filesystem pro provádění operací se souborovými systémy.

6.1.3 Libclang v jazyce C++

Libclang je rozhraní v jazyce C pro Clang a poskytuje relativně malé API, které zpřístupňuje prostředky pro parsování zdrojového kódu do abstraktního syntaktického stromu (AST), načítání již parsovaných AST, procházení AST, přiřazování fyzických umístění zdroje k prvkům v AST a další prostředky, které podporují vývojové nástroje založené na Clangu. [6] Jazyky C a C++ jsou silně typové, a proto využití knihoven vyžaduje mnohem hlubší znalost datových typů a princip jejich fungování.

```
// ...

std::string code = "int main() {return 0;}";
CXIndex index = clang_createIndex(0, 0);
const char* args[] = {"-x", "c", "-std=c99"};

CXUnsavedFile unsaved_file;
unsaved_file.Filename = "fakefile.c";
unsaved_file.Contents = code.c_str();
unsaved_file.Length = code.size();

CXTranslationUnit tu = clang_parseTranslationUnit(
    index,
    "fakefile.c",
    args,
    sizeof(args) / sizeof(const char*),
    &unsaved_file,
    1,
    CXTranslationUnit_None
);

CXCursor cursor = clang_getTranslationUnitCursor(tu);

clang_disposeTranslationUnit(tu);
clang_disposeIndex(index);
```

Zdrojový kód 6.7: Ukázka vytvoření AST v C++

Výše uvedená ukázka kódu 6.7 znázorňuje vytvoření AST z řetězce obsahujícího zdrojový kód v jazyce C. Svoji podobou nepatrně připomíná Python kód 6.5 s tím rozdílem, že je zde nutné, mimo jiné, uvolnit dynamicky alokovanou paměť. S dynamicky alokovanou pamětí je nutné se potýkat i při rekurzivním procházení vygenerovaného stromu. Aby se předešlo únikům paměti, byla implementována pomocná funkce, která převede AST do mnohem přívětivější datové struktury.

[6] viz dokumentace `https://clang.llvm.org/doxygen/group__CINDEX.html`

```
struct ast_location_s {
    unsigned int line;
    unsigned int column;
};

struct ast_token_s {
    ast_location_s location;
    CXTokenKind kind;
    string token;
};

struct ast_node_s {
    int depth;
    ast_location_s location;
    string kind;
    string spelling;
    ast_location_s spelling_location;
    string display_name;
    string type;
    vector<struct ast_token_s> tokens;
    vector<struct ast_node_s> children;
};
```

Zdrojový kód 6.8: Datová struktura pro C++

Porozumění reprezentaci abstraktního syntaktického stromu (viz zdrojový kód 6.8) v aplikaci je klíčové pro celkové pochopení implementace. Struktura `ast_location_s` reprezentuje řádek a sloupec v originálním vstupu. Tokeny jsou reprezentovány strukturou `ast_token_s` a skládají se z lokace, druhu [7] a řetězce, který reprezentují. Samotné uzly AST jsou transformovány do `ast_node_s`. Každý uzel nese informaci o hloubce zanoření, své lokaci, typu, druhu, potomcích, tokenech a dalších. Podstatné je, že je toto uspořádání dat vhodné pro rekurzivní procházení (viz zdrojový kód 6.9).

```
void recursion(const ast_node_s &node) {
    cout << node.spelling << endl;
    for(const ast_node_s &children : node.children) {
        recursion(children);
    }
}
```

Zdrojový kód 6.9: Ukázka rekurze v datové struktuře

[7] viz dokumentace `https://clang.llvm.org/doxygen/group__CINDEX__LEX.html`

6.1.4 Konfigurační soubor

Konfigurační soubor je textový soubor používaný programy k ukládání nastavení a preferencí, která řídí chování aplikací. V současné době existuje mnoho konfiguračních formátů, ale jen některé dosáhly širšího uplatnění. Tyto soubory umožňují uživatelům a administrátorům systému přizpůsobit fungování softwaru bez nutnosti změny jeho kódu, což usnadňuje správu a aktualizace. Extensible markup language (XML) a JavaScript Object Notation (JSON) jsou datové jazyky, které se používají k ukládání dat v jednoduchém textovém formátu, který mohou číst lidé a téměř všechny počítače [76]. XML, JSON, ani jiné formáty, jako YAML či TOML, nejsou nativně v jazyce C++ podporovány. Existují pouze knihovny třetích stran, které práci s těmito formáty umožňují.

Z kapitoly věnující se návrhu aplikace je zřejmé, že konfigurační soubor je částečně strukturován do formátu **klíč-hodnota**. Úložiště typu klíč-hodnota je využíváno v mnoha aplikacích díky vysokému výkonu při zpracování [77]. Na rozdíl od výše zmíněných datových formátů stačí pouze nalézt oddělovač, který určuje, kde končí klíč a začíná hodnota. Ačkoliv ani pro tento druh zápisu dat neexistuje built-in knihovna, tak je možné požadovanou funkcionalitu velmi jednoduše implementovat dle vlastních pravidel.

- Prázdné řádky jsou ignorovány.
- Řádky začínající znakem # jsou ignorovány (komentáře).
- Klíč a hodnota nesmí mít být prázdné.
- Vícenásobné hodnoty pro stejný klíč se slučují.

V programovacím jazyce C++ se přímo nabízí pro ukládání dat ve formátu klíč-hodnota struktura `unordered_map<string, string>`. Pro přiblížení významu definovaných pravidel představuje oddíl 6.10 obsah konfiguračního souboru a zdrojový kód 6.11 výsledek zpracování do datové struktury `unordered_map`.

```
# default setting
tab_size =

# texts
text_column_cs = Sloupec
text_penalty_cs = Penalizace
text_rule_cs = Pravidlo
text_errors_in_file_cs = Chyby v souboru

# rule: for sequence
rule_for_sequence_required = true
rule_for_sequence_sequences = i-j-k
rule_for_sequence_sequences = r-c
```

Zdrojový kód 6.10: Ukázka rekurze v datové struktuře

```
std::unordered_map<std::string, std::string> config = {
    {"text_column_cs", "Sloupec"},
    {"text_penalty_cs", "Penalizace"},
    {"text_rule_cs", "Pravidlo"},
    {"text_errors_in_file_cs", "Chyby␣v␣souboru"},
    {"rule_for_sequence_required", "true"},
    {"rule_for_sequence_sequences", "i-j-k\nr-c"},
};
```

Zdrojový kód 6.11: Výsledek zpracování konfiguračního souboru.

Řádky začínající znakem mřížky a prázdné řádky byly ignorovány. Klíč `tab_size` má prázdnou hodnotu, a proto byl taktéž vynechán. Vícenásobnou hodnotu lze zaznamenat u `rule_for_sequence_sequences`. V tomto případě došlo ke sloučení s oddělovačem nového řádku. Pro úplné pochopení uplatnění pravidel lze nahlédnout do zjednodušené implementace načítání konfigurace 6.12 v jazyce Python.

```python
# read lines
for line in config_file:

    # delete whitespaces
    line = line.strip()

    # skip lines
    if not line or line.startswith('#'):
        continue

    # is there key and value
    if '=' in line:

        # get key and value
        parts = line.split('=', 1)
        key = parts[0].strip()
        value = parts[1].strip()

        # add to structure
        if key and value:
            if key not in config:
                config[key] = value
            else:
                config[key] += "\n" + value
```

Zdrojový kód 6.12: Zjednodušený kód pro načítání konfigurace

6.2 Práce se soubory

Každý soubor obsahující zdrojové kódy ke zpracování lze charakterizovat několika hodnotami. Ze všecho nejdříve je známa cesta k souboru, ze kterého lze načíst jeho obsah po řádcích a nakonec z něho vytvořit abstraktní syntaktický strom kódu. Pro tyto informace lze definovat datovou strukturu znázorněnou níže v 6.13.

```
// structure for file
struct file_s {
    string path;
    vector<string> lines;
    ast_node_s ast;
};
```

Zdrojový kód 6.13: Datová struktura pro načítané soubory

Manipulace se souborovým systémem v nízkoúrovňových programovacích jazycích může být problematická, jelikož je silně ovlivněna typem operačního systému. To znamená, že funkce pro práci se soubory a adresáři se mohou lišit mezi různými systémy, jako jsou Windows a macOS, kde se lze setkat s odlišnými názvy a funkcionalitami. Od verze C++17 je k dispozici knihovna `Filesystem`. Knihovna souborového systému byla původně vyvinuta jako *boost.filesystem*, byla publikována jako technická specifikace ISO/IEC TS 18822:2015 a nakonec byla začleněna do normy ISO C++ od verze C++17. [8] Díky tomuto začlenění je možné pracovat se soubory nezávisle na operačním systému. Za účelem přehlednosti a dodržení principu DRY (Don't Repeat Yourself) byly veškeré operace se souborovým systémem soustředěny do sady funkcí umístěných v souboru `src/file.cpp`. Zde je jejich stručný přehled:

- `custom_create_dir` - Vytvoří nový adresář na zadané cestě.
- `custom_delete_dir` - Odstraní adresář a veškerý jeho obsah na zadané cestě.
- `custom_exist_dir` - Ověří, zda na zadané cestě existuje adresář.
- `custom_create_dest_dir` - Vytvoří všechny adresáře na zadané cestě.
- `custom_get_dirs_in_path` - Vyhledá všechny adresáře na zadané cestě.
- `custom_get_c_files_in_path` - Prohledá rekurzivně zadanou cestu a najde všechny soubory s povolenými příponami.

Podrobný výčet a specifické umístění těchto funkcí v projektu jsou motivovány zkušenostmi z testovací fáze, kdy práce se souborovým systémem na různých operačních systémech představovala hlavní zdroj potíží. Kromě toho je nezbytné, aby zkompilovaná aplikace disponovala adekvátními **oprávněními** pro manipulaci se souborovým systémem, což je důležité pro její bezproblémový běh.

[8] viz dokumentace `https://en.cppreference.com/w/cpp/filesystem`

6.3 Kompabilita

Předchozí sekce, které se zaměřily na výběr programovacího jazyka, knihoven a na práci se souborovým systémem, byly probrány s velkou pečlivostí. Tento postup byl zvolen s cílem objasnit fungování některých procesů uvnitř aplikace. Na základě technické specifikace bylo nezbytné zajistit, aby aplikace mohla být spuštěna v linuxovém prostředí. I přesto bylo díky pečlivému výběru technologií dosaženo stavu, kdy je program teoreticky kompatibilní také s operačními systémy Windows a macOS. Autor věří, že tato univerzální kompatibilita otevírá dveře k širšímu využití.

6.4 Průběh programu

Konceptuální návrh poskytl obecný vhled do toho, jak bude aplikace fungovat. Během vývoje se však vyskytly specifické problémy a vznikly nové nápady, které vyžadovaly individuální řešení. Mezi tyto inovace patří například zavedení podpory pro výstupy v různých jazycích, a také vytvoření zcela nového typu výstupu pro již existující skript, který se využívá v rámci kurzu Procedurálního programování. Níže uvedený seznam popisuje průběh aplikace.

1. **Kontrola počtu argumentů.** Na začátku program kontroluje, zda byl spuštěn s dostatečným počtem argumentů. Pokud ne, vypíše návod k použití a ukončí se.

2. **Načtení konfigurace.** Aplikace se pokusí načíst konfigurační soubor do mapy. Pokud se konfigurační soubor z nějakého důvodu nepodaří přečíst, program vypíše chybovou zprávu a skončí.

3. **Jazyk výstupu.** Program si uloží jazyk z argumentu, ve kterém má být výstup vygenerován.

