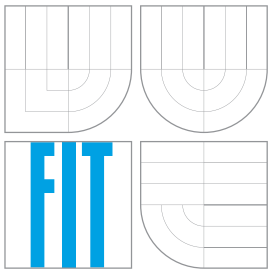
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

FILTRACE VAROVÁNÍ PŘEKLADAČŮ

COMPILER WARNINGS FILTER

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — MÁRIA KRAJČOVIČOVÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETR ZEMEK
SUPERVISOR

BRNO 2013

Abstrakt

Cílem bakalářské práce bylo navržení a vytvoření programu, který filtruje varování bez zásahu do zdrojového kódu. Práce se zabíva překladači GCC, Clang, Javac a GHC, jejich funkčností a varováními, které tyhle překladače produkují. Výsledkem práce je aplikace, která je schopna filtrovat varování pro uvedené překladače.

Abstract

Goal of this bachelor's thesis was design and implementation of program which filtrate warnings from compiler output without hitting the source codes. The bachelor's thesis describes compilers as GCC, Clang, Javac and GHC. It describes also their functionality and types of warnings from their outputs. Result of the bachelor's thesis is an aplication which is able warning filtration for mentioned compilers.

Klíčová slova

Překladač, GCC, Javac, Clang, GHC, varování

Keywords

Compiler, GCC, Javac, Clang, GHC, warning

Citace

Mária Krajčovičová: Filtrace varování překladačů, bakalářská práce, Brno, FIT VUT v Brně, 2013

Filtrace varování překladačů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Petra Zemeka

.....
Mária Krajčovičová
12. května 2013

Poděkování

Ďakujem vedúcemu práce Ing. Petrovi Zemekovi za odbornú pomoc a rady pri spracovaní tejto bakalárskej práce.

Obsah

Kapitola 1

Úvod

Úlohou tejto práce je oboznámiť čitateľa s jednotlivými prekladačmi, ktoré sú použité v tejto práci a medzi ktoré patria GCC, Clang, Javac a GHC. Čitateľ je taktiež oboznámený s rôznymi varovaniami týchto prekladačov ako aj rozdielmi v týchto varovaniach u jednotlivých prekladačov, prípadne v jednotlivých verziách vrámci jedného prekladača. Čitateľovi sú ukázané konkrétne prípady varovaní, ktoré je vhodné filtrovať a následne na základe získaných informácií je v práci navrhnutá aplikácia, ktorej cieľom bude filtrovať tieto varovania. Rovnako je čitateľovi ukázaný spôsob implementácie a testovania tejto aplikácie.

Konkrétne v druhej kapitole je čitateľ oboznámený bližšie s pojmom „prekladač“, je tu objasnené, čo to prekladač vlastne je. Taktiež je v tejto kapitole vysvetlené aká je funkcia prekladača a aké rôzne spôsoby prekladu zdrojového kódu existujú. Ďalej je popísané zloženie prekladača a jednotivé fázy, ktoré sa počas prekladu vykonajú.

Tretia kapitola je venovaná konkrétnym prekladačom, ich štruktúre a a druhom varovaní, ktoré tieto prekladače produkujú. Postupne sú tu v jednotlivých častiach rozobrané prekladače GCC, Clang, Javac a GHC.

V štvrtej kapitole sú popísané konkrétne varovania jednotlivých prekladačov, ktoré je možné v istých situáciach považovať za bezproblémové a teda je vhodné tieto varovania filtrovať. V jednotlivých častiach kapitoly sú ku každému prekladaču ukázané príklady takýchto varovaní a následne je objasnené, prečo sa dajú tieto varovania považovať za bezproblémové a mali by byť filtrované.

Piata kapitola obsahuje návrh aplikácie, ktorej úlohou je filtrovanie neželaných varovaní. Kapitola ukazuje spôsob ako je možné jednotlivé varovania z výstupu prekladačov filtrovať bez zásahu do zdrojového kódu prekladaného súboru. Ďalej sú v tejto kapitole popísané problémy spojené s filtrovaním varovaní u jednotlivých prekladačov, s ktoré budú musieť byť v implementovanej aplikácií riešené. Rovnako táto kapitola obsahuje spôsob riešenia týchto problémov. V tejto časti práce je taktiež rozobraná možnosť rozšíriteľnosti aplikácie a spôsob akým je možné túto rozšíriteľnosť implementovať.

