Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Automatická statistická a metrická analýza zdrojového kódu se zameřením na jazyk ANSI C

Plzeň, 2012 Jiří Fatka

Místo pro zadání

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne..

Abstract

Automatic Statistic and Metric Analysis of Source Code with Focus on the ANSI C Language

Thesis is focused on tools, created for metric analysis of source code that is written in programming language ANSI C. The results of this analysis are comparisons of several student essays' source codes, which are later used for proofing plagiarism.

The tool for metric analysis is designed to be a regular compiler; therefore it contains most parts of regular ANSI C compiler.

For storing of source code footprints, new data format was created. It is used by the comparing tool for faster source code comparison with other essays' source codes. A special application is designed for work with this data format.

The tools use a shared library that is designed for source code analysis and it can be extended, with extra modifications, to support other programming languages.

Obsah

1 Úvod **7**

2 Metody porovnávání **9**
2.1 Metody založené na textu . . . 9
2.2 Metody založené na tokenech . . . 9
2.3 Metody založené na stromech . . . 9
2.4 Metody založení na PDG . . . 10
2.5 Metody založené na metrikách . . . 10
2.6 Hybridní metody . . . 10

3 Jazyk ANSI C **11**
3.1 Fáze překladu . . . 11
3.2 Preprocesor . . . 12
3.2.1 Vkládání souborů . . . 12
3.2.2 Podmíněný překlad . . . 13
3.2.3 Makra . . . 13
3.3 Syntaxe . . . 14
3.3.1 Datové typy . . . 14
3.3.2 Proměnné . . . 17
3.3.3 Funkce . . . 17
3.3.4 Konstrukce . . . 18
3.4 Knihovny . . . 20
3.5 Verze jazyka C . . . 21
3.5.1 K&R C . . . 21
3.5.2 ANSI C . . . 21
3.5.3 C90 . . . 21
3.5.4 C94 . . . 21
3.5.5 C99 . . . 21
3.5.6 C11 . . . 22

4 Metrická analýza **23**
4.1 Funkce . . . 23
4.1.1 Tělo funkce . . . 24
4.2 Proměnné . . . 25
4.3 Datové typy . . . 25
4.4 Struktury . . . 25
4.5 Výčtové typy . . . 26

5 Technologie **27**
5.1 Požadavky . . . 27
5.2 Programovací jazyk . . . 27
5.3 ANSI C parser . . . 28
5.4 ANSI C preprocesor . . . 28
5.5 Uložení otisku zdrojového kódu . . . 29

6 Realizace **31**
6.1 Rozdělení projektu . . . 31
6.1.1 Analyzátor kódu . . . 31
6.1.2 Překladač . . . 31
6.1.3 Čtečka otisků zdrojových kódů . . . 31
6.1.4 Porovnávání binárních otisků . . . 31
6.2 Zpracování zdrojového kódu . . . 32
6.2.1 BasicLoader . . . 32
6.2.2 Token . . . 32
6.2.3 BasicTokenizer . . . 33
6.2.4 BasicParser . . . 33
6.2.5 BasicExpression . . . 33
6.3 Popisná struktura otisku . . . 34
6.3.1 Project . . . 34
6.3.2 Unit . . . 34
6.3.3 Type . . . 35
6.3.4 Enum . . . 35
6.3.5 Structure . . . 35
6.3.6 Member . . . 36
6.3.7 Variable . . . 36
6.3.8 Function . . . 36
6.3.9 Statement . . . 37
6.3.10 Declaration . . . 37
6.3.11 Expression . . . 38
6.4 ANSI C . . . 38
6.4.1 Loader . . . 38
6.4.2 Tokenizer . . . 38
6.4.3 Preprocesor . . . 39
6.4.4 Parser . . . 42
6.5 INI . . . 43
6.5.1 File . . . 44
6.5.2 Section . . . 44
6.6 Ukládání a načítání otisku . . . 44
6.7 Porovnání otisků . . . 45
6.7.1 Výpočet shodnosti . . . 45
6.7.2 Porovnání množin . . . 45
6.7.3 Porovnávané metriky . . . 46
6.7.4 Příklady . . . 48
6.8 Možnosti vylepšení . . . 52
6.8.1 Porovnávání . . . 52
6.8.2 Podpora pro další programovací jazyky . . . 52

6.8.3 Popisná struktura otisku . . . 52
6.8.4 Přepracování analytické části . . . 52
6.8.5 Interpret . . . 53

7 Závěr **54**

Rejstřík **55**

Seznam zkratek **56**

A Zdrojové kódy **59**

B Tabulky **61**

C Diagramy **62**

D Reakce na syntaktické chyby **65**

E Formát souboru otisku **67**

E.1 Formáty částí . . . 68
E.1.1 Project . . . 68
E.1.2 Unit . . . 68
E.1.3 Variable . . . 68
E.1.4 Type . . . 69
E.1.5 Structure . . . 69
E.1.6 Member . . . 69
E.1.7 Enum . . . 69
E.1.8 Function . . . 70
E.1.9 Declaration . . . 70
E.1.10 Statement . . . 71
E.1.11 Expression . . . 73

F Uživatelská příručka **74**

F.1 Požadavky na aplikace . . . 74
F.2 Překlad . . . 74
F.2.1 Požadavky . . . 74
F.2.2 Linux . . . 74
F.2.3 Windows . . . 74
F.2.4 Binární soubory . . . 74
F.3 Program `cc` (překladač) . . . 75
F.4 Program `objdump` . . . 75
F.5 Program `comparator` . . . 76

Kapitola 1

Úvod

Opisování prací je v dnešní době Internetu velice snadné a není problém s minimálním úsilím vydávat cizí práci za vlastní. Je to celosvětový problém v rámci celého školství a manuální hledání shodných prací je při velkém množství odevzdaných samostatných prací časově náročné a někdy i nemožné. Stále častěji se tak objevují nástroje, které odevzdávané práce analyzují a porovnávají je s ostatními pracemi ve snaze nalézt shodné části a případně určit, zda je autor skutečným autorem odevzdané práce.

Tato práce se zabývá vytvořením nástrojů pro metrickou analýzu zdrojových kódů napsaných v jazyce *ANSI C* a hledání shodných nebo alespoň částečně podobných samostatných prací. Cílem je upozornit na podobné samostatné práce a snížit tak potřebu úplné ruční kontroly odevzdaných samostatných prací. Výstupem nástroje by měla být statistika se seřazeným seznamem prací, se kterými je odevzdaná práce podobná.

Pro účely porovnávání bude využito metrik získaných pomocí metrické analýzy zdrojových kódů. Tyto metriky by se měly uložit do vhodně zvoleného typu úložiště, aby nebylo potřeba při každém porovnávání provádět znovu analýzu všech odevzdaných prací.

Kapitola 2 popisuje různé metody porovnávání zdrojových kódů, které se v současné době ve světě používají. Zbytek práce se zaměřuje jen na metodu založenou na metrikách.

V kapitole 3 je v jednoduchosti popsán jazyk *ANSI C*, pro který práce vzniká. Není cílem popsat všechny vlastnosti a schopnosti jazyka, ale jen čtenáři přiblížit samotný programovací jazyk, jak vypadá, čeho je schopen apod.

Kapitola 4 se zabývá metrikami, které jsou vhodné pro porovnávání zdrojových kódů. Jsou zde popsány i ostatní metriky, které lze získat ze zdrojového kódu, ale jejich uložení nebo porovnávání by bylo pro tuto práci složité.

Kapitola 5 popisuje nejvhodnější volby pro implementaci nástrojů. Důvody proč byly zvoleny tyto technologie a ne jiné.

V další kapitole (kapitola 6) je popsána realizace knihovny pro analýzu zdrojových kódů v jazyce *ANSI C*, popis datové struktury pro uložení získaných metrik a taktéž způsob porovnávání získaných metrik. Na konci kapitoly jsou zmíněny možná vylepšení, která by výsledky porovnávání zdrojových kódu zpřesnily.

Závěr (kapitola 7) shrnuje výsledky této diplomové práce.

Jak vytvořený nástroj reaguje na syntaktické chyby ve zdrojových kódech lze nalézt v příloze D.

V příloze E je detailní popis datového formátu otisku, který všechny vytvořené nástroje využívají. Tento popis lze využít k vytvoření dalších nástrojů pracujících s těmito otisky.

Příloha F obsahuje uživatelský manuál, kde je popsán postup překladu vytvořených nástrojů a popis jejich použití.

Kapitola 2

Metody porovnávání

Pro porovnávání zdrojových kódů existuje několik metod, které přistupují k porovnávání rozdílným způsobem. Popis jednotlivých metod byl převzat z *A Survey of Software Clone Detection Research* [3]. Většina z uvedených technik je určena převážně na hledání shodných částí kódu v rámci jednoho programu se zaměřením na optimalizaci kódu programu.

2.1 Metody založené na textu

Tyto metody jsou založeny čistě na práci s textem, kdy se zdrojový kód rozdělí na řádky nebo řetězce, které jsou následně porovnávány. Ve většině případů se zdrojový kód neupravuje a je použit přímo, jen někdy se využívají postupy:

- odstranění komentářů,
- odstranění bílých znaků (tabulátory, odřádkování a mezery),
- normalizace kódu.

Některé implementace si upravují kód dále a to např. nahrazováním názvů proměnných speciálními hodnotami.

2.2 Metody založené na tokenech

V těchto metodách se používá postup, kdy se celý zdrojový kód převede na sekvenci tokenů. V této sekvenci jsou vyhledány shodné pod-sekvence tokenů. Na rozdíl od metod založených na textu, jsou tyto metody mnohem méně náchylnější na změny ve zdrojovém kódu, především na formátování.

Stejně jako u metod založených na textu se upravuje sekvence tokenů podle určitých pravidel a to hlavně podle zpracovávaného programovacího jazyka.

2.3 Metody založené na stromech

Zdrojový kód zpracovaný těmito metodami je převeden pomocí parseru daného programovacího jazyka na strom nebo abstraktní syntaktický strom (AST). V takto

vytvořeném stromu se následně vyhledají podobné podstromy. Obdobně jako u předchozích metod se upravuje strom zahozením názvů proměnných a literálů, aby neměly vliv na porovnávání.

2.4 Metody založení na PDG

Metody založené na *PDG* (Program Dependency Graph) jsou vyspělejší než předchozí metody, jelikož uchovávají i sémantickou informaci zdrojového kódu. Jsou schopny zachytit průběh výpočtu programu i průchod dat v programu. Obdobně jako u metod založených na stromech vzniká strom, který je následně použit pro vyhledání podstromů.

Výhodou těchto metod je odolnost vůči přehazování konstrukcí, vkládání a odebírání kódu.

2.5 Metody založené na metrikách

Tyto metody sbírají různé metriky částí zdrojového kódu a porovnávají získané vektory metrik místo porovnávání samotného kódu. Metriky, označované jako „fingerprinting functions“, jsou vypočítány pro jednu nebo více syntaktických jednotek (třída, funkce a dokonce i konstrukce) a jsou použity pro vyhledání shod mezi těmito syntaktickými jednotkami. Ve většině případů je zdrojový kód zpracován do *AST*/*PDG* (viz předchozí metody).

Mayrand a spol. vypočítávají některé metriky (počet řádek, počet volání funkcí, atd.) pro každou funkci programu. Funkce s podobnými metrikami jsou označeny jako kopie. Metriky se vypočítávají z názvů, uspořádání, výrazů a jednoduchého chování funkcí. Pro reprezentaci kódu funkce používá tzv. *Intermediate Representation Language* (*IRL*). Tato metoda je schopna detekovat jen kopie celých funkcí a nerozezná částečné kopie.

Kontogiannis a spol. vytvořili abstraktní detekční nástroj na základě předpokládané shody pomocí Markovových modelů. Tento přístup nehledá shodný kód, ale jen měří podobnost dvou programů.

Porovnávání těmito metodami je velice rychlé, ale může vést k nepřesnému označení shodných programů na základě schodných metrik rozdílných kusů kódu.

2.6 Hybridní metody

Některé další metody používají jiný princip, který je většinou založený na kombinaci předchozích metod.

Kapitola 3

Jazyk ANSI C

Programovací jazyk *ANSI C* je standardizovaná verze jazyka *C*, která byla přijata *American National Standards Institute* (ANSI) v roce 1989 (někdy označován jako *C89*). Téměř totožná verze byla standardizována institutem *ISO* v roce 1990 (znám jako *C90*).

Samotný jazyk *C* byl vytvořen Dennisem Ritchie a Brianem Bernighanem v roce 1978 pro účely operačního systému *Unix*. Jazyk byl poprvé publikován v knize „The C Programming Language“ [4]. Tato verze jazyka je dnes označována jako *K&R C*.

Pro následující text kapitoly bylo čerpáno ze standardu *ANSI C* [1].

3.1 Fáze překladu

Standard [1] definuje 8 fází překladu, kterými musí projít zdrojový kód k výslednému programu nebo knihovně:

1. Zdrojový soubor je převeden do znakové sady zdrojového kódu a případné trigraphy[1] (viz tab. 3.1) jsou nahrazeny.

2. Znaky odřádkování, kterým předchází znak zpětného lomítka jsou zahozeny a je tak spojeno více řádek do jedné.

3. Zdrojový kód je rozdělen na tokeny preprocesoru a sekvence bílých znaků (včetně komentářů). Každý komentář a sekvence bílých znaků (bez odřádkování) jsou nahrazeny jednou mezerou.

4. Direktivy preprocesoru jsou vyhodnoceny a makra nahrazeny. Direktiva `#include` způsobí zpracování vkládaného souboru pro fáze 1 – 4, rekurzivně.

5. Každá escape sekvence v znakových konstantách nebo řetězcích je nahrazena odpovídajícím znakem.

6. Sousedící řetězce jsou spojeny.

7. Bílé znaky nejsou dále důležité a jsou zahozeny. Tokeny preprocesoru jsou převedeny na tokeny jazyka.

[1] Trigraphy jsou speciální sekvence znaků, které jsou nahrazeny jedním znakem. Vznikly především kvůli omezenému počtu znaků v používaných znakových sadách, které neobsahovaly znaky potřebné programovacím jazykem.

8. Dochází k linkování do výsledného obrazu programu, který obsahuje veškeré informace potřebné pro správný běh v cílovém prostředí.

Tabulka 3.1: Trigraphy

Trigraph	Náhrada
??=	#
??(	[
??/	\
??)	]
??'	^
??<	{
??!	|
??>	}
??-	~

3.2 Preprocesor

Dříve byl preprocesor oddělen od překladače a sloužil k podmíněné úpravě zdrojového kódu pro samotný překlad. Standard preprocesor zahrnuje do překladových fází (viz 3.1) a je s preprocesorem počítáno jako s nedílnou součástí jazyka.

V předmluvě [2] jsou uvedeny dva druhy implementace preprocesoru, kdy první (*text-to-text*) je oddělenou částí a druhá (*token-oriented*) je přímou součástí překladače.

3.2.1 Vkládání souborů

Jedná se o asi nejpoužívanější část preprocesoru. Preprocesor vloží místo direktivy obsah uvedeného souboru. Toto se používá především ke vkládání hlavičkových souborů, kde jsou deklarované symboly jiné překladové jednotky a není tak nutno je vždy ručně uvádět (překladač potřebuje znát symboly před jejich použitím).

Standard [1] definuje tři druhy zápisu. První uvádí název vkládaného souboru do `<>` a druhý do `""`. Rozdíl mezi nimi je minimální a standard [1] říká, že druhý způsob nemusí být podporován a v tom případě (nebo pokud selže), tak je interpretován jako první způsob. Standard [1] nedefinuje přesné chování jednotlivých zápisů a nechává to na jednotlivých implementacích. Většinou ale platí, že první způsob slouží ke vkládání systémových hlavičkových souborů a druhý ke vkládání uživatelských. Poslední druh zápisu využívá možností náhrady maker, kdy výsledek musí odpovídat jednomu z prvních dvou zápisů.

Vkládaný soubor musí být zpracován znovu podle bodů 1 – 4 fází překladu (viz 3.1).

Zdrojový kód 3.1: Vkládání souborů

```
1 #include <stdlib.h>
2 #include "file.h"
```

3.2.2 Podmíněný překlad

Podmínky dovolují upravovat výsledný zdrojový soubor na základě výsledku vyhodnocení výrazů. Lze tak specifikovat kód, který bude přeložen jen za určitých podmínek (např. cílová platforma).

Podmíněný překlad se ve zdrojovém kódu uvádí pomocí direktiv `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` a `#endif`. V případě `#if` a `#elif` se uvádí výraz, který je vyhodnocen a na jehož výsledku se další kód zpracovává. Ve výrazu lze uvést operátor `defined`, který vyhodnocuje existenci makra. Direktivy `#ifdef` a `#ifndef` testují existenci makra přímo.