4. **Výstupní adresář.** Program poté vytvoří výstupní adresář pro uložení výstupů. **Pokud adresář již existuje, je smazán a znovu vytvořen!**

5. **Zpracování podle volby.**

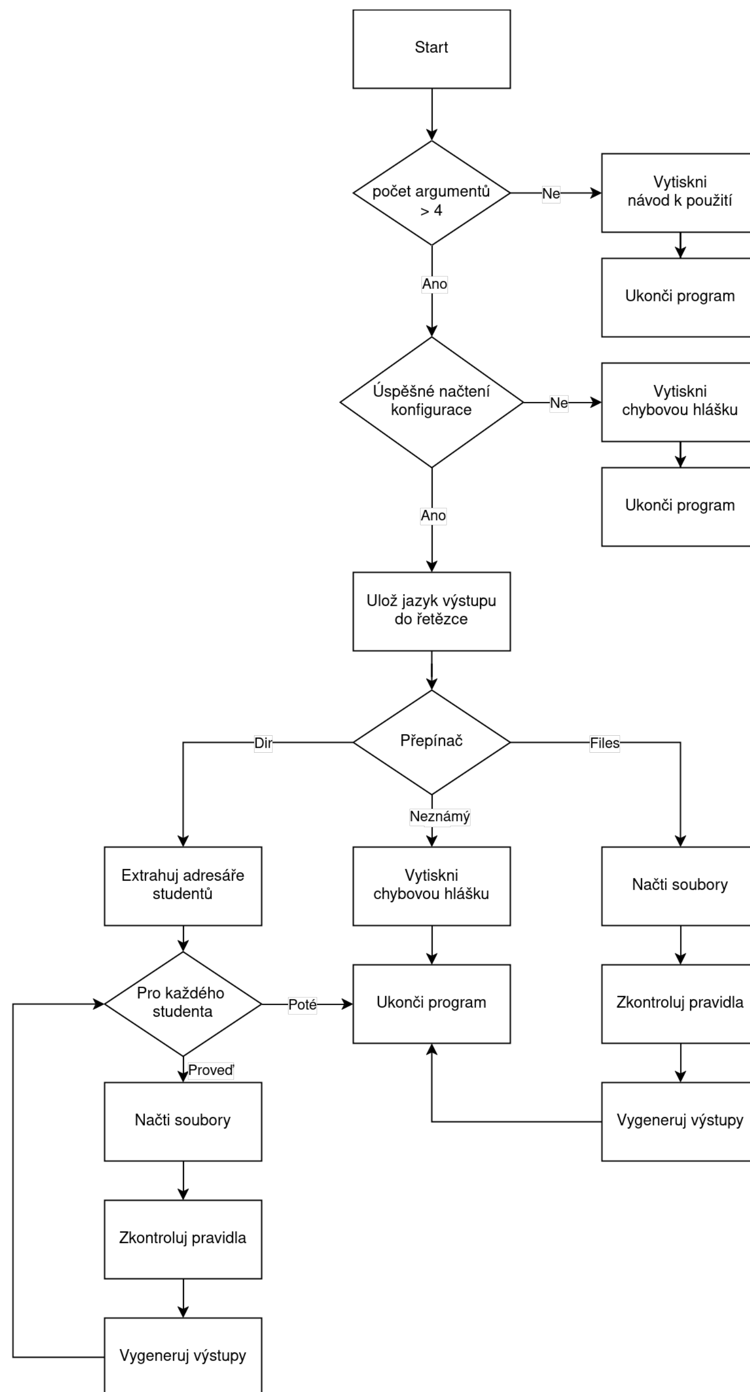
   (a) **Volba `-files`.** Program předá cesty souborů z argumentů do funkce pro zpracování, která aplikuje konfigurační nastavení, vyhodnotí definovaná pravidla a vygeneruje výstupy.

   (b) **Volba `-dir`.** Program ze složky z argumentu extrahuje adresáře jednotlivých studentů a pro každého studenta provede stejný proces jako v případě volby `-files`.

   (c) **Neznámá volba.** Pokud je zadána jiná volba, než je uvedeno výše, program vypíše chybovou zprávu a ukončí se.

6. **Ukončení programu.** Vzhledem k tomu, že program nevyužívá dynamicky alokovanou paměť, může bezpečně a okamžitě ukončit svůj běh.

Diagram 6.1 graficky znázorňuje výše popsaný průběh aplikace.



Obrázek 6.1: Diagram průběhu programu

6.5 Kontrolní výstupy

Před implementací pravidel pro kontrolu stylu zápisu kódu bylo důležité zadefinovat kontrolní funkce schopné zobrazit již získaná data v přehledné vizuální formě. To se týkalo hlavně konfiguračního souboru a analyzovaných zdrojových kódů studentů. Zatímco výpis párů klíč-hodnota či polí řetězců obsahující řádky souborů je triviální, tak pro výpis hodnot abstraktního syntaktického stromu bylo nezbytné implementovat vlastní rekurzivní funkci `print_ast_node` (viz ukázka použití 6.14).

```
// print config
unordered_map<string, string> config;
for(const auto &pair : config) {
    cout << pair.first << "' = '" << pair.second << "'" << endl;
}

// print files
vector<file_s> files;

for(const auto &file : files) {

    // path
    cout << file.path << endl;

    // lines
    for(const auto &line : lines) {
        cout << line << endl;
    }

    // AST
    print_ast_node(file.ast, 0);

}
```

Zdrojový kód 6.14: Kontrolní výpis získaných hodnot

6.6 Pravidla pro psaní čitelného kódu

V kapitole 3 byl představen výčet základních pravidel pro psaní čitelného kódu. Na základě tohoto seznamu byla implementována pravidla, která nejenže tato doporučení částečně implementují, ale také je v některých částech nad rámec původního znění rozšiřují. Pro přehlednost a rozložení problémů do menších částí bylo každé pravidlo implementováno do samostatného souboru ve formě funkce (viz příloha 2). Každá funkce konzistentně přijímá jako vstupní parametry konfiguraci a soubory k analýze a produkuje výstup ve formě seznamu detekovaných chyb. Tyto chyby

jsou reprezentovány pomocí lokace v původním souboru, názvu souboru a příslušné chybové zprávy. Navíc je poskytnuta možnost zvýraznění specifických řádků při zobrazení ve výstupu (viz ukázka 6.15).

```
// rule error
struct rule_error_s {
    unsigned int line;
    unsigned int column;
    vector<int> lines_to_print;
    string message;
    string file;
};

// rule declaration
vector<rule_error_s> rule_name(
    unordered_map<string, string> &config,
    vector<file_s> &files
);
```

Zdrojový kód 6.15: Podoba hlavičky funkce pro pravidla

Každé implementované pravidlo lze deaktivovat. V konfiguračním souboru lze nalézt klíč ve formě `rule_name_required` s hodnotou true nebo false. Některá pravidla jsou založena na statistických metodách, a proto nemusí vždy generovat validní výsledky. Možnost volby, která pravidla budou aktivní a která ne, zvyšuje uživatelskou přívětivost. Výsledná struktura implementovaného pravidla bude tedy odpovídat formátu vyznačenému v 6.16.

```
vector<rule_error_s> errors;

if(config["rule_name_required"].compare("true") == 0) {
    for(const file_s &file : files) {
        // check rule
    }
}

return errors;
```

Zdrojový kód 6.16: Obecná struktura implementovaného pravidla

6.6.1 Konzistentní názvy

Konzistentní názvy v programování znamenají, že každá proměnná a funkce v kódu by měla mít název, který jasně a přesně popisuje její účel nebo obsah. Současně je důležité, aby byl použitý jednotný styl pojmenování v celém kódu. To se týká nejen proměnných a funkcí, ale i konstant, pro které je obvyklé používat konvenci UPPER_CASE. Dílčí problémy byly pro přehlednost implementovány jako níže popsané funkce.

Ověření **názvu proměnné** pomocí strojových metod je složité, protože jeho význam je často odvozen z běžných slov nebo zkratek v nějakém jazyce. V programování mohou být použity i velmi krátké názvy proměnných, včetně jednopísmenných, které mají svůj specifický význam. Z těchto důvodů je nezbytné provádět ověřování názvů proměnných manuálně. K tomuto účelu byla vytvořena funkce `print_stats`, která implementuje rekurzivní procházení abstraktního syntaktického stromu programu a vypisuje názvy všech deklarovaných funkcí a proměnných spolu s jejich datovými typy. U proměnných jsou navíc uvedeny tokeny, ze kterých se deklarace skládá.

```c
#include <stdio.h>

// main function
int main(void) {

    int i = 1;
    int j = 0;

    return i + j;

}
```

Zdrojový kód 6.17: Ukázka vstupu pro funkci print_stats

Pro ukázku mějme jednoduchý zdrojový kód 6.17 s hlavní funkcí `main`, v níž jsou deklarovány proměnné `i` a `j`. Jak je níže naznačeno, `print_stats` vypíše pro každou nalezenou funkci její název a všechny proměnné v ní deklarované.

```
File:test/test.c

    Function name: main
    Data types: int (void)

    Variables:
        Variable: i | Type: int | Tokens: int i = 1
        Variable: j | Type: int | Tokens: int j = 0
```

Jednotný styl pojmenování byl implementován v samostatné funkci `rule_naming_convention`. Tato kontrola se soustředí primárně na funkce a proměnné definované programátorem. Názvy mohou být klasifikovány do několika kategorií:

- Camel case
- Snake case
- Screaming snake case
- Pascal case
- Neznámý styl

Ačkoli je možné argumentovat, že existují i další styly pojmenování, jejich výskyt je natolik vzácný, že je lze zanedbat. Konstanty mohou být podle pravidel pojmenovány pouze ve stylu screaming snake case, zatímco názvy spadající do kategorie neznámého stylu jsou automaticky považovány za chybné. Poté se nalezne nejčastěji používaný styl pojmenování a ostatní kategorie jsou identifikovány jako odchylky. Implementace se musela vypořádat s faktem, že Clang považuje definování vlastní datové struktury či výčtového typu za deklaraci proměnné. Pro správné vyhodnocování bylo nutné kontrolovat uzly AST, zda jejich tokeny neobsahují příslušná klíčová slova programovacího jazyka.

```
// the most used style
int my_snake_function(void) {

    // valid constant name
    const int MY_CONST = 1;

    // different styles will be marked as error
    int PascalCaseVar, camelCaseVar;

    // the most used style
    int my_snake_1, my_snake_2, my_snake_3, my_snake_4;

    return 0;

}
```

Zdrojový kód 6.18: Ukázka konvencí pojmenování pro vyhodnocení pravidlem

Ukázka 6.18 obsahuje správně pojmenovanou konstantní proměnnou `MY_CONST` a nejpočetněji zastoupenou konvencí pojmenování je snake case, a proto proměnné `PascalCaseVar` a `camelCaseVar` budou označeny za chybné.