Šiesta kapitola popisuje implementáciu aplikácie, ktorá bola navrhnutá v predchádzajúcej kapitole. Rovnako je tu ukázané na konkrétnych prípadoch ako boli riešené problémy s filtráciou varovaní u jednotlivých prekladačov.

Siedma kapitola práce je venovaná testovaniu aplikácie. Obsahuje konkrétne testovacie prípady pre jednotlivé prekladače, na ktorých je ukázané filtrovanie varovaní, ktoré si užívateľ neželá zobrazovať.

Na záver sú zhrnuté jednotlivé poznatky, ktoré sú publikované v tejto práci a zároveň sú zhodnotené výsledky, ktoré boli dosiahnuté.

Kapitola 2

Prekladače

Pri písaní tejto kapitoly som čerpala z [10], [6], [12], [8] a [11]. Prekladač alebo kompilátor je počítačový program, ktorý slúži na transformáciu zdrojového kódu, ktorý je napísaný v niektorom programovacom jazyku do iného programovacieho jazyka, najčastejšie strojového kódu. Strojový kód alebo tiež strojový jazyk je súbor inštrukcií priamo vykonateľných procesorom počítača. Strojový kód sa skladá z reťazcov bitov, tieto bity zodpovedajú inštrukciám, čo sú elementárne príkazy, ktoré vie počítač vykonať.

Presnejšie prekladač číta zdrojový kód, ktorý je napísaný v *zdrojovom jazyku* (angl. the source language) a transformuje ho do *cieľového jazyku* (angl. the target language). Zdrojový a cieľový program sú navzájom *funkčne ekvivalentné*, čo znamená, že tieto programy sú síce napísané v rôznych programovacích jazykoch, ale vykonávajú rovnaký program.

U prekladačov rozlišujeme *preklad nadol* a *preklad nahor*. U prekladu nadol prekladač transformuje zdrojový kód z vyššieho programovacieho jazyka do nižšieho. U prekladu nahor je to naopak, teda prekladač transformuje zdrojový kód z nižšieho programovacieho jazyka do vyššieho. Takýto prekladač sa nazýva tiež *dekompilátor*, ktorý je užitočný najmä pre znovuzískanie strateného zdrojového kódu. V prípade, ak je cieľom prekladu získanie optimálneho kódu, nazýva sa takýto prekladač *optimalizačný kompilátor*.

Súčasťou procesu prekladu sú taktiež *diagnostické správy*. Tieto správy majú podobu buď *chybových hlásení*, ktorými prekladač informuje užívateľa o prítomnosti chýb v zdrojovom kóde alebo *varovaní*, ktorými prekladač upozorňuje užívateľa na situáciu, ktorá je z hľadiska syntaxi správna, ale môže mať pre programátora iný význam ako očakáva.

2.1 Spôsoby prekladu

Spočiatku pracovali programátori s najprimitívnejšími inštrukciami počítača, nazývané aj strojový jazyk. Tieto inštrukcie mali tvar reťazcov zložených z jednotiek a núl.

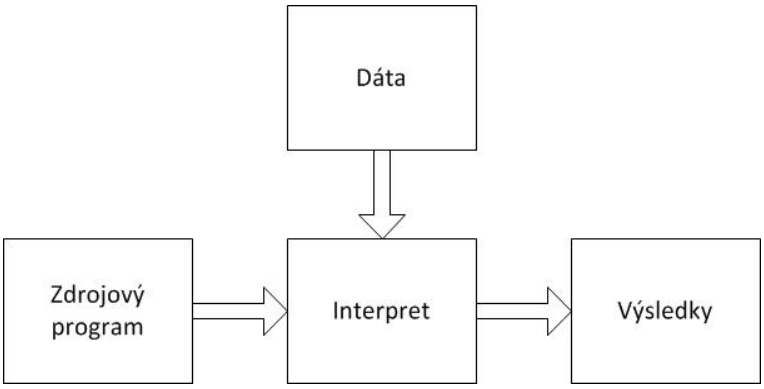
S príchodom zložitejších počítačov narastal aj počet inštrukčných súborov, čo viedlo k vzniku *assemblerov*, čo sú programy, ktoré umožňovali prevod memotechnických názvov inštrukcií do binárneho kódu, ktorý už mohol počítač vykonať.