Na následující fragmentu kódu je vidět použití podmíněného překladu pro různé platformy a příklad použití makra `NDEBUG`.

Zdrojový kód 3.2: Podmíněný překlad

```
1  #if defined WIN32
2  /* Jen pro Windows */
3  #elif defined linux
4  /* Jen pro linux */
5  #else
6  /* Ostatní */
7  #endif
8
9  #ifndef NDEBUG
10 /* Kód, který bude jen v debug verzi aplikace */
11 #endif
```

3.2.3 Makra

Makra jsou nejsilnější zbraní preprocesoru. Dovolují nahrazovat tokeny zdrojového kódu za jiné tokeny a v kombinaci s podmíněným překladem je možno generovat rozdílné zdrojové kódy.

Standard [1] definuje dva druhy maker:

object-like Provede se náhrada jediného identifikátoru.

function-like Makro se chová jako funkce tj. zadávají se argumenty, které jsou následně použity při nahrazování makrem.

Typ makra se udává při jeho definici uvedením seznamu parametrů v případě *function-like*. Není-li seznam parametrů uveden, jedná se o *object-like* makro.

Na základě tohoto typu se provádí náhrada makra ve zdrojovém kódu. Pokud je makro definováno jako *function-like*, tak v případě neuvedení argumentů nebude nahrazeno. *Object-like* makro je nahrazeno vždy.

Použití maker je uvedeno ve fragmentu kódu 3.3, kde jde vidět chování náhrady jednotlivých typů maker.

3.2.3.1 Úprava argumentů

Argumenty předané *function-like* makru lze upravit pomocí speciálních operátorů. Příklady použití jednotlivých operátorů jsou uvedeny ve fragmentu kódu 3.3.

Operátor # Tento operátor je schopen vytvořit z uvedeného tokenu nebo seznamu tokenů řetězec znaků.

Operátor ## Operátor je určen na spojování tokenů. Spojením dvou tokenů vznikne jeden token. Tento token musí být platný a proto nelze spojovat jakékoliv tokeny. Např. spojit číslo a řetězec nelze.

Zdrojový kód 3.3: Makra

```
1  /* Object-like makro */
2  #define OBJECT 5
3  /* Function-like makro */
4  #define FUNC(a, b) ((a) + (b))
5  /* Název */
6  #define NAME func
7  /* Generování řetězce */
8  #define STR(str) # str
9  /* Spojování tokenů */
10 #define FNAME(n) fname ## n
11
12 int a = OBJECT;                /* -> int a = 5; */
13 int b = FUNC(3, 2);            /* -> int b = ((3) + (2));
      */
14 int c = FUNC;                  /* -> int c = FUNC; */
15 int NAME(int);                 /* -> int func(int); */
16 char str[] = STR(Hello world); /* -> char str[] = "Hello
      world"; */
17 void FNAME(1)(void);           /* -> void fname1(void); */
18 void FNAME(X)(int x);          /* -> void fnameX(int x);
      */
```

3.3 Syntaxe

3.3.1 Datové typy

Standard [1] definuje několik základních datových typů a další lze definovat pomocí klíčového slova `typedef`. Základní typy se rozdělují na celočíselné (viz 3.3.1.1), s plovoucí čárkou (viz 3.3.1.2) a na speciální typ `void`.

Celočíselné lze specifikovat uvedením klíčového slova `signed` a `unsigned`, které udávají zda může nebo nemůže typ ukládat znaménko. Pokud není uvedeno `signed` ani `unsigned`, typ samotný je znaménkový (tj. `signed`). Jedinou výjimkou je typ `char`, který může být, z historických důvodů, implicitně `signed` i `unsigned` (záleží na platformě, kde je použit).

Přehled garantovaných minimálních rozsahů a přesností jednotlivých datových typů je uveden v tabulce B.1.

3.3.1.1 Celočíselné typy

`char`, `short`, `int`, `long` (a jejich `signed` a `unsigned` verze).

3.3.1.2 Typy s plovoucí čárkou

`float`, `double`, `long double`.

3.3.1.3 Výčtové typy

Standard [1] dovoluje definovat vlastní výčtové typy, které slouží k definici číselných konstant. Výčtový typ se definuje klíčovým slovem `enum`, následovaným nepovinným názvem a seznamem konstant ve složených závorkách. Ke každé konstantě je možno uvést její hodnotu, která musí být typu `int` (může jít i o konstantní výraz, ale jeho výsledek musí být taktéž typu `int`). Není-li explicitně uvedena hodnota konstanty, má první konstanta hodnotu 0 a ostatní o 1 větší než předchozí konstanta. Oba způsoby lze kombinovat a dosáhnout tak i případu, kdy dvě a nebo více konstant mají stejnou hodnotu, což standard dovoluje.

Zdrojový kód 3.4: Příklad definice výčtového typu

```
1  /* Deklarace výčtového typu */
2  enum ovoce {
3      jablko, /* = 0 */
4      hruska = 5,
5      pomeranc /* = 6 */
6  };
7
8  /* Použití */
9  enum ovoce ov = jablko;
```

3.3.1.4 Struktury

Struktury jsou datové typy složené ze sekvenčně uložených členských proměnných a mají nepovinný název. Každý člen je definován typem a názvem. Typ členské proměnné musí být kompletní, což znamená, že musí být definován a ne jen deklarován. To neplatí v případě ukazatelů, jejichž velikost je dána a není potřeba znát velikost datového typu.

Zdrojový kód 3.5: Příklad definice struktur

```
1  /* Struktura pro uložení čísla a názvu */
2  struct data {
3      int id;
4      char name[32];
5  };
6
7  /* Definice struktury pro obousměrný spojový seznam */
8  struct node {
9      struct node *next, *prev;
10     struct data value;
11 };
```

3.3.1.5 Uniony

Union je velice podobný struktuře, jediný rozdíl je v uložení členských proměnných. Ty jsou uloženy přes sebe a union má tedy velikost největší členské proměnné.

Zdrojový kód 3.6: Příklad použití unionu

```
1 /* Union pro uložení int nebo float */
2 union intfloat {
3     int intval;
4     float floatval;
5 };
6
7 union infloat a;
8 a.intval = 5;
9 a.floatval = 5.f; /* a.intval != 5 */
```

3.3.1.6 Kvalifikátory

Standard [1] nově přinesl dvě klíčová slova: `const` a `volatile`.

Klíčové slovo `const` zaručuje, že hodnota proměnné nemůže být změněna (může být uvedena jen při definici).

Klíčové slovo `volatile` říká, že proměnná (jinde použít nelze) může být upravena nezávisle na programu.

Obě klíčová slova lze použít v kombinaci s ukazateli. Např. v případě `const` lze zamezit možnosti změnit hodnotu ukazatele, ale změnit hodnotu, na kterou ukazuje, povolit. Následující fragment kódu pro použití klíčového slova `const` byl převzat z přehledu změn, které standard přinesl [7]. V případě `volatile` byl fragment kódu převzat přímo ze standardu [1].

Zdrojový kód 3.7: Příklady použití kvalifikátorů typu

```
1  /* Proměnná může být měněna mimo program, ale programem
      být změněna nemůže */
2  extern const volatile int real_time_clock;
3
4  /* Použití s ukazately */
5  int const *a;
6  int *const b;
7  int *const *c;
8
9  a = NULL;  /* ok */
10 *a = 0;    /* error */
11 b = NULL;  /* error */
12 *b = 0;    /* ok */
13 c = &b;    /* ok */
14 *c = NULL; /* error */
15 **c = 0;   /* ok */
```

3.3.2 Proměnné

Jazyk *ANSI C* je typovaným jazykem a proto každá proměnná musí mít určený datový typ. Datový typ nemusí být v některých případech uveden[2] a potom je použit implicitní datový typ `int`.

U proměnné lze specifikovat, kde bude proměnná uložena:

`auto` Automatické uložení.

`extern` Proměnná je uložena v jiné překladové jednotce.

`static` Proměnná existuje po celou dobu běhu programu a je dostupná jen v oblasti, kde je deklarována (překladová jednotka nebo funkce).

`register` Proměnná je uložena v registru procesoru (nelze na ní získat ukazatel).

V externí deklaraci se nesmí, dle standardu, objevit `register` a `auto`.

Zdrojový kód 3.8: Příklady proměnných

```
1 /* Implicitní typ int */
2 a = 5;
3 /* Proměnná definovaná jinde */
4 extern char b;
5 /* Konstantní pole znaků */
6 const static char[] b = "konstanta";
```

3.3.3 Funkce

Deklarace funkce je velice podobná deklaraci proměnné, jen v případě funkce je uveden seznam parametrů v závorkách. Seznam parametrů (i prázdný) jako jediný odděluje deklarace funkce od proměnné.

3.3.3.1 Typ uložení

Stejně jako u proměnných (viz 3.3.2) lze určit, kde bude funkce uložena. Pro funkce lze určit jen dva typy uložení:

`extern` Funkce je definována v jiné překladové jednotce a nebo bude definována.

`static` Funkce bude dostupná jen v překladové jednotce, kde je definována.

3.3.3.2 Prototyp

Jako prototyp se označuje deklarace funkce, která se využívá ke správnému volání dané funkce, jež může být definována v jiné překladové jednotce a nebo níže ve stejné překladové jednotce.

[2] Jen musí být uvedeno něco, podle čeho překladač pozná, že se jedná o deklaraci proměnné. U globálních proměnných se jedná implicitně o deklaraci a u lokálních je nutné uvést alespoň jednu část datového typu (specifikátor, kvalifikátor).

3.3.3.3 Návratová hodnota

Dle standardu [1] může být návratovou hodnotou typ `void` a nebo jakýkoliv datový typ, kromě pole.

3.3.3.4 Parametry

Parametry funkcí lze uvést dvěma způsoby, kdy první pochází z původního *K&R C* a druhý byl převzat z jiných programovacích jazyků.

První způsob zapisuje v závorkách jen názvy parametrů (implicitní typ `int`) a parametry se uvádějí jako deklarace před tělem funkce. Tento způsob se v dnešní době nepoužívá, jelikož neposkytuje informaci o typech parametrů bez definice funkce.

Druhý způsob umožňuje zapsání typu parametru přímo před parametr a tento způsob se dnes převážně používá.

3.3.3.5 Proměnný počet parametrů

Pomocí „...“ na konci seznamu parametrů, lze umožnit předat libovolný počet argumentů při volání funkce. Pro tyto argumenty již není možné na úrovni překladu určit datový typ a je zcela na volajícím, jaké hodnoty argumentu uvede a na volané funkci, zda dovede datové typy argumentů správně určit a to včetně jejich počtu.

Zdrojový kód 3.9: Příklady funkcí

```
1  /* Prototyp funkce s návratovou hodnotou int */
2  a(void);
3
4  /* Prototyp funkce bez návratové hodnoty */
5  void b(int a, int b);
6
7  /* Definice funkce */
8  char* getstr(int pos) {
9      /* ... */
10 }
11
12 /* Parametry ve starém formátu (K&R C) */
13 int kar_add(a, b)
14 int a;
15 /* int b; */
16 {
17     return a + b;
18 }
```

3.3.4 Konstrukce

Konstrukce se mohou vyskytovat jen v těle funkcí. Popisují chování funkce na základě dodaných dat.

3.3.4.1 Podmínky

Standard [1] definuje 2 druhy podmínkových konstrukcí: `if` a `switch`.

if Konstrukce `if` vyhodnocuje zadaný výraz, který musí být skalární, a na základě jeho výsledku se volí odpovídající cesta. Konstrukci lze rozšířit o `else` větev, kdy se při nesplnění vyhodnocované podmínky vykonají příkazy v `else` větvi.

Zdrojový kód 3.10: Konstrukce if

```
1  /* Test hodnoty proměnné c */
2  if (c == 10)
3      exit();
4
5  /* Přiřazení ve výrazu podmínky */
6  if (a = 0)
7      exit(); /* nikdy se nesplní */
8  else
9      c = 0x8;
```

switch Konstrukce `switch` se hodí na testování jednoho výrazu na několik různých konstantních hodnot (daných třeba konstantním výrazem). Konstrukce je schopna testovat jen celočíselné hodnoty, které mohou být v seznamu uvedeny jen jednou. Je-li výsledek vyhodnocení výrazu hodnota, která není v seznamu uvedena, je vykonána část `default` (je-li uvedena).

Zdrojový kód 3.11: Konstrukce switch

```
1  switch (c) {
2  case 2:
3      a += 2;
4      /* Provede se i část pro hodnotu 3 */
5  case 3:
6      doSomething();
7      break;
8  case 3 + 2 * 4:
9      /* Může zde být konstantní výraz */
10 default:
11     return;
12 }
```

3.3.4.2 Cykly

Jazyk podporuje tři druhy cyklů: `for`, `while` a `do-while`.

for Cyklus se skládá ze tří nepovinných výrazů. První výraz je vyhodnocen jednou na začátku cyklu a hodí se na inicializaci proměnných. Druhý výraz slouží jako podmínka pro běh cyklu. Je-li její výsledek `0`, cyklus je ukončen. Třetí výraz je vyhodnocen při každé iteraci a hodí se k úpravě proměnných.

Zdrojový kód 3.12: Cyklus for

```
1  for (i = 0; i < 10; ++i) {
2      data[i] = i;
3  }
```

while Cyklus obsahuje jeden podmínkový výraz, který se vyhodnocuje před každým vyhodnocením těla cyklu, a ten určuje, zda proběhne další iterace.

Zdrojový kód 3.13: Cyklus while

```
1 while (c != 'A') {
2     c = getc();
3 }
```

do-while Tento cyklus je podobný cyklu `while`, jen vyhodnocování výrazu dochází až po vyhodnocení těla cyklu.

Zdrojový kód 3.14: Cyklus do-while

```
1 do {
2     c = getc();
3 } while (c != 'A');
```

3.3.4.3 Výrazy

Jedná se o sekvenci operátorů a operandů, která specifikuje výpočet hodnoty, určuje objekt nebo funkci, generuje vedlejší jev, nebo provádí kombinaci předchozích.

Standard [1] definuje pořadí (prioritu) vyhodnocování operátorů, jen pro operátory volání funkce `()`, `&&`, `||` a `?:` není pořadí pod-výrazů definováno.

Zdrojový kód 3.15: Příklady výrazů

```
1 f(5);
2 c = 1 * 8 << 2;
3 f1(f2(), f3(d) * 3 + 1);
4 c++ = 'c';
5 "hello,␣world!";
```

3.4 Knihovny

Součástí standardu [1] je několik knihoven, které podporují základní funkčnost a dovolují tak větší přenositelnost zdrojových kódů bez nutnosti využívat *API* cílové platformy.

Mezi tyto standardizované knihovny patří:

- Knihovna pro diagnostiku (`assert.h`).
- Knihovna pro práci se znaky (`ctype.h`).
- Knihovna pro práci s chybami (`errno.h`).
- Knihovna pro typy s plovoucí datovou čárkou (`float.h`).
- Knihovna s definovanými velikostmi celočíselných typů (`limits.h`).
- Lokalizační knihovna (`locale.h`).

- Matematická knihovna (`math.h`).
- Knihovna pro vzdálené skoky (`setjmp.h`).
- Knihovna pro práci se signály procesu (`signal.h`)
- Knihovna pro práci s proměnným počtem argumentů funkce (`stdarg.h`).
- Základní definice (`stddef.h`).
- Knihovna pro práci se vstupy a výstupy (`stdio.h`).
- Knihovna základních funkcí (`stdlib.h`).
- Knihovna pro práci s řetězci (`string.h`).
- Knihovna pro práci s časem (`time.h`).

3.5 Verze jazyka C

Jelikož jazyk C je velice starý a za dobu jeho existence vzniklo několik verzí. O standardizaci jazyka se v současné době stará organizace *ISO*.

3.5.1 K&R C

Takto je označována původní verze jazyka před standardizací. Jazyk byl popsán v knize „The C Programming Language“ [4].

3.5.2 ANSI C

Jazyk popisovaný v této práci. Občas je označován jako *C89*, aby byla zachována spojitost s ostatními standardy jazyka.

3.5.3 C90

Verze standardizovaná jako organizací *ISO* v roce 1990 pod ISO/IEC 9899:1990. Jedná se v podstatě o jazyk *ANSI C*.

3.5.4 C94

Málo známá verze jazyka. Jedná se jen o upřesnění verze *C89* [8], kdy byly přidány tzv. digraphy a několik změn pro podporu více znakových sad.