Konzistence názvů proměnných je důležitá i v případě **vnořených `for` smyček**. Obvyklou praxí je používat pro indexy sekvenci proměnných `i-j-k` nebo, v případě iterace přes matice, `r-c` (řádky a sloupce). Použití jiných proměnných je neobvyklé. Vzhledem k tomu, že nelze jednoznačně stanovit, které sekvence jsou vhodné, a které ne, nabízí konfigurační soubor možnost definovat do `rule_for_sequence_sequences` přípustné sekvence proměnných, které by měly být v kódu uznány za správné. Implementace vyhledávání vnořených cyklů nebyla nikterak složitá do momentu, kdy nebyly součástí maker.

```
#define MACRO_FOR for(int l = 0; l < 3; l++) {}

for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++) {
        MACRO_FOR
    }
}

for(int i = 0; i < 3; i++) {
    for(int j = 0; j < 3; j++) {
        for(int l = 0; l < 3; l++) {}
    }
}
```

Zdrojový kód 6.19: Ukázka implementace for cyklu s makry

Zdrojový kód 6.19 zahrnuje dva segmenty s vnořenými for cykly. Důležitým faktem je, že se AST generuje až po dokončení preprocessingu. V důsledku toho by oba bloky s vnořenými cykly obsahovaly sekvenci indexů `i-j-l`. Avšak z hlediska správnosti kódu by chyba měla být identifikována pouze ve druhém bloku cyklů, protože ten první formálně obsahuje validní sekvenci `i-j`. Pro správnou identifikaci chyb je tedy nutné využívat informace o umístění kódu ve zdrojovém souboru. Makra jsou definována před vnořenými cykly, a tedy posloupnost řádků je narušena.

6.6.2 Formátování

Zatímco základní pravidla zdůrazňují omezení jako maximální počet znaků na řádku a využití pouze ASCII znaků, tak skutečný rozsah těchto pravidel je podstatně větší. Jedná se například o psaní mezery za některými slovy a čárkami, což je obvyklé i v běžném jazyce. Do sekce formátování také patří správné umístění složených závorek, vizuální oddělení operátorů od ostatních částí kódu a další. Jak je z uvedeného přehledu patrné, pravidel pro formátování je mnoho. Implementovat je souvisle do jedné funkce by bylo velmi složité a nepřehledné. Proto bylo rozhodnuto rozdělit tyto konvence do pokud možno nejmenších celků a implementovat je formou funkcí.

ASCII znaky jsou univerzálně podporovány a jejich použití zajistí, že kód bude fungovat správně na různých platformách a systémech bez problémů s kódováním. Funkce `rule_ascii_character` iteruje soubor řádek po řádku a každý řádek `char` po `charu`. Char má velikost jednoho bajtu. Jakýkoliv bajt s hodnotou vyšší než 127 není součástí standardní ASCII tabulky a je považován za chybu. Aby se předešlo vzniku příliš mnoha chyb, tak se označí maximálně jeden non-ASCII znak na řádek (viz zdrojový kód 6.20). Další výhodou tohoto přístupu je, že znakové sady, které používají pro jeden znak více bajtů zpravidla začínají bajtem o hodnotě vyšší než 127, a tudíž je implementace funkční pro běžné znakové sady.

```
for(size_t i = 0; i < file.lines.size(); i++) {
    for(size_t j = 0; j < file.lines[i].size(); j++) {
        if((unsigned char) file.lines[i][j] > 127) {
            // error
            break;
        }
    }
}
```

Zdrojový kód 6.20: Ukázka z implementace pravidla pro kontrolu ASCII znaků

Nejzákladnější pravidlo o **maximálním počtu znaků na řádek** je zavedeno funkcí `rule_line_length`. Limit je nastavitelný v konfiguračním souboru pomocí proměnné `rule_line_length_max_length`, jelikož se může lišit v závislosti na projektu. Délka řádku se měří v bajtech, ne v počtu znaků, což může způsobit problémy při použití různých znakových sad. Je důležité si uvědomit, že toto pravidlo je komplementární s pravidlem pro ASCII znaky. Bude tedy funkční pouze pokud jsou používány standardní znaky.

V tomto odstavci budou popsána **gramatická pravidla** reprezentovaná prostřednictvím funkcí `rule_space_after_comma`, `rule_space_after_statement` a `rule_space_around_operator`. Jak již jejich název napovídá, kontrolují existenci mezer za čárkami, některými klíčovými slovy jazyka C a binárními operátory tak, jak je obvyklé v přirozeném jazyce. Pro ilustraci významu formátování kódu byl připraven jednoduchý příklad 6.21, který prezentuje dva identické IF bloky. V prvním bloku byly mezery úmyslně vynechány, zatímco ve druhém byly vloženy standardním způsobem. Rozdíl v čitelnosti mezi těmito dvěma příklady je zřetelný a ukazuje, jak významně může formátování ovlivnit přehlednost kódu.

```
if(1==1&&10>0||0==0) {
    printf("%d␣%d␣%d",0,1,2);
}

if (1 == 1 && 10 > 0 || 0 == 0) {
    printf("%d␣%d␣%d", 0, 1, 2);
}
```

Zdrojový kód 6.21: Ilustrace významu mezer v kódu

V neposlední řadě se formátování kódu týká i umístění složených závorek funkcí a příkazů, jejichž kontrola byla implementována ve formě pravidel `rule_function_parentheses`, `rule_main_parentheses` a `rule_statement_parentheses`. Důležité je, že složené závorky hlavní funkce `main` by měly být umístěny na samostatných řádcích s identickým odsazením (viz zdrojový kód 6.22). Pro ostatní funkce a příkazy není striktně určeno, zda má být otevírací závorka na novém či stejném řádku. Hlavním požadavkem je, aby byl zachován konzistentní styl zápisu v celém souboru. V programování se lze setkat s různými styly odsazení složených závorek, avšak nejčastější praxí je umisťování závorek buď na stejný nebo na nový řádek s odpovídajícím odsazením, a proto budou ostatní způsoby považovány za chybné.

```
int main(void)
{
    if(1 == 1) {
        return 1;
    }
    else {
        return 0;
    }
}

int func1(void) {
    return 0;
}

int func3(void)        // invalid
  {
    return 0;
  }
```

Zdrojový kód 6.22: Různé styly zápisu složených závorek

Nejzřetelnějším pravidlem formátování je odpovídající **odsazení zdrojového kódu**. Každý blok kódu by měl být odsazen stejným počtem mezer nebo tabulátorů. Implementace pravidla `rule_function_indent`, které by provádělo kontrolu, přinesla řadu výzev. Nejprve bylo potřeba určit průměrné odsazení, což bylo komplikované kvůli možné variabilitě. Dále, analýza AST neposkytovala jednoduchou metodu pro určení míry zanoření, což si vyžádalo spoléhání se na tokeny a klíčová slova programovacího jazyka, která umožňují zanoření. Kvůli možnosti zalomení řádků bylo problematické označit za chybu odsazení, které je větší než očekávané. Implementace tedy spočívá ve sledování toku tokenů v rámci funkcí, přiřazování složených závorek ke klíčovým slovům a výpočtu hloubky odsazení. Řádky odsazené menším počtem bílých znaků, než je očekáváno, jsou považovány za chybné.

6.6.3 Komentáře

Složitější části kódu by měly být doplněny komentářem, ale rozhodnout, které části jsou natolik komplexní, aby vyžadovaly komentář, může být náročné. V reakci na tuto výzvu byla po konzultaci se zadavatelkou práce rozšířená funkce `print_stats`. Tato funkce po reimplementaci nejenže zobrazuje počet řádků každé funkce, ale také vypisuje několik tokenů před její hlavičkou, aby bylo možné zjistit, zda byla adekvátně okomentována. Touto metodou lze identifikovat příliš velké celky, které by mohly naznačovat neefektivní rozdělení práce, a tím vybídnout k manuální revizi zdrojového kódu. Níže je uvedený nový výstup pro zdrojový kód 6.17.

```
Function name: main
Data types: int (void)
Tokens: main ( void )
Is declaration: 0
Number of lines: 6

10 tokens before function (to check if there is comment):
    Token: #
    Token: include
    Token: <
    Token: stdio
    Token: .
    Token: h
    Token: >
    Token: // main function
    Token: int
    -> Token: main
    Token: (
    Token: void
    Token: )
    Token: {

Variables:
    Variable: i    | Type: int    | Tokens: int i = 1
    Variable: j    | Type: int    | Tokens: int j = 0
```

6.6.4 Makra

Makro je fragment kódu, kterému je přiřazen identifikátor ve formě řetězce. Podstatné je, že preprocesor nahradí všechna makra ve zdrojovém souboru jejich obsahem, a proto je nezbytné pracovat s tokeny na místo AST. Pravidla vyžadují, aby **netriviální numerické literály** byly reprezentovány makry. Důvod tohoto pravidla implementovaného ve funkci `rule_magic_constant` je například zjevný v programu pro výpočet obvodu kruhu (viz zdrojový kód 6.23).

```
#include <stdio.h>
#define PI 3.1415926

double circumferenceOfCircle1(double radius) {
    return 2 * PI * radius;
}

double circumferenceOfCircle2(double radius) {
    return 2 * 3.1415926 * radius;
}

int main() {
    double radius = 8.07;
    return circumferenceOfCircle1(radius);
}
```

Zdrojový kód 6.23: Program pro výpočet obvodu kruhu

Používání velkých písmen pomáhá v kódu jasně odlišit **makra** od proměnných, funkcí a jiných identifikátorů (viz zdrojový kód 6.23). Jelikož jsou makra předzpracovávána a nahrazují kód před kompilací, mohlo by dojít k nechtěným konfliktům s názvy proměnných nebo funkcí, pokud by byly pojmenovány stejně. Za tímto účelem bylo implementováno pravidlo `rule_macro_uppercase`, které tuto široce akceptovanou konvenci kontroluje.

Druhým bodem této sekce je používání speciální hodnoty **NULL** pro reprezentaci nulového ukazatele. NULL jednoznačně signalizuje, že ukazatel neodkazuje na platnou paměťovou adresu, zatímco numerická nula by mohla být zaměněna za platnou hodnotu. Pro implementaci pravidla `rule_pointer_comparison_to_zero`, která toto kontroluje, je nezbytné porozumět reprezentaci dat v AST. Při porovnávání hodnot v kódu se často používají binární operátory. Pro správnou funkci binárního operátoru musí být k dispozici dva potomci. Pokud je jeden z těchto potomků ukazatel, správná praxe vyžaduje, aby i ten druhý byl ukazatel. Při analýze každého uzlu je možné prozkoumat jeho tokeny. Jestliže je potomek tvořen jediným literálem s číselnou hodnotou nula, signalizuje to místo, kde by měla být použita hodnota NULL. Následuje příklad zjednodušeného výřezu z abstraktního syntaktického stromu, kde je ukázán uzel typu *BinaryOperator* spolu se svými dvěma potomky.

```
Node Depth: 4, Kind: 'BinaryOperator'
Token: 'ptr'
Token: '=='
Token: '0'
Node Depth: 5, Kind: 'UnexposedExpr', Type: int *
    Token: 'ptr'
Node Depth: 5, Kind: 'UnexposedExpr', Type: int *
    Token: '0'
```

Při implementaci tohoto pravidla nastával velmi vzácný jev, kdy vytvářený program končil segmentační chybou. Bylo zjištěno, že Clang není v některých případech schopen správně tokenizovat. Tento problém vznikal především ve spojení s makry a konstrukty jazyka. Aby se předešlo tomuto problému, bylo nezbytné některá již implementovaná pravidla rozšířit o explicitní kontrolu počtu tokenů.

6.6.5 Struktura kódu

Struktura kódu se týká způsobu, jak je program organizován. Jedním z přístupů je **rozdělení programu do krátkých a jednoduchých funkcí**. To souvisí s již implementovaným pravidlem na komentování složitějších částí kódu v `print_stats`, kde se zobrazují velikosti funkcí.

Po pečlivém prozkoumání požadavku na **přehlednou logiku ukončení cyklů** bylo zjištěno, že vytvoření přesného pravidla není možné. Mohlo by se například navrhnout, aby příkazy pro ukončení cyklu byly umístěny vždy na začátku bloku kódu. Toto řešení by však nebylo univerzálně aplikovatelné, protože v některých situacích, jako je iterace přes prvky matice, může být nutné cyklus ukončit až po splnění určitých podmínek v průběhu cyklu.