Neskôr sa začali vyvíjať užívateľsky orientované programovacie jazyky vyššej úrovne ako napríklad programovací jazyk *BASIC* alebo *FORTRAN*. U týchto jazykov vyššej úrovne mohli programátori pracovať s kódom, ktorý sa podobal slovám a vetám, ktoré boli potom prostredníctvom kompilátorov a interpretov prekladané opäť do strojového jazyka.

S použitím týchto „vyšších“ programovacích jazykov sa objavuje niekoľko spôsobov prekladu týchto súborov.

2.1.1 Interpret

Interpret alebo *interpretačný prekladač* je program, ktorý prekladá a mení inštrukcie programového kódu priamo na konkrétnu akcie. Na obrázku 2.1 je vidieť schéma činnosti interpretu.



Obrázek 2.1: Interpret

Práca s interpretom je pre programátorov pomerne jednoduchá, keďže interpret kód číta a zároveň ho ihneď prevádza do inštrukcií. U interpretov je taktiež možné modifikovať text programu a to aj priamo za behu. U jazykov ako *BASIC* je možné meniť príkazy bez toho, aby sa musel znovu prekladať celý program. Pri výskyte chyby sú vždy presné informácie o jej výskyte, čo pri programovaní zjednodušuje odhalenie jej príčiny. Keďže interprety negenerujú pri preklade strojový kód, sú týmto značne strojovo nezávislé. Pre prenos na iný počítač potom stačí len interpret znovu skompilovať.

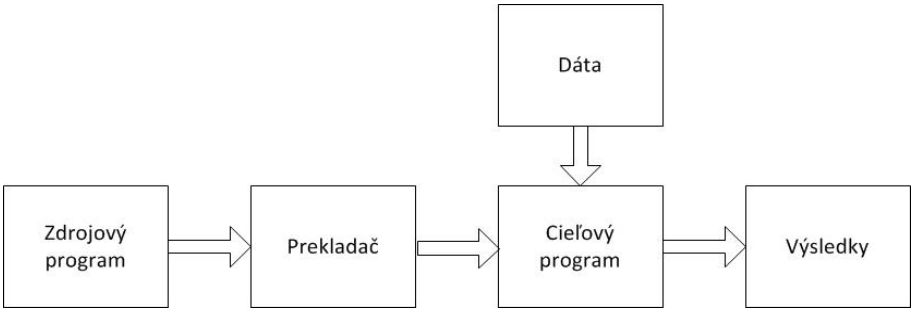
Nevýhodou interpretov je, že preklad je pomalší ako ako pri kompilácií, pretože je potrebné vždy analyzovať zdrojový príkaz, vždy keď naň program narazí. Pre pomer rýchlostí medzi interpretovaním a kompilovaním programu sa uvádzajú hodnoty medzi 10:1 až 100:1, kde sa hodnoty závisia od konkrétneho programovacieho jazyka. U interpretov je taktiež väčšia pamäťová náročnosť ako u kompilátorov, pretože aj pri behu programu musí byť neustále k dispozícií celý prekladač.

V súčastnosti sa väčšina interpretujúcich programov označuje ako skripty a samotný interpret ako skriptové jadro (Script Engine). V niektorých jazykoch ako napríklad Visual Basic sa interpret nazýva behová knižnica. V Jazyku Java sa zasa interpret označuje pojmom virtuálny stroj.

2.1.2 Kompilátor

Kompilátor alebo *kompilačný prekladač* prekladá program do ekvivalentného programu v strojovom jazyku počítača. Na obrázku 2.2 je vidieť schéma činnosti kompilačného prekladača. Po preklade, ktorému sa hovorí aj kompilácia sa vytvorí takzvaný „objektový“ súbor. Tento súbor však ešte nie je spustiteľný. Následne kompilačný prekladač spustí zostavovací program, ktorý sa nazýva *linker*. Linker následne objektové súbory zostaví do spustiteľného programu. Vo fáze objektového súboru ešte nie sú známe adresy premenných a funkcií. Linker zaisťuje nahradenie relatívnych adries absolútnymi a prevedie odkazy na príslušné funkcie knižníc.