3.5.5 C99

Novější verze jazyka, standardizovaná pod názvem ISO/IEC 9899:1999, je vylepšena o inline funkce, nové datové typy, pole s proměnnou délkou, makra s proměnným počtem parametrů a další [10]. I když se jedná o starý standard, tak není všemi překladači ještě zcela podporován[3].

[3]Většina překladačů standard podporuje, jen překladač VC++ nemá žádnou podporu, jelikož se Microsoft plně zaměřuje na jazyk C++.

3.5.6 C11

Poslední verze jazyka, která byla standardizovaná pod názvem ISO/IEC 9899:2011 v prosinci 2011. Novinkou ve standardu je podpora vláken, vylepšená podpora *UNICODE*, statické asserce[4] atd.

[4]Alternativa k `#if` a `#error`, která dovoluje vyhodnocovat podmínky až v pozdější části překladu společně s jazykem a ne jen preprocesorem.

Kapitola 4

Metrická analýza

Zaměřením této práce je porovnávání dat získaných metrickou analýzou. Pro potřeby porovnávání bylo možné použít několik typů přístupu získávání metrik.

Prvním bylo získání metrik z objektových souborů po přeložení zdrojových kódů. Tento přístup měl vadu v tom, že z objektových souborů nelze získat mnoho metrik. Jsou tam jen metriky o existujících symbolech, které potřebuje linker na spojení s ostatními objektovými soubory. Nelze tak zjistit, zda je onen symbol proměnná nebo funkce, jaký má proměnná typ, jaké má funkce návratový typ a parametry. Použít by se dal přeložený kód v objektovém souboru, ale není tak jisté, jak se kód změní při změně zdrojového kódu. Hodně záleží na chování překladače jak jednotlivé části zdrojového kódu přeloží a jednoduchou změnou může ve zdrojovém kódu překladač poznat známý kus kódu a přeložit ho jinak.

Dalším způsobem byla přímá analýza zdrojového kódu, vytvoření popisné struktury a určení metrik pro jednotlivé části. Tento způsob je ideální, jelikož umožňuje získat velké množství metrik, které využívá i samotný překladač při překladu. Ze zdrojového kódu lze určit přesně funkce, jejich tělo, použité proměnné, chování funkcí a programu jako celku.

Druhý způsob poskytuje větší množství metrik, proto byl zvolen pro potřeby porovnávání zdrojových kódů.

4.1 Funkce

Nejvíce metrik obsahují funkce, což je dáno jejich rozsahem ve zdrojových kódech. Pro naše účely je vhodné určit tyto hodnoty:

- název funkce,
- návratový typ,
- parametry,
- cykly,
- podmínky,
- deklarace.

Podobné funkce mají ve většině případů shodný návratový typ a parametry. Název funkce může být snadno změněn, takže by neměl hrát velkou roli. Největší vliv na shodnost má tělo funkce, které nelze tak snadno změnit beze změny chování a funkčnosti. Pro správnou funkčnost musí být zachován algoritmus, podle kterého funkce pracuje a proto je většinou zachován počet cyklů a podmínek a ve většině případů i deklarací.

4.1.1 Tělo funkce

Těla shodných funkcí ve většině případů obsahují stejný počet deklarací proměnných, podmínek a cyklů. Bez výrazné změny chování funkce nelze tyto metriky změnit a výrazná změna chování funkce vyžaduje větší zapojení autora tj. snižuje se shodná část kódu.

Obejít to lze několika způsoby, které nejsou vhodné do samotné funkčnosti zdrojového kódu. Jedná se o přidání podmínek, které se nikdy nesplní, cykly, které nic neprovádějí a deklarace proměnných, které nejsou použity. Všechny tyto úpravy se špatně odhalují, jelikož je nutné zapojení větší analýzy. Pro podmínky by bylo nutné vyhodnotit výraz a určit zda může být někdy splněn, pro cykly zjistit zda neprobíhá předem daný počet cyklů a zda cyklus něco provádí. U proměnných by bylo nutné určit, zda jsou v kódu použity.

4.1.1.1 Deklarace

Deklarace uvnitř funkce odpovídají deklaracím globálních proměnným. Menší problém vzniká v tom, že jazyk *ANSI C* dovoluje deklarovat funkci (tzv. prototyp) i uvnitř funkce. Tento způsob deklarace není moc znám a ani se moc nepoužívá, ale nelze předpokládat, že se ve zdrojovém kódu nemůže vyskytnout. Pro naše účely budou deklarace funkcí v tělech funkcí ignorovány.

4.1.1.2 Podmínky

Existují dva druhy podmínek: `if`, `switch`. V určitých případech lze přepsat jeden na druhý. Bohužel je podmínka `switch` složitá a porovnání s `if` je velice obtížné.

4.1.1.3 Cykly

Jak je zmíněno v kapitole o *ANSI C* (viz kapitola 3), lze určit tři druhy cyklů. Rozdíl mezi `while` a `do-while` je jen v místě vyhodnocování podmínky, zato cyklus `for` může obsahovat navíc další dva výrazy. Tento cyklus lze navíc přepsat do podoby cyklu `while`, proto nemůže být bez větší analýzy kontrolován typ cyklu. Pro porovnávání tak musí stačit jen samotná přítomnost cyklu.

4.1.1.4 Výrazy

Nejsložitější část jazyka pro uložení do popisné struktury. Problém je především v rekurzi výrazů, kdy jeden výraz může být složen z několika dalších a ty také. Výraz může obsahovat volání funkcí, které je problém nějakým způsobem uložit. Pro potřeby porovnávání musí stačit jen informace o přítomnosti výrazu.

4.2 Proměnné

Ve zdrojovém kódu se objevují dva druhy proměnných: globální, lokální. Globální jsou definované pro celou překladovou jednotku a lokální jsou definované uvnitř funkce. Nejvýraznější rozdíly mezi proměnnými lze nalézt v datovém typu, jenž je popsán níže. Název proměnné nehraje takovou roli, jelikož lze snadno změnit.

4.3 Datové typy

Metrik, které lze určit u datových typů, je mnoho. Z jazyka *ANSI C* lze určit následující metriky:

- základní typ (specifikátory),
- kvalifikátory,
- ukazatele,
- velikosti pole,
- struktura,
- výčtový typ.

Ne všechny metriky může mít datový typ určen. Např. struktura a výčtový typ nemůžou být uvedeny v jednom typu. Tyto údaje říkají, že datový typ je přímo struktura nebo výčtový typ, proto nemůže být datový typ zároveň strukturou a výčtovým typem.

Při porovnávání je problém s množstvím určených metrik. Pokud je datový typ např. `int`, jedná se o datový typ, který je ve zdrojovém kódu přítomen téměř všude a z globálního hlediska by neměl mít takovou váhu. Na druhou stranu datový typ `int**[5]`, má uvedeno více metrik a ve zdrojovém kódu se takový datový typ nebude vyskytovat příliš často, proto by měl mít větší váhu při porovnávání. Hodnocení datových typů na základě statistické přítomnosti daných datových typů ve zdrojovém kódu by bylo zbytečně složité, proto nebylo do práce zahrnuto. Obnášelo by to projít celou popisnou strukturu, najít veškeré datové typy, vytvořit pro ně statistiku přítomnosti a tu pak využít při porovnávání. Další problém by byl s hodnocením typu předávaným výše, kdy by hrozilo, že při větším počtu primitivních datových typů by mohly být zdrojové kódy falešně označeny za nepodobné.

4.4 Struktury

Seskupené proměnné jsou označovány jako struktury. Jejich metriky jsou pro porovnávání důležitější, protože toto seskupení proměnných výrazně odlišuje strukturu od samostatně definovaných proměnných.

Metriky vhodné pro porovnávání jsou:

- členské proměnné,
- pořadí členských proměnných,

- velikost struktury.

Většinu těchto metrik lze jednoduchou úpravou zdrojového kódu změnit. Změna pořadí členských proměnných ovlivní druhou metriku a přidání další proměnné ošálí první a poslední metriku.

Odstranění těchto nedostatků by bylo možné, ale pro tuto práci složité. V případě změny pořadí by se problém snadno vyřešil řazením proměnných podle datového typu, ale bylo by nutné definovat, jak se mají datové typy řadit. U přidání členských proměnných by bylo nutné kontrolovat, zda je daná členská proměnná použita a jak. To obnáší průchod celého zdrojového kódu.

4.5 Výčtové typy

Výčtové typy jsou z hlediska metriky a jejího použití v porovnávání téměř nepoužitelné. V jazyce *ANSI C* to jsou jen seskupené konstanty. Platnost konstant je globální a není vázána na daný výčtový typ. Snadno tak lze přeskupit konstanty výčtových typů mezi sebou beze ztráty funkčnosti. Jediným atributem, který lze použít pro porovnávání, je název konstanty.

Kapitola 5

Technologie

5.1 Požadavky

Před samotným vývojem bylo nutné si promyslet, jakou funkcionalitu budou nástroje podporovat a jaké požadavky na ně budou kladeny:

- Zpracování zdrojového kódu napsaného v jazyce *ANSI C* (včetně hojně používaných věcí mimo standard jako jsou *C++* komentáře nebo deklarace proměnných mimo začátek bloku).
- Uložení otisku zdrojového kódu do vhodného úložiště pro budoucí potřeby.
- Vypsání informací o zdrojovém kódu z otisku.
- Porovnání zpracovaných zdrojových kódů (tj. porovnání otisků).
- Platformní nezávislost.

5.2 Programovací jazyk

Z hlediska požadavku na platformní nezávislost aplikací, bylo nutné zvolit programovací jazyk, který není vázaný jen na omezený počet platforem. Jelikož aplikace budou určeny na práci s velkým množstvím otisků zdrojových kódů, bylo potřeba zvolit jazyk, který je rychlý a neomezuje ho žádný interpret, což vylučuje všechny skriptovací jazyky. Výběr se tak omezil na nejvíce používané nativní a řízené jazyky:

C Asi nejvíce přenositelný jazyk[1]. Překládá se do nativního kódu pro danou platformu, proto ho neovlivňuje rychlost interpretu. Zaručuje velkou rychlost vykonávání správně napsaného kódu.

C++ Kdysi jen objektové rozšíření programovacího jazyka *C*. Dnes se jedná o samotný jazyk, který je za svou složitost pověstný. Stejně jako jazyk *C* je překládán do nativního kódu.

Java Velice oblíbený jazyk s podporou pro mnoho platforem. Nevýhodou je paměťová náročnost (téměř všechny objekty ukládá na haldu) a občasné zpomalení díky *Garbage Collectoru*.

[1] Tedy v rámci přenositelnosti použitých knihoven.

C# Propracovaný jazyk s velkým množstvím knihoven. Nevýhodou je omezení na platformu *Windows*. Pro ostatní platformy existuje *Mono*, které nelze stále považovat za použitelné. Ač se jedná o řízený jazyk, díky *JIT* překladači je schopen být rychlý.

Programovací jazyk C# z kandidátů vypadl jako první, jelikož je omezen jen na platformu *Windows*. Java taktéž nebyla vhodným kandidátem, protože je zaměřovaná spíše na uživatelské aplikace, kde není potřeba až tak rychlého výpočtu a je dostatek operační paměti.

Z hlediska zaměření této práce byl nejdříve zvolen jazyk *C*, ale po pozdějším zjištění, že se projekt stává nepřehledným a lepším řešením by byl jazyk podporující *OOP*, byl zvolen jazyk *C++*. Přesněji byl zvolen ve verzi *ISO C++98*[2], která je přesně standardizována a je podporována většinou překladačů. Mezi uvažovanými byl i jazyk *ISO C++11*[3], ale ten byl standardizován až na konci roku 2011 a většina překladačů ho v té době podporovala jen okrajově.

5.3 ANSI C parser

Hlavním účelem parseru bude načtení zdrojového kódu a uložení jeho popisu do vnitřní struktury, se kterou se bude dále pracovat.

Dobrým nápadem bylo využití schopností překladačů, ale většina z nich není určena na výstup v této části zpracování zdrojového kódu. Např. *GCC* je schopno vyprodukovat *XML* soubor, který je velice složitý a jeho zpracování by bylo časově náročné. *Clang* je už na toto lépe navržen, ale existují verze jen pro *Mac* a *Linux* a navíc v době rozhodování nebyl moc rozšířen.

Použití existující knihovny byl taktéž problém, jelikož většina z nich je vázána na nějaký framework či jinou větší knihovnu a nebo se jedná o implementaci pro nějaký skriptovací jazyk.

Jednou možností bylo použít část zdrojových kódů překladače *GCC*, ale kvůli složitosti a provázanosti s ostatními části překladače a sdílení kódu pro překlad programovacího jazyka *C++* a *Objective-C* to nebylo možné.

Nakonec bylo rozhodnuto o vytvoření vlastní implementace *ANSI C* parseru, který bude pracovat na principu rekurzivního sestupu (tj. nebude využívat žádný generátor lexikálního analyzátoru). Toto řešení poskytuje přímé generování popisné struktury při parsování. Eliminuje tak jakýkoliv mezistupeň a snižuje rychlost zpracování zdrojového kódu. Bylo by sice možné pomocí generátorů lexikálních analyzátorů (např. *Lex*) vygenerovat kód, ale výsledný kód by byl v jazyce *C* a špatně by se do něj přidával kód na ukládání do popisné struktury. Z tohoto hlediska bylo lepší volbou vytvoření celého parseru.

5.4 ANSI C preprocesor

Před samotným zpracováním zdrojového kódu parserem je nutné zpracovat kód preprocesorem, který vytvoří výsledný kód na základě vyhodnocení direktiv preprocesoru.

[2]Existuje také standard C++03, ale ten se liší od C++98 minimálně (jen upřesnění chování), proto se většinou uvádí jen C++98.

[3]Dříve C++0x.

Prvním nápadem bylo využít preprocesor přítomného překladače. Toto řešení je bohužel závislé na existenci předem definovaného překladače, což znemožňuje přenositelnost. Dále je zde také problém s výstupním kódem generovaným preprocesorem překladače, který ve většině případů přidává vlastní kód mimo standard. Tento kód překladači pomáhá v optimalizaci kódu, ale brání ve správném načtení kódu vlastním parserem. Řešením je správné nastavení parametrů preprocesoru, ale to opět omezuje přenositelnost a vyžaduje znalost použitého překladače.

Další možností bylo použít existující knihovnu preprocesoru a napojit ji na programy. Bohužel moc takových knihoven neexistuje:

smartcpreprocessor Je napsán v jazyce F#, který je vázán na *Windows*. Na *Linuxu* lze použít pod *Mono*, které ale nelze stále považovat za dostatečně použitelné a navíc mixovat nativní kód a řízený není vždy dobrý nápad.

Wave Jedná se sice o *C99/C++* preprocesor, ale rozdíl oproti *ANSI C* je minimální. Hlavní problém je nutnost použití *Boost* knihoven, které zvětší množství zdrojového kódu a sníží rychlost překladu aplikací a je zbytečné jen kvůli této malé části práce přibalit tak velkou knihovnu.

Poslední možností je napsání vlastního preprocesoru. Toto řešení zbaví nástroje závislostí na dalších knihovnách a navíc může být napsán přímo na míru a být tak snáze integrován do vytvořených nástrojů.

5.5 Uložení otisku zdrojového kódu

Zpracováním zdrojového kódu vytvoří nástroj v operační paměti otisk daného zdrojového kódu a lze s ním snadno pracovat. Tento otisk existuje jen po dobu běhu nástroje a nebylo by ho možné využít v ostatních nástrojích. Z tohoto důvodu je potřeba ukládat zpracovaný zdrojový kód v dlouhodobějším úložišti. Tento problém lze vyřešit několika způsoby a to včetně jeho ignorování.

1. Neukládat žádný speciální soubor a pokaždé zpracovat původní zdrojový kód
    - přednosti
        - Není potřeba ukládat otisk zdrojového kódu na disk (a pak načítat) a vymýšlet vlastní datový formát.
    - nedostatky
        - Nutnost pokaždé zpracovávat zdrojový kód.
        - Může být pomalé při větším počtu zdrojových kódů.
2. Ukládat otisk do binárního souboru
    - přednosti
        - Soubor má přesně daný formát a lze tak rychle načíst celý otisk zdrojového kódu.
        - Není nutné znovu zpracovávat původní zdrojový kód.
        - Při větším počtu zdrojových kódů je rychlejší než předchozí řešení.

- nedostatky
  - Vytvoření vlastního formátu datového souboru.
  - Při změně datového formátu souboru je potřeba znovu zpracovat zdrojové kódy.

3. Využití objektové databáze

   - přednosti
     - O uložení dat se stará databáze, není tedy potřeba vymýšlet vlastní datový formát.
     - Není potřeba načítat celý otisk – lze si vyžádat jen potřebné části.
   - nedostatky
     - Závislost na knihovně databáze.
     - Úprava tříd otisku, aby vyhovovaly databázi.
     - Nutnost volby vhodné objektové databáze.