Dobrá organizace a rozčlenění kódu do krátkých a jednoduchých funkcí předchází jeho opakování. Pro zjištění, zda **kód neobsahuje duplicitní (velmi podobné) bloky**, je možné využít strukturu abstraktního syntaktického stromu. Princip spočívá v rozdělení AST na všechny možné podstromy, které se následně seřadí podle velikosti, tedy podle počtu uzlů, z nichž jsou složeny. Hledáme první dva podstromy, které jsou identické v tom smyslu, že se typy jejich uzlů shodují. Pokud takové dva podstromy najdeme, identifikovali jsme části kódu, které jsou si podobné. Míru podobnosti lze určit velikostí podstromu. Konfigurační soubor umožňuje nastavit míru pro implementované pravidlo skrze parametr `rule_duplicates_threshold`, jehož výchozí hodnota je nastavena na 20. Takto formulovaný proces kontroly může být v případě velkých vstupů výpočetně náročný. Nicméně žádný z testovaných zdrojových kódů nevedl k významnému zpomalení aplikace.

```
{
    Node *to_free = queue_ptr->head;
    queue_ptr->head = queue_ptr->head->next;
    queue_ptr->clear(to_free->data);
    free(to_free);
    queue_ptr->size--;
    removed = true;
}
// ...
{
    Node *to_free = current->next;
    current->next = current->next->next;
    queue_ptr->clear(to_free->data);
    free(to_free);
    queue_ptr->size--;
    removed = true;
}
```

Zdrojový kód 6.24: Ukázka dvou podobných bloků kódu

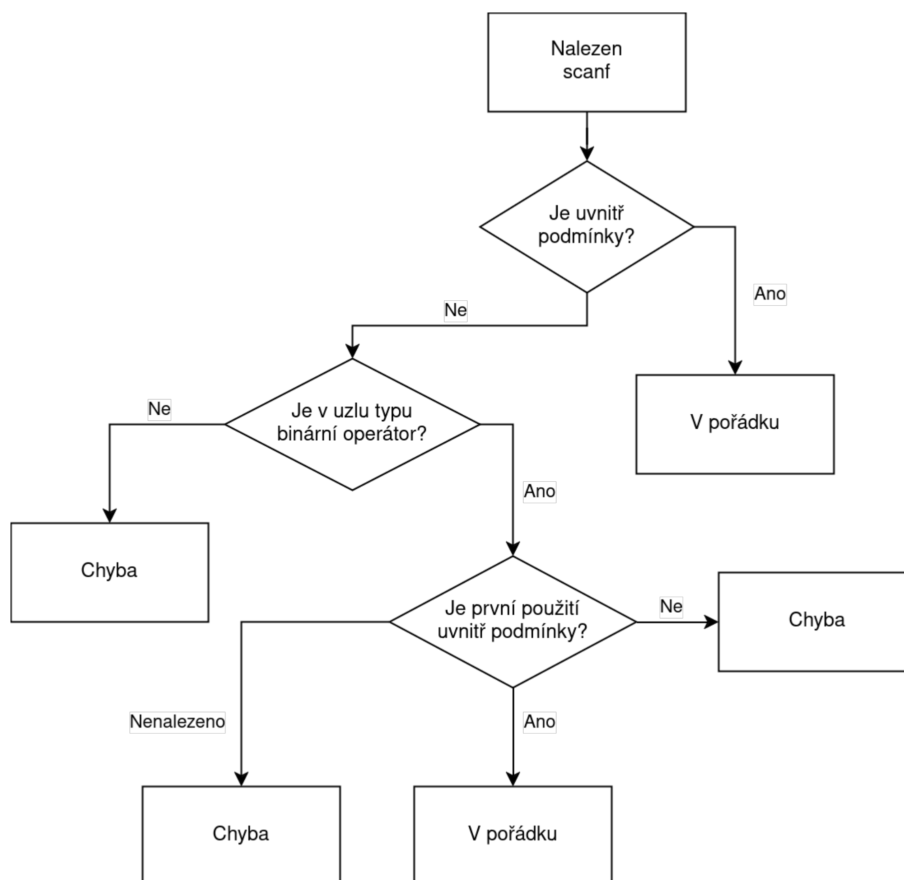
Výše uvedená ukázka zdrojového kódu 6.24 byla implementovaným pravidlem `rule_duplicates` vyhodnocena správně. Z kódu je zřejmé, že by bylo možné tuto část kódu implementovat do samostatné funkce. Avšak v určitých případech může být podobnost mezi stromy náhodná, což vede k nesprávnému vyhodnocení, jak ukazuje 6.25. To naznačuje, že tento přístup má své silné i slabé stránky.

```
for(int i = 0; i < m_count; i++)
{
    if(op_array[i] != '0').
        return i;
}
// ...
for(int i = 0; i < m_count; i++)
{
if(op_array[i] == '*')
    return i;
}
```

Zdrojový kód 6.25: Ukázka dvou podobných bloků kódu, které se liší

Závěr této sekce se bude věnovat **kontrole vstupů v programu**, jejíž implementace se ukázala jako velmi komplikovaná. Běžně se data čtou ze standardního vstupu pomocí funkcí typu `scanf`, které vrací počet úspěšně načtených položek. Lze se setkat se dvěma přístupy. `Scanf` může být umístěn přímo v podmínce nebo mimo ni. V prvním případě lze předpokládat, že návratová hodnota je ihned zkontrolována. Ve druhém případě je důležité zkontrolovat, zda byla návratová hodnota uložena pro pozdější ověření. Ať už byla uložena do nově deklarované proměnné, již existující proměnné, pole nebo datové struktury, tak reprezentace v AST bude rozdílná. Podstatné je nejen ověřit, že návratová hodnota byla zkontrolována, ale také, že kontrola byla provedena v souladu s pravidly *coding style*.

Pravidlo `rule_xscanf` vyhledává v abstraktním syntaktickém stromu zdrojového kódu výskyty funkcí typu `scanf`. Pokud je volání funkce již umístěno přímo v podmínce, pravidlo ho ignoruje. V ostatních případech by mělo být volání `scanf` součástí uzlu typu binárního operátoru. Pokud se volání funkce nenachází na takové lokaci, interpretuje se to jako ignorování návratové hodnoty. Z binárního operátoru lze dále určit, do jaké proměnné byl výsledek uložen, přičemž je třeba vzít v úvahu, že název proměnné může být složen z více tokenů. Nakonec se analyzuje blok kódu, kde došlo k volání, aby se našlo místo prvního použití. Pokud první použití proměnné není v rámci podmínky, označí se tato situace jako chyba (viz diagram 6.2).



Obrázek 6.2: Diagram pravidla scanf

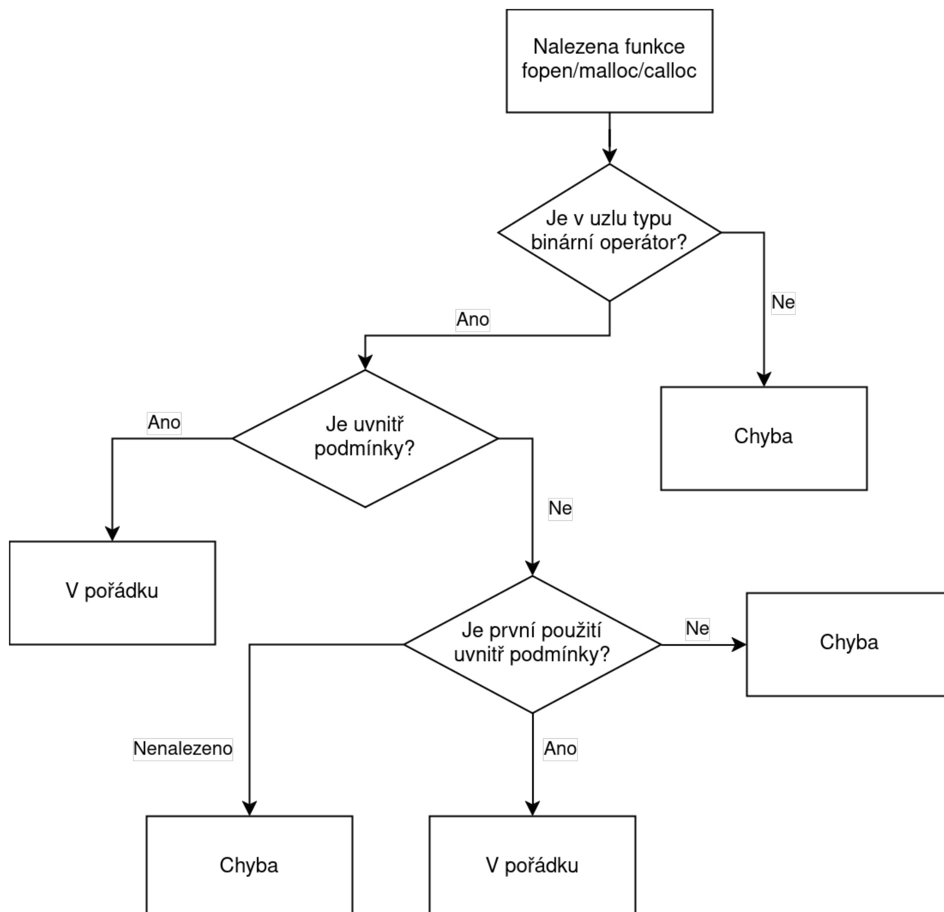
Volání funkcí `scanf` a ověření jejich výsledků se provádí v rámci **stejného bloku kódu**. Pokud by se kontrola uskutečnila až později nebo v jiné funkci, mohlo by být pro čtenáře nejasné, zda k této kontrole vůbec došlo.

6.6.6 Lokálnost

Princip lokálnosti v programování znamená, že proměnné by měly být definovány co nejblíže místu svého prvního využití, aby byla doba jejich životnosti co nejkratší. Aplikovat toto pravidlo v praxi by znamenalo vytvoření grafu, který by vykresloval vztahy mezi proměnnými, a implementaci dalších funkcionalit pro rozpoznání ideálního umístění v kódu. Realizace lokálnosti by byla náročná a její rozsah by odpovídal samostatné bakalářské práci. Jako kompromis byly alespoň implementovány funkce `rule_var_not_used` a `rule_global_variable`, které detekují nepoužité proměnné a nekonstantní globální proměnné.

6.6.7 Práce s pamětí

Práce s pamětí v programovacím jazyce C, se týká především **dynamické alokace paměti** a **manipulace se soubory**, což zahrnuje volání funkcí jako `malloc`, `calloc` a `fopen`. Je důležité, aby programátoři po použití těchto funkcí neodkladně kontrolovali návratové hodnoty. Tento postup je podobný již existujícímu pravidlu `rule_xscanf`, které vyžaduje, aby se návratové hodnoty těchto funkcí ověřovaly. V tomto případě je navíc nezbytné, aby se návratové hodnoty vždy ukládaly do proměnných, jelikož bude nutné je později uvolnit. Diagram 6.3 znázorňuje princip fungování pravidel `rule_file_open` a `rule_memory_allocation`, která vychází z implementace `rule_xscanf`.



Obrázek 6.3: Diagram pravidel pro práci s pamětí a soubory

Jak již bylo uvedeno výše, je důležité **alokovanou paměť** také **řádně uvolnit**. Pro ověření, zda k uvolnění paměti skutečně došlo, existují nástroje jako Valgrind. Z hlediska kódovacího stylu by se volání funkce pro uvolnění paměti mělo v programu pro každou proměnnou objevit jen jednou, a to v rámci přehledné logiky, která k tomuto volání vede. Toto omezení je implementováno v pravidlech `rule_file_close` a `rule_memory_deallocation`.