Kompilačné prekladače prinášajú so sebou krok naviac tým, že umožňujú preklad zdrojového kódu, ktorý je ľuďom zrozumiteľný do objektového kódu, ktorý je zrozumiteľný



Obrázek 2.2: Kompilátor

stroju. Tento krok naviac je nepohodlný, ale výhodou je, že preložené programy bežia veľmi rýchlo, pretože časovo náročná úloha preloženia kódu prebehne len raz a pri spustení sa už nevyžaduje. Výhodou je aj to, že spustiteľný program je možné šíriť a spúšťať medzi užívateľmi, ktorý žiadny kompilátor nemajú.

Nevýhodou u niektorých kompilačných prekladačov je náročnejšie hľadanie chýb v zdrojovom programe, pretože informácie o mieste chyby sú vyjadrené v pojmoch strojového jazyka ako napríklad výpis pamäti alebo adresy. Avšak v moderných kompilačných prekladačoch sa už nachádzajú pomocné dátové štruktúry, ktoré umožňujú ladenie programu na úrovni zdrojového jazyka ako napríklad vypisovať hodnoty pomocných premenných, vykonávať program postupne po jednotlivých riadkoch alebo postupnosť volaní funkcií v programe. Niektoré kompilačné prekladače sa označujú aj pojmom IDE (Integrated Development Environment), pretože obsahujú aj textové editory, do ktorých sa píše kód a ten je možné následne preložiť.

2.2 Štruktúra prekladača

Základnou činnosťou prekladača je analyzovať zdrojový program a k nemu následne vytvoriť odpovedajúci cieľový program. Preklad zdrojového programu prebieha v niekoľkých fázach. Tento proces sa dá rozdeliť na dve časti a to na *analytickú časť* a *syntetickú časť*. Pri analýze je zdrojový program rozložený na jeho základné časti, z ktorých sú následne pri syntéze budované moduly cieľového programu.

Delenie na analytickú a syntetickú časť prekladača zodpovedá deleniu na *prednú časť* prekladača (angl. front end) a *zadnú časť* prekladača (angl. back end). Menšie odlišnosti nastávajú pri *medzikóde*, kde predná časť prekladača vykonáva syntézu a koncová časť tento kód analyzuje.

Samotný prekladač sa typicky skladá z niekoľkých komponentov:

- Lexikálny analyzátor
- Syntaktický analyzátor
- Sémantický analyzátor
- Generátor cieľového programu/interpretácia

2.2.1 Analytická časť prekladača

Úlohou analytickej časti je rozložiť zdrojový kód. Analýza sa skladá z lexikálnej, syntaktickej a sémantickej analýzy, pričom nezávisí na cieľovom kóde. Pri zmene cieľového kódu sa mení len syntéza a analýza ostáva nezmenená.

Lexikálna analýza

Lexikálny analyzátor alebo *scanner* je prvá časť prekladača, ktorá postupne číta znaky zdrojového programu a prevádza ich na postupnosť najmenších častí s vlastným významom, ktoré nazývame *lexikálne symboly*. Tieto symboly následne slúžia ako vstup pre syntaktický analyzátor.

Okrem samotného symbolu sa do syntaktického analyzátora posielajú taktiež údaje o druhu konkrétneho symbolu (či sa jedná o operátor, identifikátor, kľúčové slovo atď.). Celý súbor údajov, ktoré sa predávajú z lexikálneho analyzátora do syntaktického sa označujú ako *token*. Znaky ako medzery alebo komentáre, ktoré pre program nemajú žiadny zmysel sú lexikálnym analyzátorom počas analýzy obyčajne vynechávané.

Chyby sa v lexikálnej analýze odhaľujú len v prípade, ak znaky na vstupe netvoria žiadny symbol jazyka.