Ukládání do vlastního souboru byla jednoznačná volba. Nástroje se souborem mohou pracovat velice rychle a není potřeba žádná další knihovna. Nevýhodou je potřeba návrhu vlastního datového formátu a následná implementace pro ukládání a načítání. Změna datového formátu se většinou neděje moc často, proto nebude nutné tak často obnovovat soubory s otisky.

Kapitola 6

Realizace

Hlavním cílem při návrhu projektu byla případná rozšířitelnost o další jazyky a znovupoužitelnost již vytvořeného kódu. Proto bylo využito ve velké míře *OOP* a vytvořeny základní třídy, jež byly rozšířeny pro specifický programovací jazyk, v našem případě *ANSI C*.

6.1 Rozdělení projektu

6.1.1 Analyzátor kódu

Pro potřeby práce bude potřeba vytvoření několik samostatných aplikací, které budou sdílet část pro práci s otiskem zdrojového kódu. Z tohoto hlediska je výborným řešením vytvoření sdílené části jako knihovny.

Tato část se bude starat o většinu práce. Především bude zpracovávat zdrojový kód a vytvářet tak jeho otisk do odpovídající datové struktury, kterou poskytne ostatním nástrojům.

6.1.2 Překladač

Pro vytvoření správného otisku je nutné přesně simulovat chování překladače. Je to dáno tím, že v *makefile* zpracovávaného projektu je přesně popsáno jak má být aplikace přeložena a může tam být přidáno speciální makro, které změní výsledný kód apod. V případě linkeru je nutné vědět jaké objektové soubory se spojují. Ve zpracovávaném projektu může být víc zdrojových kódů, kdy se některé z nich ve výsledném binárním souboru nepoužijí.

6.1.3 Čtečka otisků zdrojových kódů

Pro účely procházení i kontroly otisků zdrojových kódů je výhodné si vytvořit aplikaci, která bude schopna vytisknout přehled otisků uložených v datových souborech. Hlavní zaměření je výpis statistik daného objektového nebo binárního souboru, případně i výpis detailu uloženého symbolu.

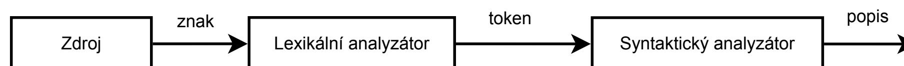
6.1.4 Porovnávání binárních otisků

Nejdůležitější nástroj této práce. Stará se o porovnávání zpracovaných samostatných prací mezi sebou a hledá shody mezi nimi. Jeho výstupem by měl být dle shodnosti

seřazený seznam porovnaných samostatných prací.

6.2 Zpracování zdrojového kódu

Zpracování probíhá ve stejných krocích jako u překladačů tj. zdrojový kód se zpracuje *lexikálním analyzátorem* (tokenizer), který vytvoří *lexikální elementy* (tokeny), ty jsou zpracovány v *syntaktickém analyzátoru* (parser), jehož výstupem je výsledná popisná struktura zdrojového kódu. Postup je zobrazen na obrázku 6.1.



Obrázek 6.1: Postup zpracování zdrojového kódu

6.2.1 BasicLoader

`BasicLoader` je navržen na načítání znaků ze zdrojového souboru a jejich převod do znaků vyžadovaných *lexikálním analyzátorem.*

Nejdůležitější funkcí je převod znaku odřádkování, který je na každé platformě rozdílný[1] a proto je nutné zaručit, že *lexikální analyzátor* dostane vždy jeden tvar. Ve standardu jazyka *C++* je sice řečeno, že na dané platformě je vždy jeho znak odřádkování vracen jako `'\n'`, ale problém vzniká při načítání zdrojových souborů vytvořených na jiné platformě. `BasicLoader` detekuje různé kombinace znaků a vždy je mění na znak `'\n'`. Jedinou podmínkou správného fungování je otevření zdrojového souboru v binárním módu, jelikož textový mód může na některých platformách způsobit problém s pohybem po souboru pomocí kurzoru. Patrné to bylo převážně na platformě *Windows*, která využívá pro znak ukončení sekvenci 2 znaků a při uložení pozice kurzoru před čtením a následné nastavení kurzoru po čtení způsobilo podivné chování v podobě opakovaného čtení záhadného řetězce, který nebyl přítomen ve zdrojovém kódu.

Pro správné čtení si třída ukládá informace o vstupním streamu, naposledy přečteném znaku a aktuální pozici. Taktéž podporuje bufferování, kdy se čtené znaky ukládají do bufferu a je možno pak výsledný řetězec použít pro uložení do tokenu.

6.2.2 Token

Lexikální element generovaný *lexikálním analyzátorem* (tokenizerem). Vytvořený *lexikální element* obsahuje:

- Typ co bylo právě přečteno ze zdrojového kódu (identifikátor, číslo, operátor, ...).
- Klíčové slovo. V některých případech může být identifikátor klíčovým slovem. Pro *syntaktický analyzátor* je zbytečné porovnávat sekvenci přečtených znaků se všemi klíčovými slovy, proto *lexikální analyzátor* nastavuje speciální číselnou hodnotu, která umožňuje rychlou identifikaci klíčového slova.

[1] Windows - `'\r\n'`; Linux - `'\n'`; Mac - `'\r'`

- Sekvence znaků, ze kterých se daný *lexikální element* skládá.
- Počáteční pozice *lexikálního elementu* ve zdrojovém kódu.

Jelikož je sekvence znaků reprezentujících token uložena jako řetězec, vzniká u čísel problém s převodem. Třída poskytuje funkce pro převod hodnoty na libovolný typ, pro který je definován převodník (všechny typy podporované `std::ostringstream`). Díky specializaci šablon v *C++* je možno definovat vlastní převodník.

6.2.3 BasicTokenizer

Základní implementace *lexikálního analyzátoru* se stará o generování *lexikálních elementů* ze sekvencí znaků dodané třídou `BasicLoader`. Bez implementace pro daný jazyk není třída moc použitelná, jelikož nelze předem říci jaké typy tokenů bude generovat. Základní verze je schopna generovat jen typy jako je identifikátor, celé číslo a ostatní znaky.

6.2.4 BasicParser

Obdobně jako `BasicTokenizer` neposkytuje bez implementace pro zpracovávaný zdrojový kód žádnou funkčnost. Její samostatné použití neumožňuje ani to, že se jedná o abstraktní třídu. Třída hlavně poskytuje základní funkce, které mohou být využity v jednotlivých implementacích. Mezi tyto funkce patří čtení tokenu, zahození aktuálního tokenu, testy aktuálního tokenu a vyhození syntaktické chyby. V případě chyby je schopen zobrazit část řádky, na které chybu nalezl, ale chování této funkčnosti je závislé na chování implementace, jelikož základní třída nemá možnost jak určit konec řádky. Vhodnější by bylo přenést tuto funkčnost do třídy `BasicLoader`, ale tato funkčnost byla zavedena v pozdějších fázích vývoje, proto bylo takovéto řešení snadnější.

6.2.5 BasicExpression

Tato třída je navržena k vyhodnocování výrazů na základě typu tokenů (operátorů) a pro rozdílné datové typy operandů. Třída vznikla z důvodu eliminace shodného kódu pro vyhodnocování výrazů u *ANSI C* preprocesoru a *ANSI C* parseru, kde jediný rozdíl byl v datovém typu operandů (`int` vs. `int+float`).

Třída je navržena jako šablona, kde je nutno uvést datový typ operátoru, operandu a funktor[2], který bude použit na vyhodnocování. Funktor může být typu `void`, ale pak nebude možné výraz vyhodnotit. Pokud je uveden validní funktor, tak musí být schopen vyhodnotit operace dané typem operátorů pro zadaný datový typ operandů.

Základem je seznam hodnot, které uchovávají operand nebo operátor (a zda je to operand nebo operátor). Hodnoty jsou vyhodnoceny na základě postfixové notace[3], jelikož se nejlépe programově vyhodnocuje. O vyhodnocení operandu a operátorů se stará dodaný funktor, který dostane jako parametry operátor a dva operandy, jež se na základě dodaného operátoru vyhodnotí a vrátí výsledek. V případě, že

[2]Funktor je třída nebo struktura, která se chová jako funkce. Jeho výhodou je především to, že jeho hlavní funkce lze snadno vložit na místo a není tak nutné volat funkci podle ukazatele.

[3]tj. `45+6*` pro `(4+5)*6`

je dodaný operátor unární, musí funktor nastavit speciální příznak, podle kterého vyhodnocovací funkce pozná, že nemá druhý operand zahazovat.

Pomocí této třídy nelze vyhodnocovat všechny druhy výrazů. Lze takto vyhodnotit jen konstantní výrazy, jež lze vyhodnotit v době překladu.

6.3 Popisná struktura otisku

Popis otisku zdrojových kódu je potřeba ukládat ve formě, kterou lze snadno procházet. Základem pro třídy byly entity dané samotným jazykem: projekt, unita, proměnná, funkce, struktura, výčtový typ a datový typ. Přehled závislostí tříd popisné struktury je vyobrazen na části obrázku C.2.

6.3.1 Project

Představuje celý projekt složený z více překladových jednotek. Tyto překladové jednotky lze „slinkovat“ do jedné a mít tak přehled o všech entitách v projektu nezávisle na umístění v překladové jednotce. Při „linkování“ se zahazují deklarace funkcí z jiných překladových jednotek, aby nezasahovaly do porovnávání.

Data

- seznam překladových jednotek,
- pomocný seznam výčtových typů,
- pomocný seznam struktur,
- pomocný seznam globálních proměnných,
- pomocný seznam definovaných funkcí.

6.3.2 Unit

Překladová jednotka představuje jeden zdrojový soubor, který bývá překládán překladačem do objektového souboru. Je složen ze všech základních entit, které se vyskytují ve zdrojovém kódu jednoho souboru, tj. datový typ, výčtový typ, struktura, proměnná a funkce.

Data

- seznam definovaných typů, uložených podle názvu,
- seznam definovaných výčtových typů,
- seznam definovaných struktur a unionů,
- seznam globálních proměnných,
- seznam deklarovaných a definovaných funkcí.

6.3.3 Type

Datový typ používaný proměnnými a funkcemi. Každý typ nese informaci o specifikátorech (základní datové typy), kvalifikátorech (`const` / `volatile`) a zda se jedná o výčtový typ nebo strukturu. Dalšími uchovávanými informacemi jsou ukazatele a jejich kvalifikátory a rozměry pole.

Na základě těchto informací lze určit velikost datového typu – velikosti jsou ale závislé na velikostech datových typů na aktuální platformě.

Data

- kvalifikátory typu,
- seznam kvalifikátorů ukazatelů, kdy jedna hodnota znamená jednu úroveň ukazatele,
- specifikátory typu - určují základní datové typy (viz 3.3.1),
- ukazatel na strukturu - je-li typ definován jako struktura,
- ukazatel na výčtový typ - je-li typ definován jako výčtový typ,
- seznam rozměrů pole - počet hodnot odpovídá počtu dimenzí.

6.3.4 Enum

Výčtový typ sloužící k definici celočíselných konstant. Konstanty jsou ukládány pod názvem s definovanou hodnotou. Třída se chová přesně dle definice standardu (viz 3.3.1.3), kdy nově vložená konstanta bez hodnoty, dostane hodnotu o 1 větší než předchozí konstanta. Pokud se jedná o první konstantu, pak dostane hodnotu `0`.

Data

- název výčtového typu,
- seznam konstant ukládaných podle názvu,
- hodnota naposledy vložené konstanty.

6.3.5 Structure

Struktura je jediná možnost jak skládat více datových typů do jednoho. Je složena z členských proměnných (viz 6.3.6).

Specialitou je možnost nastavení, zda se jedná o `struct` nebo o `union`. Z hlediska popisné struktury mezi nimi není rozdíl. Rozdíl je patrný až při výpočtu velikosti a používáním proměnné, kdy u `unionu` se všechny členské proměnné překrývají (tj. zabírají stejné paměťové místo).

Data

- název struktury,
- příznak `struct` nebo `union`,
- seznam členských proměnných.

6.3.6 Member

Rozšíření třídy `Variable` o tzv. fieldset, který udává kolik bitů zabírá daná členská proměnná ve struktuře.

Data

- všechna data shodná s třídou `Variable`,
- velikost bitového pole.

6.3.7 Variable

Popisuje proměnné uvedené ve zdrojovém kódu. Používá se i pro popis parametrů funkce. Je složená jen z názvu, datového typu a typu uložení - automatický, registr, statický apod.

Data

- název proměnné,
- specifikátor uložení,
- datový typ.

6.3.8 Function

Funkce je nejsložitější entita ze všech, jelikož se skládá z návratového typu, parametrů a těla funkce. Tělo je složeno z konstrukcí (viz 6.3.9).

Data

- název funkce,
- specifikátor uložení,
- návratový typ,
- seznam parametrů,
- zda má proměnný počet parametrů,
- nepovinné tělo funkce.

6.3.9 Statement

Základní kámen popisu těla funkce. Pro specifické části používá specializované potomky, kterými jsou:

- compound - blok konstrukcí,
- selection - `if`, `switch`,
- iteration - `for`, `while`, `do-while`,
- labeled - návěstí pro switch a nebo `goto`,
- expression - výrazy, volání funkcí,
- jump - `goto`, `break`, `return`.

Díky těmto třídám lze popsat celé tělo funkce. Popis má tvar stromu, kde *compound-statement* je hlavní kořen a ostatní typy jsou listy, pokud nemají vnořenou konstrukci.

Pro rychlejší práci je použit výčtový typ `StatementType`, jež umožňuje rychlé rozpoznání typu konstrukce. Jedná se o mnohem rychlejší způsob než volat pro každý typ `dynamic_cast` a testovat, zda není `NULL`[4]. V případě výčtového typu se jedná o porovnání jedné hodnoty, zatímco v případě `dynamic_cast` může jít o porovnání názvu třídy a jiných operacích[5].

Některé specializace umožňují spočítat celkový počet deklarací, cyklů, výrazů a podmínek, jež jsou v ní obsaženy.

6.3.10 Declaration

Speciální třída, která spojuje deklarace proměnných a funkcí. Jediný rozdíl mezi deklarací proměnné a deklarací funkce je uvedení seznamu parametrů pro deklaraci funkce. Ve výsledku lze převést na proměnnou nebo funkci.

Data

- kvalifikátory typu,
- seznam kvalifikátorů ukazatelů, kdy jedna hodnota znamená jednu úroveň ukazatele,
- specifikátory typu - určují základní datové typy (viz 3.3.1),
- ukazatel na strukturu - je-li typ definován jako struktura,
- ukazatel na výčtový typ - je-li typ definován jako výčtový typ,
- seznam rozměrů pole - počet hodnot odpovídá počtu dimenzí,
- název deklarátoru,
- zda se jedná o funkci,

[4] C++98 nepodporuje `NULL` pro nulové ukazatele. Místo toho se má používat hodnota `0`.
[5] Přesná implementace *RTTI* (a tedy i `dynamic_cast`) je závislá na překladači.

- seznam parametrů,
- zda má proměnný počet parametrů,
- vnořená deklarace.

6.3.11 Expression

Jednoduchá třída pro uložení výrazu, který se vyskytuje uvnitř těla funkce. Výraz neukládá v žádné použitelnější formě a slouží jen k uložení tokenů, ze kterých se výraz skládá. Nelze tedy použít na vytvoření stromu volání funkcí.

Data

- seznam tokenů výrazu.

6.4 ANSI C

Zde je popsána implementace zpracování zdrojového kódu pro programovací jazyk *ANSI C*. Většina tříd je rozšířením základních tříd popsaných v sekci 6.2.

6.4.1 Loader

Rozšíření třídy `BasicLoader` o zpracování tzv. *trigraphů*. Jedná se o speciální sekvence 3 znaků, které jsou ve výsledku nahrazeny jedním znakem. Přehled všech trigraphů je v tabulce 3.1 uvedené v kapitole 3 popisující jazyk *ANSI C*.

6.4.2 Tokenizer

Vylepšení základní třídy `BasicTokenizer` (viz 6.2.3) o generování tokenů definovaných standardem. Jeho výstupem jsou tokeny patřící jak preprocesoru, tak parseru. Standard [1] sice uvádí, že tokenizer nejdřív vytvoří tokeny pro preprocesor, které jsou následně převedeny na tokeny parseru. Přesné dodržení tohoto chování by bylo složitější, proto tokenizer vytváří oboje druhy tokenů najednou a nepřevádí je.

Tokenizer generuje tokeny typu:

- odřádkování,
- mezera,
- identifikátor,
- celé číslo,
- číslo s plovoucí čárkou,
- řetězec,
- znak,
- operátory (více druhů),

- závorky.