6.7 Adresářová struktura projektu

Pro správu projektu byl zvolen **GitHub**. Git a GitHub jsou vývojové platformy, které pomáhají vývojářům hostovat a kontrolovat kód, spravovat projekty a vytvářet software společně s miliony dalších vývojářů [78].

Dobře organizované a snadno pochopitelné C++ projekty na GitHubu dodržují určitou strukturu adresářů a souborů. V kořenovém adresáři projektu na GitHubu by měl být umístěn soubor `README.md`, který je napsán v Markdownu a poskytuje základní přehled o projektu, instrukce k sestavení a další důležité informace. Dále by zde měl být soubor `.gitignore`, který informuje Git o souborech nebo adresářích, které mají být ignorovány, například sestavené binární soubory nebo závislosti. Pokud projekt využívá systém pro automatizovanou kompilaci, jako je Make, je vhodné umístit příslušné konfigurační soubory přímo do kořenového adresáře. Adresář `src` slouží k uložení zdrojových kódů. Nacházejí se zde všechny `.cpp` soubory tvořící logiku aplikace. Paralelně, adresář `include` obsahuje hlavičkové soubory `.hpp` definující rozhraní tříd, funkcí a dalších komponent. Na základě těchto konvencí bude adresářová struktura vypadat následovně:

- `include` - Adresář pro hlavičkové soubory; z velké části svým složením koresponduje s adresářem `src`
- `src` - Adresář pro zdrojové kódy projektu
  - `main.cpp` - Hlavní logika programu
  - `output.cpp` - Funkce generující výstupy programu
  - `config.cpp` - Funkce pro načítání konfigurace programu
  - `file.cpp` - Funkce pro práci se souborovým systémem
  - `ast_util.cpp` - Funkce pro generování AST
  - `rule_*.cpp` - Soubory obsahující implementovaná pravidla
- `tests` - Adresář určený pro testovací kód; testování bude popsáno v pozdějších kapitolách
- `.gitignore` - Soubor, který určuje, jaké soubory nebo adresáře mají být ignorovány při verzování
- `README.md` - Markdown soubor poskytující úvod do projektu a instrukce k instalaci
- `config.conf` - Konfigurační soubor typu klíč-hodnota obsahující proměnné nutné ke správnému běhu aplikace
- `makefile` - Soubor určený pro `make`; definuje pravidla pro sestavení projektu a další závislosti

6.8 Spuštění programu

Výsledná aplikace bude spuštěna z příkazové řádky a informace jí budou předány ve formě argumentů. Argumenty příkazové řádky jsou hodnoty, které se předávají programu při jeho spuštění z příkazové řádky (nebo terminálu). Tyto argumenty umožňují specifikovat konfiguraci nebo chování programu bez nutnosti měnit jeho kód. Program bude očekávat alespoň pět argumentů včetně názvu programu:

- `<name>` - Název programu.
- `<lang>` - Jazyk, ve kterém budou vygenerovány výstupní soubory. Prozatím je podporován pouze **český jazyk**, jehož zkratka je `cs`. Jazykové mutace lze vytvářet v konfiguračním souboru. Jednotlivé klíče textů končí jazykem, ve kterém jsou napsány. Například `text_column_cs` nebo `text_reason_cs`.
- `<path/to/config>` - Cesta ke konfiguračnímu souboru, který obsahuje nastavení nezbytná pro správnou funkci programu. Cesta může být absolutní i relativní, ale ve druhém případě je nutné vzít v potaz, že cesta začíná v adresáři, ze kterého je program spuštěn.
- `<option>` - Specifikuje, jaký druh zpracování se má provést. Program podporuje dvě možnosti:
  - `-files` - Indikuje, že další argumenty budou cesty k jednotlivým souborům určeným ke zpracování.
  - `-dir` - Označuje, že následující argument je cesta k adresáři, který obsahuje soubory studentů ke zpracování.
- `<paths>` - V závislosti na předešlém argumentu jsou očekávány buď cesty k jednotlivým souborům, nebo cesta k určitému adresáři. Pro možnost `-files` jsou očekávány další argumenty ve formě cest ke zdrojovým kódům. Pro možnost `-dir` je očekáván **jeden** argument určující cestu k adresáři.

Z výše uvedeného popisu lze odvodit obecný zápis argumentů v příkazové řádce, tedy `./application <lang> <path/to/config> <option> <paths>`. Následuje několik ukázek použití. Povšimněte si prosím pozorně, že přepínač `-dir` označuje **adresář**. Tento adresář je výsledkem rozbalení TGZ souboru ze systému BRUTE, který poskytuje data v komprimované podobě.

```
./application cs config.conf -dir /Downloads/HW04
./application cs config.conf -dir ./data/HW01
./application cs config.conf -files ./test/test.c
./application cs config.conf -files ./test/test.c ./test/main.c
```

6.9 Výstup programu

Výstup se stal **nejdůležitější** a nejdéle implementovanou částí aplikace, jelikož bylo nezbytné transformovat získané poznatky do přehledné formy, která by jasně a stručně poskytla všechny informace. Z návrhu aplikace je patrné, že výstup má být uložen do souboru. Nicméně neříká nic o formátu. Základním požadavkem bylo, aby byl soubor čitelný na různých zařízeních. Formát **PDF** byl z volby vyloučen, jelikož není schopen zobrazit příliš dlouhé řádky. Také v programování oblíbené soubory formátu **Markdown** byly vyloučeny, jelikož s nimi neumí zacházet všichni uživatelé a pro jejich zobrazení je nezbytný pokročilý textový editor. Nakonec byl pro výstup vybrán **standardní textový soubor**, jehož obsah se bude vhodně formátovat. Návrh také počítá se dvěma druhy výstupu - pro studenta a pro vyučujícího.
V pozdější fázi projektu přibyl požadavek na **další formát výstupu**, který by dokázal zpracovat již existující skript využívaný v rámci některých kurzů programování.

6.9.1 Výstup pro studenty

Pro organizaci dat v souboru byly k dispozici dvě strategie. První možností bylo seřadit identifikované chyby chronologicky, podle jejich pořadí ve vstupním souboru. Druhá možnost spočívala v seskupení chyb dle pravidel. Druhá varianta se ukázala být výhodnější, jelikož na začátek každého pravidla bylo možné vypsat krátký odstavec, který čtenáře seznamuje s kladenými požadavky. Například:

```
Pravidlo: Složené závorky funkce main()
==========================================

V programovacím jazyce C se doporučuje, aby složené závorky
funkce main byly na novém řádku se stejným odsazením. Tato
konvence usnadňuje vizuální oddělení bloku kódu a umožňuje
snadnější sledování struktury programu.
Ukázka:
int main(void)
{
    // ...
}
```

Samotné popisy a jejich jazykové mutace jsou definovány v konfiguračním souboru a lze je snadno modifikovat dle potřeby. Chyby, které spadají pod stejné pravidlo, jsou organizovány dle vstupních souborů. U každé chyby je zobrazen kontext, místo chyby je zvýrazněno ukazatelem a dále je uveden krátký popis spolu s hodnotou penalizace. Míra penalizace je předem definována v konfiguračním souboru pro každé pravidlo. Výstupní soubor bude vytvořen ve složce `output` s názvem `student_<jazyk>.txt`. Výstup pro studenta může vypadat následovně:

```
|
| Výstup kontroly pravidel stylu zápisu programu
|


   Pravidlo: Magické konstanty
   ============================

   Používání magických konstant, tj. pevně zadaných čísel přímo v kódu,
   je nevhodné ze dvou hlavních důvodů. Prvním důvodem je nesrozumitelnost
   kódu, protože tato čísla mohou být obtížně interpretovatelná bez dalšího
   kontextu. Dále mohou vést k chybám v kódu. Pokud je třeba změnit hodnotu
   konstanty v různých částech programu, může se snadno stát, že zapomeneme
   aktualizovat všechny její výskyty.

    Chyby v souboru: ukoly/PRP_HW02/user1/main.c

        55 |           return 100;
           | -----------------^---
           | Magické konstanty nejsou povoleny. (-0.1)

   Pravidlo: Makra
   ===============

   Makra by měla být psána velkými písmeny, aby bylo na první pohled zřejmé,
   že se jedná o makra a nikoli o obyčejné proměnné nebo funkce. Velká
   písmena slouží jako vizuální signalizace, která upozorňuje na to,
   že makra mohou mít speciální chování a že jejich použití může mít jiné
   důsledky než použití běžných proměnných nebo funkcí.

    Chyby v souboru: ukoly/PRP_HW02/user1/main.c

         7 | #define MaxNum 1000000
           | --------^---
           | Názvy maker musí být zapsány velkými písmeny. (-0.1)

   Součet penalizace: -0.2
```

6.9.2 Výstup pro vyučující

Výstup pro vyučující je pro přehlednost rozdělen do dvou souborů: `teacher_<jazyk>.txt` a `statistics.txt`. Dříve v této kapitole byla popsána funkce `print_stats`, která vypisuje informace o zdrojovém kódu pro ruční kontrolu. Výstup této funkce je uložen právě do souboru `statistics.txt`. Jelikož učitelé detailně znají pravidla pro psaní čitelného kódu, není nutné, aby jejich výstupní soubory obsahovaly doprovodné popisy. To činí výpisy více technickými a zároveň lépe čitelnými. Úpravou předchozí ukázky pro studenta vznikne následující výstup.

```
|
| Výstup kontroly pravidel stylu zápisu programu
|


    Pravidlo: Magické konstanty
    ============================

     Chyby v souboru: ukoly/PRP_HW02/user1/main.c

        55 |           return 100;
           | -----------------^---
           | Magické konstanty nejsou povoleny. (-0.1)

    Pravidlo: Makra
    ===============

     Chyby v souboru: ukoly/PRP_HW02/user1/main.c

         7 | #define MaxNum 1000000
           | --------^---
           | Názvy maker musí být zapsány velkými písmeny. (-0.1)

    Součet penalizace: -0.2
```

6.9.3 Výstup pro skript

Skript v systému BRUTE zpracovává pomocí regulárních výrazů zdrojový kód doplněný o komentáře ve specifickém tvaru, který začíná znakem dolaru a obsahuje číslo pravidla, vážnost, barvu a popis (viz ukázka 6.26).

```
int a;    //$<rule_number>,<severity>,<color> <comment>
```

Zdrojový kód 6.26: Ukázka formátu komentáře pro zpracování

Každé pravidlo uvedené v konfiguračním souboru zahrnuje tyto parametry, které lze upravit podle individuálních požadavků. Pro každý vstupní soubor se zdrojovými kódy vznikne i výstupní, jehož název bude složen z cesty toho původního, aby nedošlo k překrývání jmen. Obsahem se budou mírně lišit. Nejenže přibydou nové komentáře, ale formátování bílých znaků bude jiné, jelikož program vnitřně nahrazuje tabulátory za mezery.

6.10 Shrnutí

Kapitola poskytla ucelený náhled do vývoje aplikace, přičemž se zaměřila na celý proces od výběru nástrojů a technologií až po konkrétní fáze vývoje a implementaci pravidel pro psaní čitelného zdrojového kódu. V úvodu kapitoly byl zdůrazněn význam výběru programovacího jazyka a externí knihovny na celkový výkon aplikace. Dále se kapitola věnovala struktuře projektu a formě spuštění programu. Konfigurace byla realizována prostřednictvím konfiguračního souboru ve vlastním formátu typu klíč-hodnota. Podstatnou částí kapitoly se stal popis zpracování vstupních souborů do vnitřní struktury včetně práce se souborovým systémem, na které pozvolna navázaly průběh programu a nástroje pro kontrolní výpisy. Závěr kapitoly se podrobně věnoval implementovaným pravidlům pro psaní čitelného zdrojového kódu včetně formátu jejich výstupu pro studenty a vyučující.