Syntaktická analýza

Syntaktický analyzátor alebo *parser* využíva tokeny z lexikálneho analyzátoru k analýze zdrojového jazyka, ktorou sa snaží zistiť, či zdrojový text odpovedá gramatike prekladaného jazyka. Úlohou syntaktického analyzátora je taktiež rozpoznať, či je program zapísaný správnym spôsobom, napríklad či po úvodnom návestí „begin“ naväzuje v jazykoch, ktoré si to vyžadujú aj koncové návestie „end“, alebo pri vyhodnocovaní matematických výrazov určuje v akom poradí sa budú vykonávať jednotlivé časti (násobenie má prednosť pred sčítaním atď.). Syntaktický analyzátor behom svojej práce vytvára *derivačný strom* (parse tree), ktorý popisuje štruktúru zdrojového programu a zároveň slúži ako vstup sémantického analyzátora.

Sémantická analýza

Sémantický analyzátor uzatvára analytickú časť prekladača. V tejto časti prekladu sa postupne prechádzajú symboly získané syntaktickou analýzou a priraďuje sa im význam. Zároveň sa tu vykonávajú rôzne kontroly, ktoré závisia už na konkrétnom programovacom jazyku. Najčastejšie sú to rôzne typové kontroly alebo kontroly deklarácií, ktoré zisťujú, či sú použité premenné deklarované a zároveň, či je ich použitie vzhľadom k dátovému typu správne. Taktiež pri operáciach kontroluje správne použitie jednotlivých operátorov, prípadne sa prevedie u niektorých jazykov implicitná konverzia dátového typu. Implicitná zmena dátového typu sa vykoná podľa priorít jednotlivých typov, s tým, že operand, ktorý má typ s nižšou prioritou sa automaticky pretypuje na typ operandu s vyššou prioritou. Napríklad v programovacom jazyku Pascal má premenná typu *real* väčšiu prioritu ako premenná typu *integer*, preto pri sčítaní premenných typu *integer* a *real* nastane implicitná zmena, kde typ *integer* bude zmenený na typ *real*. U programovacích jazykov, ktoré implicitné pretypovanie nepodporujú nastáva v takýchto prípadoch sémantická chyba.

2.2.2 Syntetická časť prekladača

Úlohou syntetickej analýzy je kontrola jednotlivých kódov a ich následná optimalizácia. Zároveň sa v tejto časti generuje cieľový kód, prípadne sa vykonáva interpretácia. Syntéza závisí na cieľovom kóde, preto akákoľvek zmena v cieľovom kóde vyvolá zmenu. Do syntetickej časti patria:

Optimalizácia kódu

Cieľom optimalizácie je vylepšiť kód tak, aby jeho výsledkom bol rýchlejší alebo kratší cieľový kód prípadne zníženie pamäťových nárokov. U optimalizácie rozlišujeme *vysokú* a *nízku úroveň* optimalizácie, kde *vysoká* alebo tiež *strojovo nezávislá* je na úrovni zdrojového kódu a medzikódu a *nízka* alebo tiež *strojovo závislá* je na úrovni inštrukcií a architektúry počítača.

Generovanie medzikódu

Niektoré prekladače po dokončení analýzy zdrojového kódu generujú *medzikód*, nazývaný aj *intermediálny kód*. Medzikód je vytvorený po analýze kódu, ale preklad do cieľového kódu nastáva až v okamžiku spustenia, čo umožňuje istú platformovú nezávislosť.

Medzikód slúži aj ako podklad pre optimalizáciu a generovanie cieľového kódu. U interpretov môže slúžiť aj ako konečný produkt v prekladači.

Generovanie cieľového kódu

Generovanie cieľového kódu je posledná fáza prekladu. Generovaným cieľovým kódom je strojový kód alebo program v jazyku assembler. Zároveň tu dochádza k prideleniu pamäte premenným programu. Väčšina prekladačov však neprevádza priamo analyzovaný zdrojový kód do cieľového, ale generuje si špeciálny typ kódu, ktorý sa nazýva *medzikód*. Generátor cieľového kódu potom následne prekladá inštrukcie medzikódu do funkčne ekvivalentnej postupnosti strojových inštrukcií.