Odřádkování a mezera nejsou pro parser důležité. Pro preprocesor nejsou důležité zase čísla s plovoucí čárkou (pracuje jen s celými čísly).

6.4.3 Preprocesor

Preprocesor byl navržen, aby se choval jako *ANSI C* tokenizer a pracoval tak na principu *text-to-token*. Při čtení tokenů vnitřně zpracovává direktivy a nahrazuje tokeny, které odpovídají makrům. Náhrady si ukládá do speciálního bufferu, ze kterého vrací tokeny dříve než začne číst dále ze vstupního souboru.

Celý cyklus zpracování zdrojových tokenů na výsledné tokeny je vyobrazen na obrázku C.3. Při požadavku o načtení dalšího tokenu dojde ke kontrole, zda se aktuálně nenachází zpracování zdrojového souboru ve `false` větvi `#if-#else` bloku. Pokud ano, jsou zahozeny veškeré další tokeny, dokud není nalezena direktiva, která opustí blok a nebo změní jeho stav na `true`. Lze-li tokeny vracet, načte se token, podle kterého se následně provedou akce:

- *identifikátor* – Pokus o náhradu makrem – Pokud se jedná o makro, tak se zahodí token a případné argumenty makra.
- *hash (#)* – Zpracování direktivy – zahození tokenů direktivy.
- *mezera* – Zahození tokenu, pokud není nastaven příznak vracení mezer.
- *komentář* – Zahození tokenu, pokud není nastaven příznak vracení komentářů.
- *konec řádky* – Zahození tokenu, pokud není nastaven příznak vracení konců řádek.
- *C++ komentář* – Zahození tokenu – není součástí standardu.

V ostatních případech je čtení ukončeno a vrácen přečtený token.

6.4.3.1 Zpracování podmínek

Jelikož preprocesor funguje stejně jako tokenizer na principu postupného čtení tokenů, je nutné si nějak zachovávat aktuální stav v jednotlivých podmínkách včetně zanoření. Pro každý podmínkový blok je vytvořena speciální proměnná (třída `Condition`), která uchovává informace o výsledku výrazu bloku a ukládá se do zásobníku (a při opouštění z bloku se zahazuje). Kvůli více-podmínkovým blokům nelze použít jedinou hodnotu typu `bool`, protože v ní nejsme schopni udržet informaci o tom, zda nějaká z předchozích částí bloku nebyla vyhodnocena jako `true` a tedy další už nesmí být označeny jako `true`. O to se stará druhá hodnota typu `bool`.

Pokud je v takto vytvořeném zásobníku nějaká hodnota označena jako `false`, načítaný token nepatří do výsledného zdrojového kódu. Bohužel se nepodařilo algoritmus upravit tak, aby nebylo nutné při každém čtení tokenů procházet celý zásobník podmínek a dostupnost ověřit jen hodnotou na vrchu zásobníku. Při pokusech se vždy objevil problém se špatnou funkčností.

Samotné výrazy podmínek jsou vyhodnocovány pomocí třídy `BasicExpression` (6.2.5) a její verzí pro datový typ `int`.

6.4.3.2 Definice maker

Zpracování definice *object-like* je jednoduchá, jelikož se jedná jen o název a tělo makra, kterým bude makro nahrazeno.

Definice *function-like* makra obsahuje navíc seznam parametrů, obdobně jako má funkce (od toho ten název). Seznam parametrů je nutné zpracovat a uložit do speciálního seznamu. Na rozdíl od klasické funkce, tady není potřeba uvádět typ a stačí jen uložit název, za který se bude argument při volání nahrazovat.

Redefinice maker Standard [1] umožňuje redefinici makra, pokud se jedná o stejnou definici nezávislou na počtu a velikosti mezer. Implementace s tímto počítá a dovoluje takto definované redefinice.

6.4.3.3 Náhrada maker

Náhrada *object-like* makra je jednoduchá, jen se zahodí původní identifikátor a náhrada se uloží do výstupního bufferu.

U *function-like* maker je to složitější, jelikož argumenty se používají v nahrazené formě a to jen v případě, že parametr není použit ve spojení s operátory `#` a `##`. Nejdříve je vždy nutno zpracovat argumenty, které jsou odděleny čárkou, ale vzniká zde problém s platností jednotlivých čárek. Čárka u vnořeného makra nemůže být použita jako oddělovač daného makra, proto je nutno počítat se zanořením závorek.

V případě náhrady maker vzniká problém s rekurzivní nebo cyklickou náhradou maker, kdy stejné makro je stále nahrazováno a může způsobit i *livelock*. Dle standardu [1] se uvnitř náhrady makra nemůže samotné makro nahradit znovu. Implementace preprocesoru řeší tento problém ukládáním seznamu nahrazených maker do aktuálního výstupního bufferu a při jeho vyprázdnění, povoluje uvedená makra. Tím je zaručeno, že v těle makra nemůže být nahrazeno samo nahrazované makro a to i při větším zanoření. Složitější problém vzniká u nahrazování *function-like* maker, kde je makro uvedeno jako argument a následně i v těle nahrazovaného *function-like* makra. Dle standardu [1] nesmí být v místě použití argumentu makro nahrazeno a ve zbytku těla nahrazeno být může. Vzhledem k návrhu implementace preprocesoru je problém toto chování správně implementovat[6]. Všechno popsané chování je uvedeno ve fragmentu kódu 6.1, kde je vždy v komentáři uveden výsledek po průchodu preprocesoru.

6.4.3.4 Chyby ve zdrojovém kódu

Syntaxe kódu preprocesoru není tak složitá a nemůže se v ní vyskytnout mnoho chyb. Program kontroluje jen správnost podmínkových bloků, zda jsou správně ukončeny. U definice *function-like* makra kontroluje, zda není definována s duplicitním názvem parametru. Těla maker taktéž nesmí obsahovat na začátku nebo na konci operátor `##`.

[6] Např. TinyCC tento problém vůbec neřeší a klidně v těle makra nahradí makro nahrazené v argumentu makra.

Zdrojový kód 6.1: Příklad cyklického nahrazování maker

```
1  #define X Y
2  #define Y X
3  #define f(x) f(x+1)
4  #define z z[1]
5
6  X    /* -> X */
7  f(2) /* -> f(2+1) */
8  z    /* -> z[1] */
9  f(z) /* -> f(z[1]+1) */
10      /* ne f(z[1][1]+1) */
```

6.4.3.5 Rychlost

Zpracování zdrojového kódu preprocesorem je náročná věc a je důležitá taktéž jeho rychlost.

Skript použitý pro měření časů je uveden ve výpisu A.1.

Preprocesory Pro test rychlosti byly použity následující preprocesory:

- *GCC* - gcc (Ubuntu/Linaro 4.4.4-14ubuntu5.1) 4.4.5.
- *Clang* - clang version 2.8 (branches/release_28).
- *CC* - Vlastní implementace preprocesoru.

Do testu nebyl zařazen preprocesor překladače *TinyCC*, který by bylo nutné upravit, aby splňoval všechny podmínku testu.

Podmínky testu

- Použití systémových hlavičkových souborů.
- Zpracování překladače *TinyCC 0.9.25*. Překladač je napsán v jazyce *C* a jedná se o dostatečně velký projekt na otestování rychlosti a navíc je navržen tak, že stačí překládat jediný soubor, který si spojuje všechno a ve výsledku není nutné linkování objektových souborů.
- Použití implicitního nastavení překladače tj. nepoužití žádných dalších přepínačů kromě `-E` pro preprocesor.
- 10 průchodů a výsledný čas je určen jejich průměrem.
- Žádné náročné aplikace běžící na pozadí v průběhu testu.

Testovací stroj Jelikož jsou testované překladače vyvinuté hlavně na *Unix*, je tedy jasné, že by testování mělo být provedeno právě na *Unixu*/*Linuxu*.

Testovacím strojem byl:

- virtualizované Ubuntu 10.10 x64 (VirtualBox 4.1.10),

Tabulka 6.1: Porovnání rychlostí preprocesorů

Překladač	Rychlost
GCC 4.4	0,14 s
Clang 2.8	0,435 s
CC	0,654 s

- dvě jádra procesoru Intel Q6600 na frekvenci 2,40 GHz,
- 2,0 GiB RAM.

Výsledky testu Dle tabulky 6.1 lze poznat, že vlastní implementace preprocesoru na tom není rychlostně výrazně špatně. V porovnání s *GCC* je poznat vyspělost roky vyvíjeného překladače a rozdíl mezi rychlostí *C* a *C++*. *Clang*u hraje do karet cachování vstupního souboru, kdy je další zpracování o poznání rychlejší. Vlastní implementaci by v rychlosti pomohlo použití rychlejšího načítání znaků ze vstupního souboru[7].

Celá tabulka rychlostí jednotlivých průchodů je v příloze B.2. V uvedené tabulce je poznat, že *Clang* je při prvním průchodu výrazně pomalejší než ostatní.

6.4.4 Parser

Parser je založen na Backus-Naurově formě definované ve standardu. Pro každý neterminál je vytvořena funkce, která se chová přesně podle daných pravidel *BNF*. Jelikož *BNF* definovaná ve standardu [1] je nedeterministická a nelze jí přímo použít pro parser, bylo nutné provést některé úpravy, které nedeterministické části eliminovali.

6.4.4.1 Výrazy

Většina částí výrazů má shodný kód a je zbytečné je mít na několika místech, což ztěžuje případné úpravy. Tyto části fungují na stejném principu, kdy volají nějakou funkci, po jejím návratu testují aktuální token na operátor a na základě typu operátoru volání funkce opakují a nebo se ukončí. Zde je využita síla *C++* šablon, kdy se z jednoho kódu vygeneruje několik verzí na základě hodnoty parametru šablony. Šabloně se zadávají parametry pro typ volané funkce a speciální třídy, podle které se rozhodne, zda je aktuální token požadovaný operátor. Jelikož některé výrazy mohou mít na místě operátoru více druhů operátorů, je nutná možnost specifikovat více operátorů. O to se stará rekurzivní[8] šablona `comparator`. Příklad použití této šablony je vidět na fragmentu kódu 6.2. Parametry šablony nejsou explicitně uváděny, jelikož je dovede překladač poznat sám podle zadaných argumentů. Celý kód použité šablony je uveden ve výpisu kódu A.2.

Zdrojový kód 6.2: Využití šablony pro část výrazu

```
1 // process_shift_expression
```

[7] Využívá třídy `std::istream`, která je pomalejší kvůli různým funkčnostem.

[8] Rekurzivní zde znamená, že využívá sebe sama pro vytvoření složitějšího objektu a rekurzi ukončuje svojí specializací.

```
2 process_simple_expression(expr, &Parser::
     process_additive_expression,          comparator<
     TOKEN_TYPE_LSHIFT, comparator<TOKEN_TYPE_RSHIFT> >());
3
4 // process_logical_or_expression
5 process_simple_expression(expr, &Parser::
     process_logical_and_expression,          comparator<
     TOKEN_TYPE_OR_OR>());
```

6.4.4.2 Vyhodnocování konstantních výrazů

Syntaxe jazyka definuje konstantní výrazy, což jsou výrazy vyhodnotitelné v době překladu. Konstantní výraz může obsahovat jen konstanty (včetně konstant daných výčtovým typem) a operátory. Pro vyhodnocení je využívána šablona `BasicExpression` (viz 6.2.5), které je dodána speciální třída s názvem `IntFloat`. Tato třída je schopna uložit hodnotu `int` nebo `float` (typ si ukládá pomocí příznaku) a na základě toho provádět operace pomocí přetížených operátorů. Z vnějšího pohledu tato třída vypadá jako primitivní datový typ (`int` nebo `float`), jen na pozadí se provádí složitější operace.

6.4.4.3 Chyby ve zdrojovém kódu

Parser téměř přesně dodržuje syntaxi jazyka a v případě chyby ukončuje svůj běh s popisem dané chyby a místem, kde se chyba vyskytla. Místo chyby je schopen označit jen v případech, kdy je chyba dána aktuálním (chybným) tokenem. Na rozdíl od klasických překladačů není schopen obnovy po chybě a pokračování ve zpracovávání zbytku zdrojového kódu.

V případě chyby informuje aplikace ve tvaru inspirovaném překladačem *GCC* a *Clang*.

Parser dovoluje v kódu několik výjimek mimo standard, jelikož jsou definovány v novějších verzích standardu jazyka *C* a překladače je v implicitním nastavení podporují[9]:

- *C++* řádkový komentář,
- deklarace proměnné/funkce v tělu funkce mimo začátek bloku.

Přehled reakcí na některé syntaktické chyby ve zdrojovém kódu je uveden v příloze D.

6.5 INI

Pro potřeby porovnávání je potřeba možnosti upravování parametrů. Vzhledem k jejich velkému množství nepřipadá v úvahu zadávání jako argumentů při volání nástroje. Nejvhodnějším řešením je uložení do konfiguračního souboru ve formátu *INI* (tj. název = hodnota). Vzhledem k jednoduchosti syntaxe *INI* souborů a podpoře

[9] Odevzdané samostatné práce většinou tyto výjimky využívají a metrická analýza by tak selhala na těchto výjimkách.

Zdrojový kód 6.3: Příklad složeného ukládání

```
1 void Output::write(const desc::Variable& variable)
2 {
3     // Zapíšeme název proměnné
4     write(variable.name);
5     // Uložíme název souboru
6     write(variable.filename);
7     // Typ uložení
8     write(variable.storage);
9     // Typ proměnné
10    write(variable.type);
11 }
```

pro zpracování zdrojových textů přímo ve vytvořené knihovně, bylo zbytečné volit externí knihovnu pro práci s *INI* soubory.

Jelikož syntaxe *INI* souboru není standardizována a každý používá trochu rozdílnou verzi, základ zůstává stejný. Základ je převzat z popisu formátu na wikipedii [11], jen speciality nejsou podporovány. Pro potřeby aplikací je potřeba jen načtení klíče a hodnoty v určité sekci.

6.5.1 File

Třída popisující jeden *INI* soubor. Skládá se ze seznamu sekcí přístupné přes název sekce. Pokud není explicitně ve zdrojovém *INI* souboru uvedena sekce, je vytvořena implicitní.

6.5.2 Section

Ukládá hodnoty zapsané v *INI* souboru ve formátu název - hodnota. Načtené hodnoty lze pomocí speciálních funkcí převést na ostatní typy jako je `int`, `float` apod.

6.6 Ukládání a načítání otisku

Po vytvoření otisku zdrojového kódu v operační paměti, je nutné tento otisk uložit do souboru pro další použití. Bez toho by bylo nutné pokaždé zpracovat celý zdrojový kód, což může být ve výsledku časově náročné.

Popis formátu souboru, do kterého se otisk ukládá, je uveden v příloze E. Jedná se o binární formát, kde má každá část otisku předem definované místo.

Ukládání i načítání je postaveno na základních datových typech, pro které jsou definované specifické funkce, jež zaručují přesnou velikost i typ uložení napříč platformami. Z těchto funkcí jsou pak složeny ostatní funkce pro složitější datové typy. Na výpisu 6.3 je vidět, jak funguje zapisování jednotlivých částí třídy `desc::Variable`, kde pro každou část instance je zapsána hodnota pomocí přetížené funkce `write`.

Obdobným způsobem je řešeno načítání takto uloženého souboru, jen s tím rozdílem, že dochází k uložení načtené hodnoty do proměnné.

Tabulka 6.2: Porovnání množin – verze první

A	B	Shodnost
a_1	b_1	100 %
a_1	b_2	30 %
a_2	b_1	60 %
a_2	b_2	100 %
		72,5 %

6.7 Porovnání otisků

Porovnávání zdrojových kódů je založeno na porovnávání jednotlivých metrik částí zdrojového kódu. Porovnávanými částmi jsou proměnné, funkce, struktury apod. Systém je založen na porovnávání základních typů, ze kterých se skládají ostatní části. V případě proměnné se porovnává název a typ a v případě překladové jednotky se porovnávají proměnné, funkce, struktury a výčtové typy.

6.7.1 Výpočet shodnosti

Jednotlivé atributy částí mají dané procentuální zastoupení v dané části pomocí distribuční funkce (daná konfiguračním souborem s nastavenými parametry). Výsledná shodnost dané části je dána vzorcem:

$$S_i = \sum_{j=1}^{n} d_j \cdot S_j;\ S_i, S_j \in [0,1] \tag{6.1}$$

kde n je počet atributů dané části, S_i je shoda dané části, S_j je shoda atributu a d_j je procentní zastoupení atributu ve výsledné shodě.