Kapitola 7

Testování

Testování hraje zásadní roli při vytváření softwaru, protože nejenže odhaluje chyby, ale také zajišťuje, že výsledný software splňuje požadavky a očekávání jeho budoucích uživatelů. Existuje celá řada různých testovacích metod, nicméně některé z nich nejsou pro tuto aplikaci relevantní. Ze struktury projektu je známo, že jednotlivá pravidla jsou implementována jako funkce. Pro takové uspořádání je vhodné jednotkové testování. *Unit testing* je fáze testování, kdy je testován nejmenší segment kódu, který lze testovat izolovaně od zbytku systému [79]. Vytvářené testy budou tedy kontrolovat návratové hodnoty funkcí, v nichž jsou implementována jednotlivá pravidla.

Programovací jazyk C++ nenabízí nativní knihovnu pro jednotkové testování. Vzhledem k velikosti projektu se zdálo zbytečné instalovat externí nástroj, který by jej umožnil, a proto bylo rozhodnuto pro vlastní řešení. Ze všeho nejdříve bylo nutné vytvořit *assert* funkci, která by vyhodnotila, zda je návratová hodnota funkce identická s hodnotou očekávanou. Pro připomenutí, každé definované pravidlo vrací datovou strukturu `vector<rule_error_s>`, tedy vektor chyb. Požadovaná *assert* funkce byla implementována jako `assert_rule` v souboru `tests/util.hpp`. V principu porovnává dva vektory chyb a informace vypisuje na standardní výstup. Pro každé pravidlo byla vytvořena série testů, aby se potvrdilo, že se jeho chování shoduje s očekáváními. Přirozeně se tedy nabídlo vytvořit vektor anonymních funkcí, přičemž každá funkce definuje konfiguraci, zdrojové kódy a očekávanou návratovou hodnotu a poté vyhodnotí výsledek (viz zdrojový kód 7.1).

Forma testování pomocí jednotkových testů a jejich implementace se ukázala být nanejvýš vhodná a flexibilní. V průběhu vytváření programu bylo vytvořeno **několik set testů**, které v mnoha případech pomohly odhalit nedostatky pravidel.

```
vector<function<void(void)>> test_rule_name({

    // test
    []() {

        // description
        string name = "...";

        // [key, value]
        unordered_map<string, string> config(/* ... */);

        // [file name, file lines]
        unordered_map<string, vector<string>> files(/* ... */);

        // expected errors
        vector<rule_error_s> expected_errors;

        // files to structure
        vector<file_s> structured_files = build_files_for_test(files);

        // observed errors
        vector<rule_error_s> observed_errors = rule_name(
            config,
            structured_files
        );

        // check results
        assert_rule(name, observed_errors, expected_errors);

    },

    // more tests

});

for(const auto &func : test_rule_name) {
    func();
}
```

Zdrojový kód 7.1: Ukázka jednotkového testu

7.1 Ověření v praxi

V první části testování bylo pomocí jednotkových testů ověřeno, že je chování pravidel v souladu s očekáváními. Nicméně bylo nezbytné ověřit aplikaci v reálném prostředí. Za tímto účelem byly vyexportovány všechny zdrojové kódy studentů v rámci kurzu Procedurální programování odevzdané v jednom semestru. Vzorek je složen přibližně ze 1700 souborů, ve kterých jsou implementovány různorodé úlohy, jejichž úplný seznam je v příloze C. Hodnocení bylo založeno na manuální revizi chyb, které program označil. Pokud byl jakýkoliv úsek kódu identifikován programem jako chybný, ale ve skutečnosti byl v souladu s přijatelným kódovacím stylem, tak byla tato chyba zařazena do kategorie *Špatně*. V případě správného vyhodnocení došlo k zařazení do kategorie *Správně*.

Pravidlo	**Celkem**	**Správně**	**Špatně**
Sekvence vnořených cyklů	1129	1129	0
Konvence pojmenování	862	862	0
ASCII znaky	363	363	0
Maximální délka řádku	3769	3769	0
Složené závorky funkcí	2145	2145	0
Složené závorky funkce main	464	464	0
Mezera za čárkou	2663	2663	0
Mezera za příkazem	6791	6791	0
Mezery okolo operátoru	4147	4147	0
Složené závorky přkazů	9376	9376	0
Odsazení kódu ve funkci	2470	2470	0
Magické konstanty	5122	5122	0
Porovnání ukazatele s nulou	8	8	0
Uppercase název makra	85	83	2
Hloubka kódu	1298	1298	0
Podobné bloky kódu	916	639	277
Počet parametrů funkce	194	194	0
Goto příkaz	55	55	0
Scanf funkce	352	276	76
Nekonstantní globální proměnné	424	424	0
Nepoužité proměnné	4	4	0
Otevření souboru	185	150	35
Uzavření souboru	34	34	0
Alokace paměti	839	679	160
Uvolnění paměti	764	764	0
Suma	44 459	43 909	550

Tabulka 7.1: Výsledky testování aplikace

V tabulce 7.1 jsou shrnuty výsledky pro jednotlivá pravidla. Celkově bylo vyznačeno téměř 45 000 chyb, z nichž byla drtivá většina označena správně. Tento výsledek je daný velmi přesnou identifikací nastalých situací v kódu. Pravidla, která se věnují souborům, paměti a načítání vstupu, se v některých případech vyhodnocovala nesprávně. Příčinou tohoto problému je složité sledování toku hodnot proměnných v kódu. Samotné uložení hodnoty do proměnné může z hlediska AST nastat ve čtyřech základních případech a název proměnné je často složen z několika tokenů. Studenti často neprovádí kontrolu načtených hodnot ve stejném bloku kódu, ale posílají je například jako parametry do jiných funkcí, což nemusí být nutně špatně. Implementace pravidla, které hledá podobné bloky kódu je založena na velikosti podstromu a na druhu uzlů. Tato implementace se ukázala být neefektivní. Vylepšit ji by bylo možné skrze hlubší kontrolu struktury stromu a tokenů, ze kterých se skládá. Pro hodnocení byla využita výchozí konfigurace, která stanoví minimální velikost podstromu na 20 uzlů. Zvýšení této hranice by mělo pozitivní vliv na výsledek.

Při manuální kontrole byly zjištěny situace, ve kterých program části kódů neoznačil jako chybné. Tento jev je důsledkem dvou faktorů. Z hlediska programu bylo v některých případech velmi složité definovat chybné stavy, a tedy implementace kladla důraz na to, aby vyznačená místa byla opravdu chybná. V druhém případě měla velký vliv knihovna Clang, jelikož její zpracování zdrojových kódů nebylo vždy přesné. Pokud při zpracování dat nastal problém, tak například neprovedla tokenizaci. Na platformě GitHub je v repozitáři *llvm-project* otevřeno přes 21 000 *issues*, z nichž se některá věnují těmto nedostatkům.

Závěr

Cílem práce bylo vytvořit aplikaci pro kontrolu stylu zápisu programu, která by byla schopna vyhodnocovat zdrojové kódy předané na vstupu v různých formátech. Motivací vzniku takového projektu byla potřeba rychlé a objektivní kontroly studentských zdrojových kódů v rámci vyučovaného předmětu Procedurální programování. V současné době jsou kódy hodnoceny manuálně. Hodnocení jsou zapsána do systému BRUTE v textové podobě nebo je vygenerováno PDF pomocí skriptu, který zpracovává speciálně formátované komentáře vložené do studentského kódu.

Nejprve bylo důležité vymezit a definovat odborné termíny pro následující kapitoly. Odborné názvosloví také pomohlo na počátku vymezit rozsah této práce. Například byl popsán pojem *kódovací styl*, byly rozebrány historické konvence pojmenování, styly odsazení nebo význam abstraktního syntaktického stromu. Formální názvy dílčí problematiky poskytly dostatečný základ pro nadcházející kapitolu, která se zabývá jednotlivými pravidly.

V rámci různých programovacích jazyků existuje široká škála způsobů, jak zapisovat zdrojové kódy, přičemž žádný z nich nelze považovat za chybný. Pro potřeby předmětu Procedurální programování byl již dříve definován seznam základních pravidel pro psaní čitelného kódu, jehož obsah se stal výchozím bodem pro literární část práce. Podrobným rozborem jednotlivých bodů seznamu bylo možné zjistit jejich význam pro kvalitu kódu. Zatímco některá pravidla jsou důsledkem historického vývoje oboru, tak některá mají svůj původ ve vnímání člověkem. Například obecné pravidlo maximálního počtu znaků na řádku má historický původ, neboť možnosti tehdejší techniky byly omezené. V dnešní době je jeho význam určen spíše fyziologií člověka, jelikož výzkumy naznačují, že pro přirozený pohyb oka je vhodné psát přibližně 90 znaků na řádku.

Před samotným návrhem a implementací aplikace proběhlo několik konzultací s vedoucí práce, jejichž cílem bylo stanovit technické požadavky. Výstupem se stala kapitola 4, která pojednává především o formátu vstupu, výstupu a kompatibilitě s operačními systémy. Mezi hlavní požadavky patří také konfigurovatelnost programu bez nutnosti zásahu do zdrojového kódu.

Na základě technických požadavků byl vytvořen konceptuální návrh, který se v obecné rovině zabývá myšlenkou a fungováním projektu. Byl představen návrh průběhu aplikace a přibližná forma konfigurace. Také byla představena struktura implementace pravidel, která se opírá o abstraktní syntaktický strom a tokenizaci zdrojového kódu. Princip fungování obsažený v návrhu byl jako takový dodržen, avšak v průběhu implementace došlo především z technických důvodů k částečným změnám.

Úvodní část implementace, která trvala několik měsíců, byla z velké části experimentální, jelikož bylo složité nalézt vhodné technologie, které by alespoň částečně řešily danou problematiku. Programovací jazyk Python a čistě v něm implementované knihovny neposkytly dostatečnou rychlost ani požadovanou funkcionalitu. Ve výsledku se ukázala být kombinace jazyka C++ a knihovny Clang více než vhodná. Možnosti organizace paměti, způsobu reprezentace dat a optimalizace při kompilaci měly pozitivní dopad na dobu běhu programu. Nicméně toto řešení mělo svá úskalí při práci se soubory, souborovým systémem a dynamicky alokovanou pamětí. Tyto problémy se podařilo vyřešit implementací pomocných funkcí, a tedy při vytváření jednotlivých pravidel nedošlo k vážným komplikacím. V příloze 2 lze nalézt kategorizovaný výčet kontrolovaných pravidel. Výstup programu byl rozdělen do tří forem. První je určena pro studenta a obsahuje podrobné popisy pravidel a všechny nalezené chyby. Druhá forma je určena pro vyučujícího, a tedy výpisy jsou více technického rázu. Kromě chyb, které jsou i ve výstupu studenta, se zde vypisují i některé statistiky, jež by měly usnadnit hodnocení zdrojového kódu. Poslední forma výstupu byla definována později, a proto není uvedena v návrhu programu. Výstupem je studentský zdrojový kód doplněný o speciální komentáře, které lze zpracovávat již existujícím externím skriptem.

Poslední kapitola se věnovala testování výsledné aplikace. Pro ověření bylo vytvořeno několik set jednotkových testů, které kontrolovaly chování jednotlivých pravidel. Jejich vznik umožnil odhalit a opravit mnoho nedostatků. Vzhledem k dobrému testování programu byly výsledky při následném ověření v praxi dobré. Na anonymizovaném datasetu studentských zdrojových kódu byla provedena automatická kontrola stylu zápisu programu s manuální revizí. Chování aplikace se ukázalo být konzistentní a výstupy, které generovala, z velké části odpovídaly očekávání. Ze 44 459 chyb označených programem jich bylo označeno 98,8% správně.

Celkově lze konstatovat, že cílů vymezených na začátku práce bylo uspokojivě dosaženo, což také dokazuje kapitola o testování a fakt, že již v průběhu implementace byla aplikace několikrát použita v reálném prostředí. Návrh a struktura projektu nabízí určitou modularitu při použití a vytváření pravidel. Do budoucna je tedy možné aplikaci jednoduše rozšířit o nová pravidla a jazykové mutace výstupu.

Bibliografie

1. OGURA, Naoto; MATSUMOTO, Shinsuke; HATA, Hideaki; KUSUMOTO, Shinji. Bring your own coding style. In: 2018, s. 527–531. Dostupné z DOI: 10.1109/SANER.2018.8330253.
2. YASIR, Rafed M.; KABIR, Dr A. Exploring the Impact of Code Style in Identifying Good Programmers. 2022.
3. MI, Qing; KEUNG, Jacky; YU, Yang. Measuring the Stylistic Inconsistency in Software Projects using Hierarchical Agglomerative Clustering. In: 2016, s. 1–10. Dostupné z DOI: 10.1145/2972958.2972963.
4. ZOU, Weiqin; XUAN, Jifeng; XIE, Xiaoyuan; CHEN, Zhenyu; XU, Baowen. How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects. *Empirical Software Engineering*. 2019, roč. 24. Dostupné z DOI: 10.1007/s10664-019-09720-x.
5. PERUMA, Anthony; NEWMAN, Christian D. Understanding Digits in Identifier Names: An Exploratory Study. In: Association for Computing Machinery, 2023. ISBN 9781450393430. Dostupné z DOI: 10.1145/3528588.3528657.
6. ECK, D.J. *Introduction to Programming Using Java*. 2019. Dostupné také z: https://books.google.cz/books?id=hMtMzQEACAAJ.
7. BINKLEY, David; DAVIS, Marcia; LAWRIE, Dawn; MORRELL, Christopher. To camelcase or under-score. In: 2009, s. 158–167. Dostupné z DOI: 10.1109/ICPC.2009.5090039.
8. AMOS, D.; BADER, D.; JABLONSKI, J.; HEISLER, F. *Python Basics: A Practical Introduction to Python 3*. Real Python, 2021. ISBN 9781775093329.
9. GOOGLE. *Google C++ Style Guide* [https://google.github.io/styleguide/cppguide.html]. 2023. Citováno dne 18.7.2023.
10. GOOGLE. *Google Java Style Guide* [https://google.github.io/styleguide/javaguide.html]. 2023. Citováno dne 18.7.2023.
11. SAMBASIVAM, Hariharan. *Naming variables using Hungarian Notation: Computimes, , 2 Edition*. 1997.
12. HORTON, Ivor. *Windows Programming Concepts*. United States: John Wiley & Sons, Incorporated, 2012. Ivor Horton's Beginning Visual C++ 2012. ISBN 1118368088; 9781118368084.
13. STANISLAV KRAČMAR, Jiří Vogel. *Programovací jazyk C - Doplňkové skriptum*. Dostupné také z: http://vyuka.janoud.cz/Programovaci_jazyk_C.pdf. Citováno dne 23.7.2023.
14. WAGNER, Tim; GRAHAM, S.L. Modeling User-Provided Whitespace and Comments in an Incremental SDE. 2000.

15. HINDLE, Abram; GODFREY, Michael; HOLT, Richard. From Indentation Shapes to Code Structures. In: 2008, s. 111–120. Dostupné z DOI: 10.1109/SCAM.2008.31.
16. OMAN, Paul W.; COOK, Curtis R. Typographic Style is More than Cosmetic. *Commun. ACM*. 1990, roč. 33, č. 5, s. 506–520. ISSN 0001-0782. Dostupné z DOI: 10.1145/78607.78611.
17. HINDLE, A.; GODFREY, M. W.; HOLT, R. C. From Indentation Shapes to Code Structures. In: IEEE, 2008, s. 111–120. ISBN 9780769533537; 0769533531;
18. PARVATHAM, Niranjan K. *Impact of Indentation in Programming*. 2013. Tech. zpr. Cornell University Library, arXiv.org.
19. MCCONNELL, Steve. *Code complete*. 2nd. Redmond: Microsoft Press, 2004. ISBN 0735619670; 9780735619678;
20. KERNIGHAN, Brian W.; RITCHIE, Dennis M. *The C Programming Language*. 2nd. Prentice Hall Professional Technical Reference, 1988. ISBN 0131103709.
21. STROUSTRUP, Bjarne. *(The) C++ programming language*. 1. vyd. Reading: Addison-Wesley Publishing Company, 1986. ISBN 0201539926; 9780201539929;
22. POOLE, H.W.; LAMBERT, L.; WOODFORD, C.; MOSCHOVITIS, C.J.P. *The Internet: A Historical Encyclopedia*. ABC-CLIO, 2005. The Internet: A Historical Encyclopedia, č. sv. 2. ISBN 9781851096596.
23. THE LINUX KERNEL DEVELOPMENT COMMUNITY. *Linux Kernel Coding Style* [https://www.kernel.org/doc/html/latest/process/coding-style.html]. [B.r.]. Citováno dne 26.7.2023.
24. ĎURAČÍK, Michal; HRKÚT, Patrik; KRSAK, Emil; TOTH, Štefan. Abstract Syntax Tree Based Source Code Antiplagiarism System for Large Projects Set. *IEEE Access*. 2020, roč. 8, s. 175347–175359. Dostupné z DOI: 10.1109/ACCESS.2020.3026422.
25. NG, Resmi. Abstract Syntax Tree Generation using Modified Grammar for Source Code Plagiarism Detection. 2014.
26. KIM, Jaehyun; LEE, Yangsun. A Study on Abstract Syntax Tree for Development of a JavaScript Compiler. *International Journal of Grid and Distributed Computing*. 2018, roč. 11, s. 37–48. Dostupné z DOI: 10.14257/ijgdc.2018.11.6.04.
27. WENHAN, Wang; LI, G; JIN, Zhi; SHEN, Sijie; XIA, Xin. *Modular Tree Network for Source Code Representation Learning*. 2020. Dostupné z DOI: 10.13140/RG.2.2.19077.99040.
28. FANG, Xuefen. Using a coding standard to improve program quality. In: *Proceedings Second Asia-Pacific Conference on Quality Software*. 2001, s. 73–78. Dostupné z DOI: 10.1109/APAQS.2001.990004.
29. BOEHM, Barry W.; BASILI, Victor R. Software Defect Reduction Top 10 List. *Computer*. 2001, roč. 34, č. 1, s. 135–137. Dostupné z DOI: 10.1109/2.962984.
30. JONES, Derek M. Operand Names Influence Operator Precedence Decisions (Part 1 of 2). 2007. Experiment performed at the 2007 ACCU Conference.

31. SUTTER, Herb; ALEXANDRESCU, Andrei. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN 0321113586.
32. BAILEY, Tim. *An Introduction to the C Programming Language and Software Design* [eBook]. The University of Sydney, [b.r.]. ISBN N/A. ISSN N/A. PDF (153 pages).
33. MACKENZIE, Charles E. *Coded-Character Sets: History and Development*. USA: Addison-Wesley Longman Publishing Co., Inc., 1980. ISBN 0201144603.
34. GUSTEDT, J. *Modern C*. Manning, 2019. ISBN 9781617295812.
35. HEIDE, Lars. *Punched-Card Systems and the Early Information Explosion, 1880-1945*. Baltimore: Johns Hopkins University Press, 2009. ISBN 0801891434; 9780801891434; 0801898722; 9780801898723; 9781421427874; 1421427877;
36. MASSACHUSETTS, University of. *IBM Punch Card*. Dostupné také z: `https://www.umass.edu/molvis/workshop/imgs/ibm_card.htm`. 29.10.2023.
37. BOVET, Daniel; CESATI, Marco. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005. ISBN 0596005652.
38. ITKONEN, Mark; GROTESKI, antiikva. Markus Itkonen Typography and readability. In: 2007.
39. KOSTIĆ, Marija; BATANOVIĆ, Vuk; NIKOLIC, Bosko. Monolingual, multilingual and cross-lingual code comment classification. *Engineering Applications of Artificial Intelligence*. 2023, roč. 124, s. 106485. Dostupné z DOI: `10.1016/j.engappai.2023.106485`.
40. JABRAYILZADE, Elgun; YURTOĞLU, Ayda; TÜZÜN, Eray. Taxonomy of inline code comment smells. *Empirical Software Engineering*. 2024, roč. 29. Dostupné z DOI: `10.1007/s10664-023-10425-5`.
41. WOODFIELD, S. N.; DUNSMORE, H. E.; SHEN, V. Y. The effect of modularization and comments on program comprehension. In: *Proceedings of the 5th International Conference on Software Engineering*. San Diego, California, USA: IEEE Press, 1981, s. 215–223. ICSE ’81. ISBN 0897911466.
42. HARTZMAN, Carl S.; AUSTIN, Charles F. Maintenance productivity: observations based on an experience in a large system environment. In: *Conference of the Centre for Advanced Studies on Collaborative Research*. 1993.
43. COLLAR, Emilio; VALERDI, Ricardo. Role of Software Readability on Software Development Cost. 2014.
44. FOWLER, Martin. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999. ISBN 0-201-48567-2.
45. MARTIN, Robert C. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. vyd. USA: Prentice Hall PTR, 2008. ISBN 0132350882.
46. ANQUETIL, Nicolas; DELPLANQUE, Julien; DUCASSE, Stéphane; ZAITSEV, Oleksandr; FUHRMAN, Christopher; GUÉHÉNEUC, Yann-Gaël. What do developers consider magic literals? A smalltalk perspective. *Information and Software Technology*. 2022, roč. 149, s. 106942. Dostupné z DOI: `10.1016/j.infsof.2022.106942`.

47. HAMED, Tarfa. *Pointers in C++*. 2021. Dostupné z DOI: 10.13140/RG.2.2.31168.25604.

48. CRAIG, Michelle; PETERSEN, Andrew. Student difficulties with pointer concepts in C. In: 2016, s. 1–10. Dostupné z DOI: 10.1145/2843043.2843348.

49. KERNIGHAN, Brian W.; PLAUGER, P. J. *The Elements of Programming Style*. 2nd. USA: McGraw-Hill, Inc., 1982. ISBN 0070342075.

50. WEINBERG, Gerald M. *Understanding the Professional Programmer*. Dorset House Publishing Co., Inc., 2000. ISBN 0932633099.

51. MORZECK, Johannes; HANENBERG, Stefan; WERGER, Ole; GRUHN, Volker. Indentation in Source Code: A Randomized Control Trial on the Readability of Control Flows in Java Code with Large Effects. In: *Proceedings of the 18th International Conference on Software Technologies - Volume 1: ICSOFT*. SciTePress, INSTICC, 2023, s. 117–128. ISBN 978-989-758-665-1. Dostupné z DOI: 10.5220/0012087500003538.

52. PARVATHAM, Niranjan. Impact of Indentation in Programming. *International Journal of Programming Languages and Applications*. 2013, roč. 3. Dostupné z DOI: 10.5121/ijpla.2013.3403.

53. AJAMI, Shulamyt; WOODBRIDGE, Yonatan; FEITELSON, Dror G. Syntax, Predicates, Idioms - What Really Affects Code Complexity? In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 2017, s. 66–76.