6.7.2 Porovnání množin

Při porovnání množin (např. funkcí) se porovnává vždy každý prvek z první množiny s každým prvkem z množiny druhé. Tím vzniká $n = N \cdot M$ porovnávání a při výpočtu shodnosti na základě vzorce 6.1 (kde $d_j = \frac{1}{n}$) by nikdy nemohly být shodné funkce označeny za shodné z důvodu ovlivnění výsledku porovnáním s ostatními prvky. Příklad pro množiny velikosti 2 je uveden v tabulce 6.2, kde obě množiny obsahují shodné prvky, ale výsledek porovnání tomu neodpovídá.

Proto vznikl nový výpočet pro množiny, kde se pro každý prvek z první množiny vybere prvek z množiny druhé s největší shodností. Takto vzniklá množina je použita ve vzorci 6.1. V tabulce 6.3 je uveden přehled shodností pro nový vzorec, kdy je nižší hodnota shodnosti pro daný prvek z první množiny přeškrtnuta a ve výpočtu ignorována.

$$S_i = \frac{1}{N} \sum_{j=1}^{N} max\left(\{S_{j1},\, S_{j2},\, \dots S_{jM}\}\right);\ S_i,\, S_{j1},\, S_{j2},\, \dots S_{jM} \in [0,1] \tag{6.2}$$

kde N je počet prvků první množiny, M je počet prvků množiny druhé, S_i je shodnost množin a S_{j1} až S_{jM} je shodnost prvku množiny druhé s daným prvkem množiny první.

Tabulka 6.3: Porovnání množin – verze druhá

A	B	Shodnost
a_1	b_1	100 %
~~a_1~~	~~b_2~~	~~30 %~~
~~a_2~~	~~b_1~~	~~60 %~~
a_2	b_2	100 %
		100 %

6.7.3 Porovnávané metriky

6.7.3.1 Projekt

Projekt obsahuje jen překladové jednotky. Vzhledem k tomu, že můžou být shodné části v rámci projektu rozděleny do jiných překladových jednotek, neporovnávají se samotné překladové jednotky, ale rovnou jejich obsah, který se pro účely testování převede do speciálních proměnných[10].

Porovnávané metriky

- obsah překladových jednotek (tj. proměnné, výčtové typy, struktury a funkce).

6.7.3.2 Překladová jednotka

Pro porovnání překladových jednotek jsou důležité hlavně funkce a proměnné. Výčtové typy a struktury jsou důležité až při samotném použití, stejně jako definované datové typy.

Porovnávané metriky

- proměnné,
- výčtové typy,
- struktury,
- funkce.

6.7.3.3 Výčtové typy

Porovnávání výčtových typů může být složité, protože lze jeden výčtový typ snadno rozdělit do více výčtových typů bez velké změny ve zbytku zdrojového kódu. Je to dáno tím, že konstanty mají globální platnost a není potřeba uvádět název výčtového typu.

[10]Převádějí se jen ukazatele, takže objekty zůstávají stále ve vlastnictví daných překladových jednotek.

Porovnávané metriky

- název,
- počet konstant,
- hodnoty konstant.

6.7.3.4 Datové typy

Nejvíce testovaná část, jelikož se vyskytuje skoro všude. Datový typ je ukládán v „rozbalené“ formě, proto definice vlastních typů výsledek porovnání neovlivní.

Porovnávané metriky

- specifikátory - brán ohled na implicitní specifikátor `int`,
- počet ukazatelů - kvalifikátory nejsou v tomto kontextu důležité,
- použitý výčtový typ - není-li uveden, porovnává se jen uvedení i v druhém datovém typu,
- použitá struktura - není-li uvedena, porovnává se jen uvedení i v druhém datovém typu,
- velikost pole (celkový počet prvků, nezávisle na dimenzích).

6.7.3.5 Struktury

Při porovnávání struktur hrají důležitou roli členské proměnné, ze kterých se struktura skládá. To ovlivňuje i velikost výsledné struktury.

Porovnávané metriky

- název,
- počet členských proměnných,
- velikost struktury,
- členské proměnné.

6.7.3.6 Členské proměnné

Jedná se jen o rozšíření porovnávání proměnných o bitové pole, které lze u členských proměnných struktur uvést. Z hlediska porovnání nemá bitové pole velký význam.

Porovnávané metriky

- proměnná,
- bitové pole.

6.7.3.7 Proměnné

U proměnných je nejdůležitější datový typ, protože název lze snadno změnit.

Porovnávané metriky

- název,
- datový typ.

6.7.3.8 Funkce

Funkce mají nejvíc metrik, které lze porovnat. Největší význam v porovnání má samotné tělo funkce, jež, pokud je rozsáhlé, vypovídá o shodnosti funkcí nejvíce. Algoritmus, který funkce implementuje, nelze ve většině případů přespat tak, aby měl jiný počet cyklů.

Porovnávané metriky

- název,
- návratový typ,
- počet parametrů,
- parametry,
- zda má proměnný počet parametrů,
- počet deklarací v těle funkce,
- počet konstrukcí cyklů,
- počet konstrukcí podmínek,
- počet výrazů.

6.7.4 Příklady

Níže je uvedeno několik porovnávání podobných zdrojových kódů spolu s výsledky. Výsledky jednotlivých příkladů porovnávání jsou uvedeny v tabulce 6.4. Samotné výsledky jsou ovlivněné nastavením parametrů porovnávání a při jejich změně mohou být výsledky rozdílné.

V příkladech nejsou uváděny direktivy a makra preprocesoru, jelikož ty se do samotného otisku zdrojového kódu nedostanou a v těchto příkladech by jen zbytečně překážely.

6.7.4.1 Datové typy

Následující kusy kódu demonstrují pokus o záměnu datových typů pomocí definice vlastních typů. Běžnou kontrolou lze tyto záměny snadno přehlédnout. Oba kusy kódu jsou ve výsledku totožné, proto by výsledek porovnávání měl označit tyto kódy za totožné.

Zdrojový kód 6.4: První kus kódu

```
1 int a;
2 float data[50];
3 struct { int x, y; } points[5];
```

Zdrojový kód 6.5: Druhý kus kódu

```
1 typedef int int2;
2 typedef float float2;
3 typedef int2 int3;
4 typedef float2 float3;
5 typedef struct { int x, y; } point;
6 int3 a;
7 float3 data[50];
8 point points[5];
```

6.7.4.2 Pole

Uvedené kusy kódu jsou téměř totožné, jen rozměry polí jsou uvedeny trochu jinak. Pole ve výsledku zabírají stejné místo v paměti a jediný rozdíl mezi nimi je v přístupu k těmto polím. Výsledek porovnání by měl označit tyto kusy kódu za totožné.

Tento typ úpravy nebude moc používaný, protože by bylo nutné upravit i kód pro práci s daným polem.

Zdrojový kód 6.6: První kus kódu

```
1 float data[50];
2 struct { float a, b; } list[1000];
```

Zdrojový kód 6.7: Druhý kus kódu

```
1 float data[5][10];
2 struct { float a, b; } list[10][10][10];
```

6.7.4.3 Záměna názvů

Asi nejčastěji používaný způsob zakrývání shodných zdrojových kódů. V uvedených kusech kódu došlo jen ke změně některých názvů proměnných a názvu funkce. Tyto kódy by měly být označeny za shodné. Výsledek nejvíce ovlivní uvedená funkce, jelikož obsahuje nejvíce metrik, které lze porovnat a většina jich je v uvedených kusech kódu shodná. Pokud by nebyla uvedena funkce, shoda by byla mnohem menší.

Zdrojový kód 6.8: První kus kódu

```
1 int array[30];
2 int* ptr;
3 int index;
4 void next(void) { ptr++; index++; }
```

Zdrojový kód 6.9: Druhý kus kódu

```
1 int pole[30];
2 int* ukaz;
3 int index;
4 void dalsi(void) { ukaz++; index++; }
```

6.7.4.4 Úprava funkce

Funkce se upravují taktéž velice často, ale jen určité části, které člověk při běžném čtení snadno přehlédne. Následující kusy kódu představují funkci na výpočet sumy hodnot v zadaném poli. Každý používá jiný typ výpočtu, ale výsledek je shodný. V tomto případě je problém označit kódy za shodné a člověk by na první pohled nepoznal, že funkce dělají totéž (tedy kromě hlaviček funkcí, které jsou shodné). Jediným použitelnějším ukazatelem shodnosti je počet cyklů.

Zdrojový kód 6.10: První kus kódu

```
1 float sum(const float* data, unsigned int size) {
2     int i;
3     float sum = 0;
4     for (i = 0; i < size; ++i) {
5         sum += data[i];
6     }
7     return sum;
8 }
```

Zdrojový kód 6.11: Druhý kus kódu

```
1 float sum(const float* data, unsigned int size) {
2     float sum = 0;
3     while (size-- > 0) {
4         sum += *data;
5         data++;
6     }
7     return sum;
8 }
```

6.7.4.5 Rozdílné programy

Důležitou informací je to, jak reaguje porovnávání na dva rozdílné programy. Nelze předpokládat, že by jejich shodu označil jako 0%, jelikož vždy se naleznou shodné části. I když jsou následující kódy velice rozdílné, jsou zde určité metriky, které jsou pro jednotlivé části kódu shodné nebo podobné. Při porovnávání hraje velkou roli

Tabulka 6.4: Výsledky porovnávání

Příklad	Shoda
Datové typy	100 %
Pole	100 %
Záměna názvů	92,049 %
Úprava funkce	74,49 %
Rozdílné programy	55,085 %

množství porovnávaného kódu, kdy v menších programech se snáze nalezne falešná shoda než ve velkých.

Zdrojový kód 6.12: První kus kódu

```
int calculate(int a, int b) {
    int i, res = 0;
    for (i = 0; i < a * b; ++i) res += (a + b * b) * i;
    return res;
}
main(void) {
    int a, b;
    scanf("%d\n", &a);
    scanf("%d\n", &b);
    printf("Result: %d\n", calculate(a, b));
    return 0;
}
```

Zdrojový kód 6.13: Druhý kus kódu

```
float data[1000];
void init(void) {
    int i;
    for (i = 0; i < 1000; ++i) data[i] = (i * i) * 0.25f;
}
float find(int id) {
    return data[id];
}
main(int argc, char** argv) {
    init();
    printf("Result: %f\n", find(atoi(argv[1])));
    return 0;
}
```

6.7.4.6 Výsledky porovnání

Uvedené příklady zdrojových kódů představují některé pokusy, které mohou být použity pro oklamání porovnávacího nástroje. Některé z nich porovnání neovlivní a některé výrazně mohou. Přehled výsledků jednotlivých příkladů je uveden v tabulce 6.4.

Tabulka 6.5: Rychlost porovnávání

Počet prací	Rychlost
5	0,076 s
50	0,286 s
200	0,858 s

6.7.4.7 Rychlost

Jedním z faktorů porovnávání je taktéž rychlost. Nelze očekávat, že porovnávání odevzdávané práce bude trvat několik minut. Tabulka 6.5 uvádí rychlost porovnávání pro jednotlivé počty prací. Časy jsou jen orientační, jelikož pro test bylo použito jen 5 rozdílných prací, které byly vždy načteny znovu. Naměřené hodnoty slouží jen pro obecnou představu, jak rychle může nástroj pro porovnávání pracovat.

6.8 Možnosti vylepšení

Nástroje vytvořené v rámci této práce nejsou dokonalé a daly by se vylepšit.

6.8.1 Porovnávání

Porovnávání sice dovede přibližně odhalit podobné zdrojové kódy, ale není schopen správně reagovat na jisté úpravy zdrojových kódů, což snižuje úspěšnost odhalení shodné samostatné práce.

6.8.2 Podpora pro další programovací jazyky

Knihovna pro práci se zdrojovým kódem byla navržena, aby mohla být rozšířena o podporu pro další programovací jazyky. Jen popisná struktura otisku odpovídá převážně požadavkům jazyka *ANSI C* a bylo by jí nutné přepracovat, aby byla schopna uložit metriky uvedené ve zdrojových kódech napsaných v jiných programovacích jazycích.

6.8.3 Popisná struktura otisku

Současná verze popisné struktury otisku neumí správně uložit výrazy a není tak možné určit volání funkcí. Pokud by tyto informace popisná struktura ukládala, bylo by možné vytvořit možný strom volání funkcí a ten použít pro porovnávání v rámci celé porovnávané aplikace.

Jak je v předchozí sekci řečeno, pro potřeby dalších programovacích jazyků by bylo nutné vylepšit popisnou strukturu do obecnější formy (případně udělat specializované verze se společným základem).

6.8.4 Přepracování analytické části

Analytická část je napsána velice obecně. Přepracování základních tříd této části do podoby *C++* šablon by přineslo lepší typovou kontrolu objektů této části a možnost

specializace pro požadavky tokenizeru a parseru zpracovávaného programovacího jazyka.

6.8.5 Interpret

Vytvořením interpretu by bylo možné simulovat chování aplikací při stejném vstupu a následně porovnat stromy volání funkcí a postup konstrukcemi. Tato simulace by přinesla více informací o chování aplikace více než statická analýza. Pro správnou funkčnost interpretu by bylo nutné ukládat kompletní popis zdrojového kódu, což současná verze nedovoluje.

Kapitola 7

Závěr

Tato práce byla koncipována jako experiment, zda je možné vytvořit automatické nástroje na objektivní vyhodnocování podobnosti zdrojových kódů pomocí metrické analýzy. Dle výsledků je jasné, že to možné je a lze tak díky výstupu nástrojů označit zdrojové kódy samostatné práce jako podobné a usnadnit tak práci člověku při hledání podobných prací. Původní představou bylo, že porovnávání bude schopno dosáhnout lepších výsledků, ale kvůli některým složitostem v jazyce *ANSI C* bylo nutné porovnání některých metrik vynechat.

Výsledné porovnávání není dokonalé a není schopno reagovat na větší úpravy v původním zdrojovém kódu, jako jsou úpravy v chování funkcí a dalších částech programu. Určitě by bylo vhodné porovnávání vylepšit, aby lépe reagovalo na často upravované části pro zamaskování původních zdrojových kódů. Na druhou stranu je relativně dobře schopno upozornit obsluhu jakou shodu mají zdrojové kódy samostatné práce s ostatními zdrojovými kódy již odevzdaných samostatných prací.

Vytvořený nástroj na metrickou analýzu zdrojového kódu je schopen zpracovat většinu zdrojových kódů a výsledek uložit do otisku. Ukládané metriky jsou však dány omezením datové struktury otisku. Program reaguje na syntakticky chybné zdrojové kódy stejně jako většina klasických překladačů, jen je pro některé části kódu přísnější, protože se snaží dodržovat *ANSI C* standard, kdežto chování klasických překladačů je ovlivněno novějšími verzemi jazyka a ostatními podporovanými programovacími jazyky.

Vytvořený datový formát pro uložení otisku není dokonalý a je schopen uchovat jen část metrik, které lze získat ze zdrojových kódů (dáno datovou strukturou, do které se metriky ukládají). Především není schopen uchovat výrazy v použitelnější podobě, proto nemůže porovnávání využít porovnání stromů volání jednotlivých funkcí. Také nemůže kvůli tomu vyhodnotit výraz uvedený v podmínkách a cyklech. Je to dáno tím, že výrazy v jazyce *ANSI C* jsou složité na uložení.

Pro práce obdobného charakteru lze doporučit lepší analýzu metrik, které lze získat ze zdrojových kódů a vytvoření odpovídající datové struktury pro jejich uložení. Správně uložené metriky (a v dostatečném množství) lze lépe použít pro porovnávání zdrojových kódů a lépe tak rozpoznat upravený zdrojový kód.

Rejstřík

ANSI C, 7, 11, 17, 21, 24–29, 31, 33, 38, 39, 52, 54
API, 20
AST, 9, 10

BNF, 42
Boost, 29

C, 11, 21, 27, 28, 41–43
C#, 28
C++, 27–29, 32, 33, 39, 42, 43, 52
C++11, 28
C++98, 28
C11, 22
C89, 11, 21
C90, 11, 21
C94, 21
C99, 21, 29
CC, 41
Clang, 28, 41–43

Endianness, 67

Garbage Collector, 27
GCC, 28, 41–43

INI, 43, 44
IRL, 10
ISO, 11, 21, 28

Java, 27
JIT, 28

K&R C, 11, 18, 21

Lex, 28
Linux, 28, 29, 41

Mac, 28
Mono, 28, 29

Objective-C, 28
OOP, 28, 31

PDG, 10

TinyCC, 41

UNICODE, 22
Unix, 11

VC++, 21

Windows, 28, 29, 32

XML, 28

Seznam zkratek

ANSI American National Standards Institute

API Application programming interface

AST Abstract syntax tree

BNF Backusova-Naurova forma

INI Initialization

IRL Intermediate Representation Language

ISO International Organization for Standardization

JIT Just in Time.