54. HOTTA, Keisuke; SASAKI, Yui; SANO, Yukiko; HIGO, Yoshiki; KUSUMOTO, Shinji. An Empirical Study on the Impact of Duplicate Code. *Advances in Software Engineering*. 2012, roč. 2012. Dostupné z DOI: 10.1155/2012/938296.

55. KIM, Miryung; SAZAWAL, Vibha; NOTKIN, David; MURPHY, Gail. An empirical study of code clone genealogies. *SIGSOFT Softw. Eng. Notes*. 2005, roč. 30, s. 187–196. Dostupné z DOI: 10.1145/1095430.1081737.

56. KAPSER, Cory; GODFREY, Michael. Cloning Considered Harmful Considered Harmful. In: 2006, s. 19–28. Dostupné z DOI: 10.1109/WCRE.2006.1.

57. SEDANO, Todd. Code Readability Testing, an Empirical Study. In: 2016. Dostupné z DOI: 10.1109/CSEET.2016.36.

58. DANTAS, Carlos; MAIA, Marcelo. Readability and Understandability Scores for Snippet Assessment: an Exploratory Study. In: 2021, s. 46–50. Dostupné z DOI: 10.5753/vem.2021.17217.

59. EBAD, Shouki. Investigating the Input Validation Vulnerabilities in C Programs. *International Journal of Advanced Computer Science and Applications*. 2023, roč. 14. Dostupné z DOI: 10.14569/IJACSA.2023.0140117.

60. BRAZ, Larissa; FREGNAN, Enrico; CALIKLI, Gul; BACCHELLI, Alberto. Why Don't Developers Detect Improper Input Validation? '; DROP TABLE Papers; –. In: 2021, s. 499–511. Dostupné z DOI: 10.1109/ICSE43902.2021.00054.

61. M A, Rama. *Problem Solving Through C Programming - Chapter 3*. 2013. ISBN 978-93-83214-09-9.

62. EKMEKCI, Berk; MCANANY, Charles; MURA, Cameron. An Introduction to Programming for Bioscientists: A Python-Based Primer. *PLOS Computational Biology*. 2016, roč. 12. Dostupné z DOI: 10.1371/journal.pcbi.1004867.

63. WEISSMAN, Larry. Psychological complexity of computer programs: an experimental methodology. *SIGPLAN Not.* 1974, roč. 9, č. 6, s. 25–36. ISSN 0362-1340. Dostupné z DOI: 10.1145/953233.953237.

64. NAKAMURA, Masahide; MONDEN, Akito; ITOH, Tomoaki; MATSUMOTO, Ken-ichi; KANZAKI, Yuichiro; SATOH, Hirotsugu. Queue-Based Cost Evaluation of Mental Simulation Process in Program Comprehension. In: *Proceedings of the 9th International Symposium on Software Metrics*. USA: IEEE Computer Society, 2003, s. 351. METRICS '03. ISBN 0769519873.

65. SASAKI, Yui; HIGO, Yoshiki; KUSUMOTO, Shinji. Reordering Program Statements for Improving Readability. In: 2013, s. 361–364. ISBN 978-1-4673-5833-0. Dostupné z DOI: 10.1109/CSMR.2013.50.

66. IRABASHETTI, Prabhudev; PATIL, Nilesh. Dynamic Memory Allocation: Role in Memory Management. *International Journal of Current Engineering and Technology*. 2014, roč. Vol.4, s. 531–535.

67. M.A, Jayaram. *C Programming*. 2011. ISBN 978-81-280-1454-3.

68. DEVKOTA, Pratik. *Dynamic Memory Allocation: Implementation and Misuse*. 2023. Dostupné z DOI: 10.13140/RG.2.2.34993.97129.

69. KELECHAVA, Brad. *The Origin of ANSI C and ISO C*. 2017-09. Dostupné také z: https://blog.ansi.org/2017/09/origin-ansi-c-iso-c/. Accessed: 2024-01-20.

70. ISO. *ISO C Standard 1999*. 1999. Tech. zpr. Dostupné také z: http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf. ISO/IEC 9899:1999 draft.

71. BIBAEV, Vitaliy; KALINA, Alexey; LOMSHAKOV, Vadim; GOLUBEV, Yaroslav; BEZZUBOV, Alexander; POVAROV, Nikita; BRYKSIN, Timofey. All you need is logs: improving code completion by learning from anonymous IDE usage logs. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2022, s. 1269–1279. ESEC/FSE 2022. ISBN 9781450394130. Dostupné z DOI: 10.1145/3540250.3558968.

72. JOHNSON, Bruce. Introducing Visual Studio Code. In: 2019, s. 1–12. ISBN 9781119588184. Dostupné z DOI: 10.1002/9781119588238.ch1.

73. BAHRAMBEIGI, Yousef. *LEARNING PYTHON*. 2024. ISBN 9798880138234.

74. MOGENSEN, Torben. Lexical Analysis. In: 2024, s. 1–39. ISBN 978-3-031-46459-1. Dostupné z DOI: 10.1007/978-3-031-46460-7_1.

75. WILLIE, Ebipamobonumugha. PROGRAMMING IN C++. 2024.

76. PERKINS, Benjamin; HAMMER, Jacob; REID, Jon. XML and JSON. In: 2018, s. 673–696. ISBN 9781119458685. Dostupné z DOI: 10.1002/9781119549550.ch21.

77. ZHANG, Chen; XIE, Qingyuan; WANG, Mingyue; GUO, Yu; JIA, Xiaohua. Optimal Compression for Encrypted Key-Value Store in Cloud Systems. *IEEE Transactions on Computers*. 2024, roč. PP, s. 1–14. Dostupné z DOI: 10.1109/TC.2024.3349653.

78. MUDHOLKAR, Megha; MUDHOLKAR, Pankaj. *A Beginner's Guide to Git and GitHub*. 2017. Dostupné z DOI: 10.13140/RG.2.2.20126.89927.

79. FONTES, Afonso; GAY, Gregory; NETO, Francisco; FELDT, Robert. Automated Support for Unit Test Generation. In: 2023, s. 179–219. ISBN 978-981-19-9947-5. Dostupné z DOI: 10.1007/978-981-19-9948-2_7.

Přílohy

A Implementované funkce

Kategorie	Jméno funkce v kódu	Název
Konzistentní názvy	rule_for_sequence	Sekvence vnořených cyklů
	rule_naming_convention	Konvence pojmenování
Formátování	rule_ascii_character	ASCII znaky
	rule_line_length	Maximální délka řádku
	rule_function_parentheses	Složené závorky funkcí
	rule_main_parentheses	Složené závorky funkce main
	rule_space_after_comma	Mezera za čárkou
	rule_space_after_statement	Mezera za příkazem
	rule_space_around_operator	Mezery okolo operátoru
	rule_statement_parentheses	Složené závorky příkazů
	rule_function_indent	Odsazení kódu ve funkci
Komentáře	print_stats	Výpis statistik
Makra	rule_magic_constant	Magické konstanty
	rule_pointer_comparison_to_zero	Porovnání ukazatele s nulou
	rule_macro_uppercase	Uppercase název makra
Struktura kódu	rule_code_depth	Hloubka kódu
	rule_duplicates	Podobné bloky kódu
	rule_function_parameters	Počet parametrů funkce
	rule_goto	Goto příkaz
	rule_xscanf	Scanf funkce
Lokálnost	rule_global_variable	Nekonstantní globální proměnné
	rule_var_not_used	Nepoužité proměnné
Práce s pamětí	rule_file_open	Otevření souboru
	rule_file_close	Uzavření souboru
	rule_memory_allocation	Alokace paměti
	rule_memory_deallocation	Uvolnění paměti

Tabulka 2: Implementované funkce dle základní pravidel

B Instalace a spuštění

B.1 Závislosti

Při kompilaci programu je nezbytné mít k dispozici **Clang development package**. Jeho instalační proces se liší v závislosti na operačním systému. Pro linuxové distribuce používající **APT** nástroj pro manipulaci s balíčkovacím systémem z příkazové řádky je instalační proces následující:

```
sudo apt-get update
sudo apt-get install libclang-dev
```

Důležité je zapamatovat si místo uložení knihovny na disku. Pro linuxové operační systémy bude cesta zpravidla vypadat přibližně takto:

```
/usr/lib/llvm-[VERZE]/include/clang-c
```

Aplikace z této knihovny používá jen několik funkcí, a proto bude pravděpodobně fungovat na téměř jakékoliv verzi knihovny. Správná funkčnost byla ověřena na těchto verzích:

- 14
- 15
- 16

B.2 Kompilace

Program je napsaný v programovacím jazyce **C++17**. Je tedy nezbytné mít nainstalovaný daný kompiler. Pro zjednodušení procesu kompilace je použit program **make**. Před jeho zavoláním je nezbytné aktualizovat kompiler a cestu k Clang v následujících `Makefile` souborech:

```
makefile
tests/makefile
```

Při zavolání příkazu `make` v kořenovém adresáři projektu se vytvoří spustitelná aplikace s názvem `application`.

B.3 Spuštění

Obecný příkaz pro spuštění bude mít tuto strukturu:

```
./application [JAZYK] [CESTA/KE/KONFIGURACI] [-dir|-files] [CESTY]
```

- **jazyk** - aktuálně podporovaný jazyk je `cs`, tedy čeština
- **konfigurace** - výchozí konfigurační soubor je v kořenovém adresáři projektu s názvem `config.conf`

- **přepínač** - pokud je zvolen přepínač `-dir`, tak následující cesta bude cesta ke složce rozbaleného `TGZ` souboru ze systému BRUTE; pokud je zvolen přepínač `-files`, tak následující cesty budou ke konkrétním souborům se zdrojovými kódy

Podrobněji se spuštění programu věnuje sekce 6.8, která rozebírá jednotlivé položky více do detailu. Pro ukázku mohou příkazy pro spuštění vypadat následovně:

```
./application cs config.conf -dir /Downloads/HW04
./application cs ./config.conf -dir ./data/HW01
./application cs config.conf -files ./test/test.c
./application cs ./config.conf -files ./test/test.c ./test/main.c
```

B.4 Výstup

Jelikož se program spouští z příkazové řádky, tak se v adresáři, ze kterého je zavolán, vytvoří složka s názvem `output`, do níž jsou vygenerovány výstupy. Aplikace tedy **musí mít potřebná oprávnění** pro vytváření, přepisování a mazání souborů.

B.5 Testování

Při zavolání příkazu `make` v adresáři `tests` se vytvoří spustitelná aplikace s názvem `testapp`. Při jejím spuštění se vypíše výsledek na standardní výstup.

C Úlohy použité při testování

- **PRP Úloha 1** - Načítání vstupu, výpočet a výstup
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw01`
- **PRP Úloha 2** - První cyklus
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw02`
- **PRP Úloha 3** - Kreslení
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw03`
- **PRP Úloha 4** - Prvočíselný rozklad
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw04`
- **PRP Úloha 5** - Caesarova šifra
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw05`
- **PRP Úloha 6** - Maticové počty
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw06`
- **PRP Úloha 7** - Hledání textu v souborech
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw07`
- **PRP Úloha 8** - Kruhová fronta v poli
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw08`
- **PRP Úloha 9** - Načítání a ukládání grafu
  `https://cw.fel.cvut.cz/b231/courses/b0b36prp/hw/hw09`

D Externí přílohy

K této diplomové práci je přiložen soubor `code.zip`, který obsahuje zdrojové kódy implementované aplikace stažené z platformy GitHub. Repozitář je soukromý a nelze na něj odkázat.