K&R Kernighan a Ritchie

OOP Objektově-orientované programování.

PDG Program Dependency Graph

RTTI Run-time type information

XML Extensible Markup Language

Literatura

[1] *American National Standard for Information Systems – Programming Language C* [online]. American National Standards Institute, 1988 [cit. 2011-06-30].
URL: <http://flash-gordon.me.uk/ansi.c.txt>

[2] *Rationale for American National Standard for Information Systems – Programming Language – C* [online]. American National Standards Institute, 1988 [cit. 2011-08-17].
URL: <http://www.cs.technion.ac.il/users/yechiel/CS/C++draft/rationale.pdf>

[3] Chanchal Kumar Roy and James R. Cordy. *A Survey on Software Clone Detection Research* [online]. Ontario: Queen's University at Kingston, 2007 [cit. 2012-04-18].
URL: <http://research.cs.queensu.ca/TechReports/Reports/2007-541.pdf>

[4] Brian W. Kernighan a Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, Inc., 1988. ISBN 0-13-110362-8.

[5] G. Antonoil, M. Di Penta, G.Masone, U. Villano: *Compiler Hacking for Source Code Analysis* [online]. Benevento: University of Sannio [cit. 2011-07-11].
URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.79.3899&rep=rep1&type=pdf>

[6] Paul Clough, *Plagiarism in natural and programming languages* [online]. University of Sheffield, Červen 2000 [cit. 2012-04-19].
URL: <http://ir.shef.ac.uk/cloughie/papers/plagiarism2000.pdf>

[7] Eric Giguere. *The ANSI Standard: A Summary for the C Programmer* [online]. 18. prosinec 1987 [cit. 2012-02-23].
URL: <http://www.ericgiguere.com/articles/ansi-c-summary.html>

[8] Clive D.W. Feather. *A brief description of Normative Addendum 1* [online]. [cit. 2012-05-02].
URL: <http://www.lysator.liu.se/c/na1.html>

[9] Jean-Jacques Levy. *Syntaxe BNF de C* [online]. [cit. 2011-06-22].
URL <http://www.enseignement.polytechnique.fr/profs/informatique/Jean-Jacques.Levy/poly/mainB/node23.html>

[10] Open Standards. *C - New in C9X* [online]. [cit. 2012-04-28].
URL: <http://www.open-std.org/jtc1/sc22/wg14/www/newinc9x.htm>

[11] *INI file: Format* [online]. [cit. 2012-03-05].
URL: <http://en.wikipedia.org/wiki/INI_file#Format>

Příloha A

Zdrojové kódy

Zdrojový kód A.1: Výpis testovacího skriptu preprocesorů

```
1  #!/bin/bash
2
3  if [ $# -eq 0 ]; then
4          echo "Chybí␣preprocessor!"
5          exit;
6  fi
7
8  # Preprocessor
9  CC=$1
10 TEST_COUNT=10
11 TEST_FILE="tcc-0.9.25/tcc.c"
12 TIME_CMD=/usr/bin/time
13 total=0
14
15 for (( i=1; i <= $TEST_COUNT; i++ )) do
16         t=$((( $TIME_CMD -f '%e' $CC -E $TEST_FILE; ) 1>/
              dev/null; ) 2>&1; )
17         echo $t "s"
18         total=`calc "$total+$t"`
19 done
20
21 avg=`calc "$total/$TEST_COUNT"`
22
23 echo $avg "s"
```

Zdrojový kód A.2: Zdrojový kód sdílené funkce pro výrazy

```
1  template<typename f, typename comp>
2  inline
3  void
4  Parser::process_simple_expression(Expression& expr, f call
       , comp cmp)
5  {
6      TokenType type = TOKEN_TYPE_INVALID;
7
8      // f { [ t1 | t2 ] f }
9      while (true)
10     {
11         // Zavoláme nižší úroveň
12         (this->*call)(expr);
13
14         // Přidáme operátor
15         if (type != TOKEN_TYPE_INVALID)
16             expr.add(type);
17
18         // Porovnáme token
19         if (cmp == tok())
20             type = static_cast<TokenType>(tok().type);
21         else
22             break;
23
24         // Načteme další token
25         consume_token();
26     }
27 }
```

Příloha B

Tabulky

Tabulka B.1: Velikosti datových typů

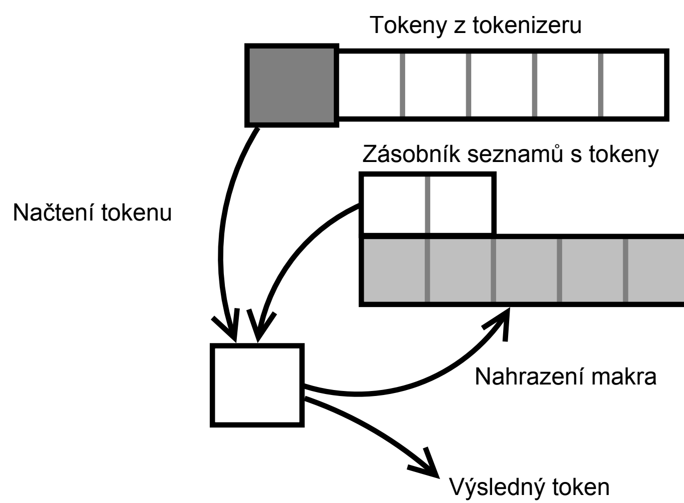
Typ	Minimální rozsah
`signed char`	-127 až +127
`unsigned char`	0 až 255
`short int`	-32767 až +32767
`unsigned short int`	0 až 65535
`int`	-32767 až +32767
`unsigned int`	0 až 65535
`long int`	-2147483647 až +2147483647
`unsigned long int`	0 až 4294967295
Typ	Minimální přesnost
`float`	6 cifer
`double`	10 cifer
`long double`	10 cifer

Tabulka B.2: Přehled časů testu rychlosti preprocesorů

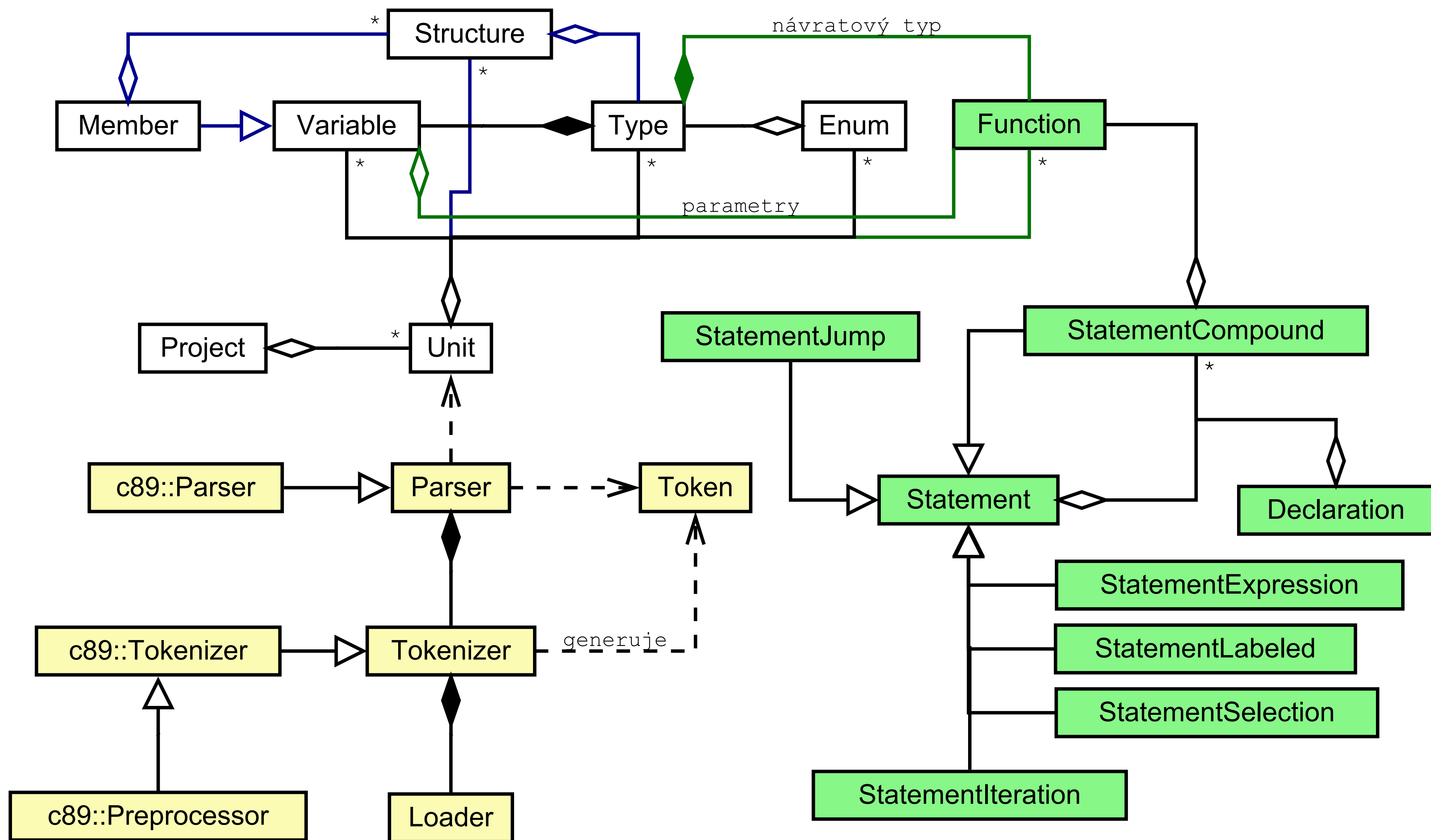
Průchod	GCC	Clang	CC
1	0,33 s	1,72 s	0,67 s
2	0,12 s	0,29 s	0,66 s
3	0,12 s	0,30 s	0,65 s
4	0,12 s	0,31 s	0,65 s
5	0,12 s	0,29 s	0,65 s
6	0,12 s	0,29 s	0,64 s
7	0,12 s	0,29 s	0,67 s
8	0,12 s	0,29 s	0,65 s
9	0,12 s	0,29 s	0,65 s
10	0,12 s	0,28 s	0,65 s
Průměr	0,14 s	0,435 s	0,654 s

Příloha C

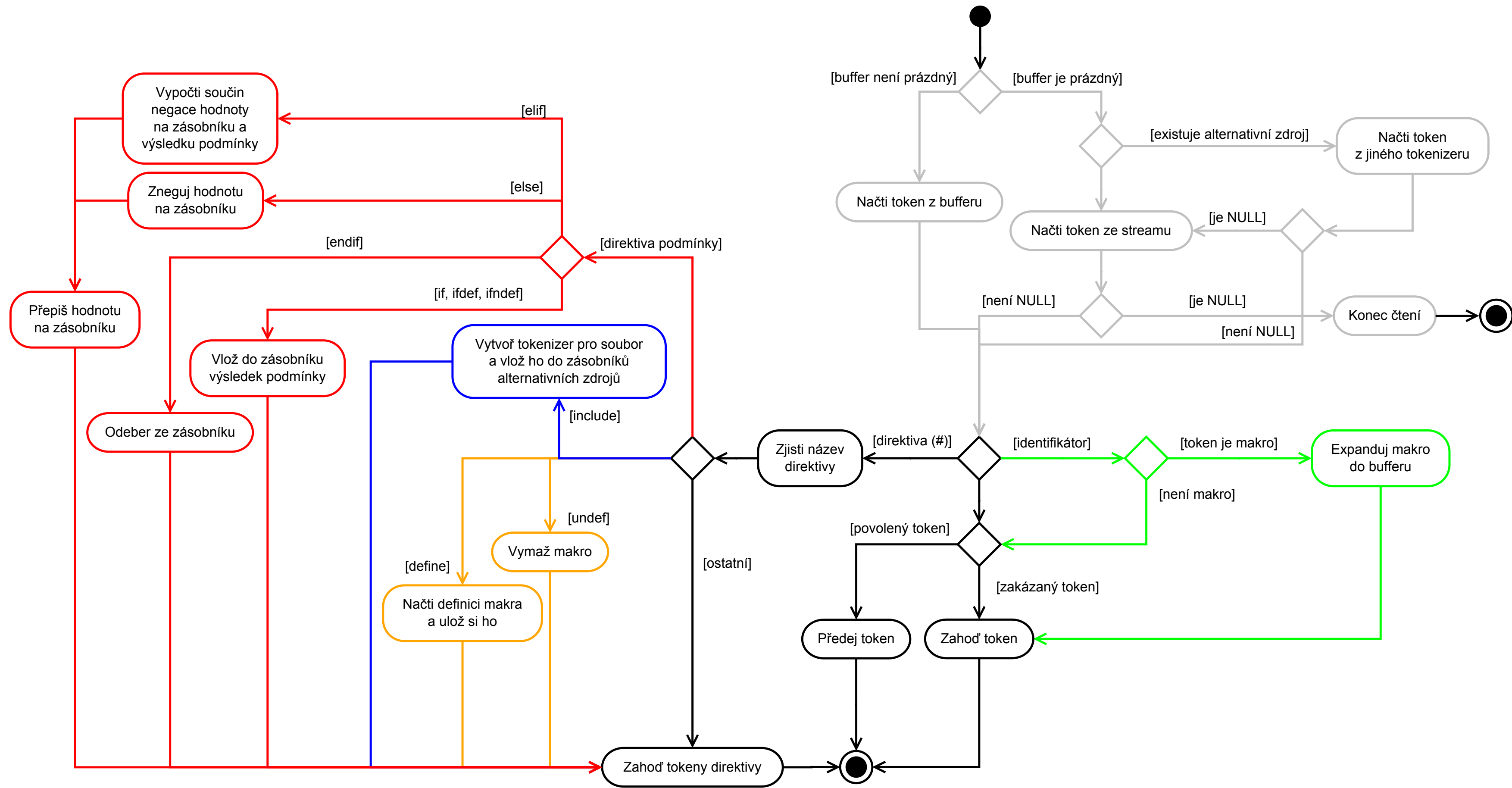
Diagramy



Obrázek C.1: Zjednodušený diagram práce preprocesoru



Obrázek C.2: Diagram vztahů



Obrázek C.3: Diagram aktivit preprocesoru

Příloha D

Reakce na syntaktické chyby

Zdrojový kód D.1: Neukončená direktiva #if

```
1 $ cat err0.c
2 #if 0
3 #if 1
4 $ cc -c err0.c
5 err0.c:0:0: error: unterminated #if
```

Zdrojový kód D.2: Chybějící výraz

```
1 $ cat err1.c
2 #if
3 #endif
4 $ cc -c err1.c
5 err1.c:1:2: error: #if with no expression
```

Zdrojový kód D.3: Nedeklarovaný identifikátor

```
1 $ cat err2.c
2 enum {
3     X, Y
4 };
5 int a[Z];
6 $ cc -c err2.c
7 err2.c:4:7: error: use of undeclared identifier 'Z'
8 int a[Z
```

Zdrojový kód D.4: Chybějící středník za deklarací

```
1 $ cat err3.c
2 int a()
3 float b()
4 return 0;
5 $ cc -c err3.c
6 err3.c:2:1: error: invalid token
7 float
```

Zdrojový kód D.5: Chybějšící název parametru

```
$ cat err4.c
int sub(int);
int add(int);
int sub(int) {
    return 3 - 5;
}
int add(int) {
    return 3 + 5 * ; /* Sem se překladač nedostane */
}
$ cc -c err4.c
err4.c: error: parameter name omitted
```

Zdrojový kód D.6: Redefinice funkce

```
$ cat err6.c
int a() {
    return 5;
}
int a() {
    return 8;
}
$ cc -c err6.c
err6.c:4:9: error: redefinition of 'a'
int a() {
```

Zdrojový kód D.7: Použití float jako rozměru pole

```
$ cat err7.c
int a[5.3f];
$ cc -c err7.c
err7.c:1:11: error: size of array has non-integer type
int a[5.3f]
```

Zdrojový kód D.8: Proměnná v konstantním výrazu

```
$ cat err9.c
int a;
enum {
    A = 0*a,
    B
};
$ cc -c err9.c
err9.c:3:8: error: use of undeclared identifier 'a'
 A = 0*a
```

Příloha E

Formát souboru otisku

Existují dva formáty souboru otisku. První je určen pro uložení překladové jednotky a druhý pro celý projekt. Pro jejich rozeznání je na začátku uložena speciální hodnota o velikosti 1B, která určuje typ uložených dat. Pro soubor s projektem je uložena hodnota `0xF1` a pro soubor s překladovou jednotkou je uložena hodnota `0xF2`.

Základem jsou primitivní datové typy doplněné o řetězec. Tento základ se používá pro uložení vlastností jednotlivých tříd popisné struktury. Kvůli přenositelnosti mezi různými platformami jsou základní typy ukládány v přesně dané velikosti (viz tabulka E.1). Datový typ `std::string` je ukládán jako klasický C řetězec, kdy konec řetězce je označen znakem `'\0'`[1].

Formát nespecifikuje *Endianness* cílové platformy a neřeší tak problém s přenosem souboru mezi platformami s jiným *Endianness*. Je předpokládáno použití jen na architektuře *x86*, které má *little-endian*.

V uložených datech nejsou přítomna žádná kontrolní data, která by ověřila správnost načítaného souboru. Debug verze aplikací používají upravený formát, kde se před každou základní hodnotou ukládá znak o hodnotě `0xFF`.

Tabulka E.1: Ukládané základní typy

Typ v aplikaci	Uložený typ
`bool`	`char`
`char`	`char`
`unsigned char`	`unsigned char`
`short`	`int16_t`
`unsigned short`	`uint16_t`
`int`	`int32_t`
`unsigned int`	`uint32_t`
`long`	`int64_t`
`unsigned long`	`uint64_t`
`std::string`	`char*`

[1] Druhou možností bylo uložení délky řetězce na začátek, ale to zabralo více místa ve výsledném souboru.

E.1 Formáty částí

Zde jsou popsány formáty jednotlivých částí (ukládaných tříd popisné struktury). Ukládá-li nějaká třída seznam, vždy platí, že na prvním místě je uložen počet hodnot pomocí typu `int32_t` (tj. 4B) a za ním následují samotné hodnoty seznamu. Pokud je ve sloupci „Velikost“ uveden otazník („?“), tak velikost není přesně daná a je závislá na datech, ze kterých se skládá daná položka.

E.1.1 Project

Jedná se jen o seznam překladových jednotek (`Unit`), který začíná počtem uložených hodnot.

Hodnota	Typ	Velikost
počet	`int32_t`	4B
Překladová jednotka	`Unit[]`	0–?

E.1.2 Unit

Překladová jednotka ukládá hlavně seznamy jednotlivých částí.

Hodnota	Typ	Velikost
název	`char*`	0–?
'\0'	`char`	1B
počet typů	`int32_t`	4B
typy	`Type`	0–?
počet výčtových typů	`int32_t`	4B
výčtové typy	`Enum`	0–?
počet struktur	`int32_t`	4B
struktury	`Structure`	0–?
počet proměnných	`int32_t`	4B
proměnné	`Variable`	0–?
počet funkcí	`int32_t`	4B
funkce	`Function`	0–?

E.1.3 Variable

V uložení proměnné není žádná specialita.

Hodnota	Typ	Velikost
název	`char*`	0–?
'\0'	`char`	1B
název souboru	`char*`	0–?
'\0'	`char`	1B
typ uložení	`int32_t`	4B
datový typ	`Type`	?

E.1.4 Type

Datový typ si v případě, že je složen z výčtového typu nebo struktury, ukládá jen jejich název.

Hodnota	Typ	Velikost
specifikátory	`int32_t`	4B
kvalifikátory	`int32_t`	4B
počet ukazatelů	`int32_t`	4B
kvalifikátory ukazatelů	`int32_t[]`	0–?
počet dimenzí	`int32_t`	4B
rozměry pole	`int32_t[]`	0–?
název výčtového typu	`char*`	0–?
`'\0'`	`char`	1B
název struktury	`char*`	0–?
`'\0'`	`char`	1B

E.1.5 Structure

Struktura ukládá informace o názvu, zda jde o `union` nebo `struct` a seznam členských proměnných.

Hodnota	Typ	Velikost
název	`char*`	0–?
'\0'	`char`	1B
název souboru	`char*`	0–?
'\0'	`char`	1B
zda je to `union`	`char`	1B
počet členských proměnných	`int32_t`	4B
členské proměnné	`Member[]`	0–?

E.1.6 Member

Jedná se jen o rozšíření třídy `Variable`, proto ukládá navíc jen velikost bitového pole.

Hodnota	Typ	Velikost
proměnná	`Variable`	?
počet bitů	`int32_t`	4B

E.1.7 Enum

Výčtový typ ukládá jen název, a seznam konstant s jejich hodnotami.

Hodnota	Typ	Velikost
název	`char*`	0–?
'\0'	`char`	1B
název souboru	`char*`	0–?
'\0'	`char`	1B
počet konstant	`int32_t`	4B
název konstanty	`char*`	0–?
'\0'	`char`	1B
hodnota konstanty	`int32_t`	4B
...	...	...

E.1.8 Function

Funkce může být uložena jako prototyp (bez těla) a nebo jako definice (s tělem). To rozděluje předposlední `bool` hodnota. Je-li `false`, poslední hodnota není uvedena.

Hodnota	Typ	Velikost
název	`char*`	0–?
'\0'	`char`	1B
název souboru	`char*`	0–?
'\0'	`char`	1B
typ uložení	`int32_t`	4B
návratový typ	`Type`	?
počet parametrů	`int32_t`	4B
parametry	`Variable`	0–?
zda má proměnný počet parametrů	`char`	1B
zda má definované tělo	`char`	1B
tělo funkce	`StatementCompound`	0–?

E.1.9 Declaration

Deklarace popisuje deklarace uváděné v těle funkcí. Může se jednat o proměnnou a nebo o prototyp funkce.

Hodnota	Typ	Velikost
typ uložení	`int32_t`	4B
specifikátory	`int32_t`	4B
kvalifikátory	`int32_t`	4B
počet ukazatelů	`int32_t`	4B
kvalifikátory ukazatele	`int32_t[]`	0–?
název typu	`char*`	0–?
'\0'	`char`	1B
název výčtového typu	`char*`	0–?
'\0'	`char`	1B
název struktury	`char*`	0–?
'\0'	`char`	1B
název deklarátoru	`char*`	0–?
'\0'	`char`	1B
zda má vnitřní deklarátor	`char`	1B
vnitřní deklarátor	`Declarator`	0–?
počet dimenzí pole	`int32_t`	4B
rozměry pole	`int32_t[]`	0–?
Zda se jedná o funkci	`char`	1B
počet parametrů	`int32_t`	4B
seznam parametrů	`Variable[]`	0–?
Zda má proměnný počet parametrů	`char`	1B

E.1.10 Statement

Jedná se o obecnou konstrukci, která podle typu ukládá specializované konstrukce. Typ uložené konstrukce specifikuje hodnota typu.

Hodnota	Typ	Velikost
typ	`int32_t`	4B
specializovaná konstrukce	?	?

E.1.10.1 StatementCompound

Konstrukce obsahuje seznam deklarací a seznam vnořených konstrukcí.

Hodnota	Typ	Velikost
počet deklarací	`int32_t`	4B
deklarace	`Declaration[]`	0–?
počet konstrukcí	`int32_t`	4B
konstrukce	`Statement[]`	0–?

E.1.10.2 StatementExpression

Konstrukce ukládá jen výraz.

Hodnota	Typ	Velikost
výraz	`Expression`	?

E.1.10.3 StatementIteration

Uložení cyklů závisí hlavně na typu cyklu. Cyklus `for` ukládá více informací než ostatní cykly.

while a do-while Tyto cykly obsahují jen výraz a konstrukci.

Hodnota	Typ	Velikost
typ	`int32_t`	4B
výraz	`Expression`	?
konstrukce	`Statement`	?

for For cyklus navíc obsahuje 2 výrazy.

Hodnota	Typ	Velikost
typ	`int32_t`	4B
inicializační výraz	`Expression`	?
testovací výraz	`Expression`	?
výraz	`Expression`	?
konstrukce	`Statement`	?

E.1.10.4 StatementJump

Tato konstrukce ukládá data na základě typu, který je vždy uložen jako první.

goto Konstrukce vyžaduje uložení názvu návěstí, na které má být proveden skok.

Hodnota	Typ	Velikost
typ	`int32_t`	4B
návěstí	`char*`	0–?
`'\0'`	`char`	1B

return Předposlední hodnota udává, zda je uvedena poslední hodnota.

Hodnota	Typ	Velikost
typ	`int32_t`	4B
zda je uveden výraz	`char`	1B
výraz	`Expression`	?

E.1.10.5 StatementLabeled

Konstrukce návěstí obsahuje název, hodnotu, pro kterou je návěstí určeno, a tělo konstrukce. Zda je přítomný název, hodnota a nebo ani jeden, tak se určuje typ návěstí. Je-li uveden název, jedná se o klasické pojmenované návěstí, které nemá hodnotu. Je-li uvedena hodnota, jedná se o `case` návěstí a název by neměl být přítomen. Není-li uvedena ani jedna hodnota, jedná se o `default` návěstí.

Návěstí Je uveden jen název návěstí a konstrukce.

Hodnota	Typ	Velikost
název	`char*`	0–?
`'\0'`	`char`	1B
hodnota	`int32_t`	4B
tělo konstrukce	`Statement`	?

case Je uvedena jen hodnota a konstrukce.

Hodnota	Typ	Velikost
`'\0'`	`char`	1B
hodnota	`int32_t`	4B
tělo konstrukce	`Statement`	?

default Je uvedena jen konstrukce.

Hodnota	Typ	Velikost
`'\0'`	`char`	1B
`0`	`int32_t`	4B
tělo konstrukce	`Statement`	?

E.1.10.6 StatementSelection

Uložení podmínkového bloku. Poslední 2 hodnoty nejsou v případě `switch` přítomny a pro `if` je poslední přítomná jen v případě, že předposlední je `true`.

Hodnota	Typ	Velikost
typ	`int32_t`	4B
výraz výběru	`Expression`	?
konstrukce	`Statement`	?
zda obsahuje `else` větev	`char`	0–1B
konstrukce `else` větve	`Statement`	0–?

E.1.11 Expression

Výraz ukládá jen seznam tokenů, ze kterých se skládá. Každý token ukládá svůj typ, klíčové slovo a hodnotu, jen pozice se neukládá.

Hodnota	Typ	Velikost
počet	`int32_t`	4B
typ tokenu	`uint32_t`	4B
klíčové slovo	`uint32_t`	4B
hodnota	`char*`	?
`'\0'`	`char`	1B
...	...	...

Příloha F

Uživatelská příručka

F.1 Požadavky na aplikace

Aplikace jsou napsány tak, aby vyžadovali jen standardní knihovny specifikované standardem *ISO C++98*. Jejich verze je daná verzemi, pro které byly aplikace slinkovány.

F.2 Překlad

Projekt je dodáván se soubory pro načtení do vývojových prostředí *Codelite* a *Visual Studio 2010*. K samotnému překladu tyto vývojové prostředí nejsou nutné, jelikož je dodáván soubor pro překlad pomocí *make*.

F.2.1 Požadavky

Aplikace využívají jen standardní knihovny standardu C++98, proto jedinými požadavky jsou překladač C++ s podporou standardu C++98 a program *make*.

F.2.2 Linux

Pro překlad lze použít vývojové prostředí *Codelite* a nebo jen program *make*. V případě překladu pomocí *make*, je použit překladač daný proměnnou `CC` a linker `LD`. Tyto hodnoty lze upravit přímým zadáním do souboru `makefile`.

F.2.3 Windows

Ve *Windows* lze použít vývojová prostředí *Codelite* a *Visual Studio 2010* a případně i překlad pomocí `makefile`. Ten je shodný s verzí pro *Linux* a proto je nutné explicitně nastavit překladač a linker.

F.2.4 Binární soubory

Výsledné binární soubory aplikací jsou, po překladu, uloženy v adresáři `bin` dané aplikace.

F.3 Program `cc` (překladač)

Aplikace je napsána tak, aby co nejvíce kopírovala chování překladače *GCC*. To znamená, že většina základních přepínačů sdílí s tímto překladačem. Ostatní přepínače překladač tiše ignoruje.

Program vždy přidává k vytvořeným souborům příponu `.fp`, aby se daly odlišit od klasických objektových souborů. Lze jí změnit přímo ve zdrojovém kódu a nebo pomocí proměnné v prostředí s názvem `FOOTPRINT_EXT`.

Při linkování objektových souborů, vytváří program navíc speciální soubor s jednotným názvem. Důvodem je snadnější rozpoznání pro další programy, aby byly schopny načíst správný soubor a nehledat ho.

Zápis

```
cc [-c|-E] [-Idir...] [-Dmacro[=defn]] [-Umacro] [-o outfile]
    infile...
```

Přepínače

`-c` Přeložení zdrojového souboru do objektového.

`-E` Použití preprocesoru.

`-Idir...` Přidání adresáře pro hledání hlavičkových souborů.

`-Dmacro[=defn]` Definice vlastního makra.

`-Umacro` Odstranění definice makra.

`-o outfile` Uložení výstupu do zadaného souboru.

Příklady

- Překlad zdrojového souboru:

```
$ cc -c main.c -o main.o
```

- Linkování objektových souborů:

```
$ cc main.o lib.o -o app
```

- Preprocesor:

```
$ cc -E main.c
```

F.4 Program `objdump`

Program pro práci s binárními otisky zdrojových souborů. Program je schopen vypsat zdrojový kód bez komentářů. Výsledný výstup lze omezit zadáním názvu symbolu, pro který chceme výstup omezit.

Zápis

```
objdump [-S] infile [symbol]
```

Přepínače

`-S` Vypíše zdrojový kód.

Příklady

- Vypíše přehled obsahu překladové jednotky `main.o`:

  ```
  $ objdump main.o
  ```

- Vypíše zdrojový kód funkce `main`:

  ```
  $ objdump -S main.o main
  ```

F.5 Program comparator

Program porovná zadané projekty. Argumenty musí být adresáře, které obsahují soubor `project.bin`. Tento soubor obsahuje otisk zdrojových souborů daného projektu.

Zápis

```
comparator [-c file] [-f file] [-i] dir...
```

Přepínače

`-c file` Nastaví vlastní soubor s hodnocením jednotlivých parametrů. Není-li přepínač uveden, je načten soubor `rating.ini` z aktuálního adresáře.

`-f file` Načte testované adresáře ze zadaného souboru.

`-i` Načte seznam testovaných adresářů ze standardního vstupu.

Příklady

- Porovná projekty `p1`, `p2` a `p3`:

  ```
  $ comparator p1 p2 p3
  ```

- Porovná projekty p1 a p2 s hodnocením uvedeným v souboru `rating.ini`:

  ```
  $ comparator -i rating.ini p1 p2
  ```

Tabulka F.1: Parametry konfiguračního souboru

Parametr	Popis	Platnost
`variables`	Proměnné	`Unit, Project`
`enums`	Výčtové typy	`Unit, Project`
`structures`	Struktury	`Unit, Project`
`functions`	Funkce	`Unit, Project`
`enum-name`	Název	`Enum`
`enum-count`	Počet konstant	`Enum`
`enum-values`	Názvy a hodnoty konstant	`Enum`
`type-specifiers`	Specifikátory	`Type`
`type-pointers`	Ukazatele	`Type`
`type-enum`	Výčtový typ	`Type`
`type-struct`	Struktura	`Type`
`type-array`	Pole	`Type`
`struct-name`	Název	`Structure`
`struct-count`	Počet členských proměnných	`Structure`
`struct-size`	Velikost	`Structure`
`struct-members`	Členské proměnné	`Structure`
`struct-member-variable`	Část proměnné	`Member`
`struct-member-bitfield`	Bitové pole	`Member`
`variable-name`	Název	`Variable`
`variable-type`	Datový typ	`Variable`
`function-name`	Název	`Function`
`function-type`	Návratový typ	`Function`
`function-parameter-count`	Počet parametrů	`Function`
`function-parameters`	Parametry	`Function`
`function-ellipsis`	Proměnný počet parametrů	`Function`
`function-declaration-count`	Počet deklarací	`Function`
`function-loop-count`	Počet cyklů	`Function`
`function-selection-count`	Počet podmínek	`Function`
`function-expression-count`	Počet výrazů	`Function`

Formát konfiguračního souboru

Konfigurační soubor obsahuje nastavení parametrů pro jednotlivé části porovnávání. Soubor je ve formátu *INI* a veškeré záznamy jsou uloženy v hlavní sekci (tj. neřeší sekce). Každý parametr má uvedenu hodnotu typu `float` a oblast platnosti. Pro tuto oblast se vždy hodnota parametru normalizuje, aby součet všech z dané oblasti byl `1`. Lze tak snadno zadat poměry jednotlivých parametrů a neřešit, zda je jejich součet `1` (což je při porovnávání potřeba).

Popisy jednotlivých parametrů a oblasti jejich platnosti jsou uvedeny v tabulce F.1.