Tabuľka symbolov

Podstatnou časťou prekladača je tzv. *tabuľka symbolov*, ktorá slúži ako centrálne úložisko dát všetkých častí prekladača. Počas analýzy zdrojového kódu sa zaznamenávajú jednotlivé identifikátory a informácie o týchto identifikátoroch. V tabuľke sú teda uložené informácie o type, východzích hodnotách alebo veľkosti objektu. Taktiež sa tu nachádza záznam o jednoznačnej identifikácií jednotlivých identifikátorov a zaistenie ich jedinečnosti.

Kapitola 3

Štruktúra a typy varovaní jednotlivých prekladačov

Táto kapitola sa venuje jednotlivým prekladačom, ktorými sú v tomto prípade GCC, Clang, Javac a GHC. Je tu taktiež zobrazená štruktúra týchto prekladačov a spôsob akým prekladajú zdrojové súbory. V kapitole sú popísané niektoré prepínače, ktoré je možné pri preklade súborov použiť a taktiež sú tu ukázané a popísané varovania, ktoré jednotlivé prekladače produkujú, prípadne rozdiely medzi varovaniami v jednotlivých verziách prekladačov.

3.1 GNU Compiler Collection

Pri písaní tejto podkapitoly som čerpala z [15] a [3]. *GNU Compiler Collection* alebo *GCC* je sada prekladačov, ktorý bol vytvorený v roku 1987 Richardom Stallmanom v rámci GNU projektu a slúži ako prekladač pre programovacie jazyky ako C (gcc), C++ (g++), Java (GCJ), Objective C (gobjc), Fortran (gfortran) a Ada (GNAT). GCC je tzv. *slobodný softvér*, čo znamená, že takto označené programy majú dostupný zdrojový kód s koncový užívatelia majú právo voľne používať, modifikovať a šíriť tento softvér.

V súčastnosti je GCC súčasť operačných systémov ako GNU/Linux, BSD alebo Mac OS X. Taktiež je súčasťou mnohých IDE ako napríklad *Code::Blocks*, *Eclipse* alebo *NetBeans*. GCC je napísaný prevažne v jazyku C a obsahuje veľké množstvo optimalizácií.

3.1.1 Štruktúra

Štruktúra GCC sa skladá z troch častí. Prvú časť tvorí predná časť (angl. front end), ktorá je plne závislá na zdrojovom kóde. V tejto časti sa zdrojový kód rozdelí na malé časti, z ktorých sa následne vytvorí abstraktný syntaktický strom. Keďže pravidlá pre rozklad kódu lexikálnym analyzátorom nie sú pre všetky programovacie jazyky rovnaké, potom aj predné časti sú pre jednotlivé programovacie jazyky odlišné, čo viedlo k problému, že každý programovací jazyk si tvoril vlastný druh syntaktického stromu. Z toho dôvodu boli vytvorený formát pre popis syntaktických stromov, ktorý je nezávislý na jednotlivých jazykoch. Tento formát sa označuje ako *GENERIC*.

Ďalšou časťou je stredná časť (angl. middle end), ktorá je nezávislá ako na zdrojovom kóde, tak aj na cieľovom kóde. Ako vstup slúži syntaktický strom prevedený do formátu GENERIC. Následne prebiehajú rôzne analýzy a optimalizácie kódu. Výsledkom je kód v jazyku označovanom ako *Register Transfer Language* (RTL), ktorý už je podobný jazyku assembler.

Poslednou časťou je zadná časť (angl. back end), ktorá je závislá na cieľovom jazyku a architektúre počítača. Vstupom je kód v RTL, v ktorom sú vykonané ešte posledné optimalizácie a následne je tento kód transformovaný do strojového kódu pre konkrétnu architektúru počítača.

3.1.2 Varovania

V GCC existuje niekoľko rôznych prepínačov, ktoré ovplyvňujú výpisy varovaní. Preklad kódu v GCC bez použitia týchto prepínačov môže viesť k vytvoreniu spustiteľného súboru, ktorého výstupom môžu byť chybné výsledky.

K prepínačom zobrazujúcim varovania patrí napríklad prepínač **-Wall**, ktorý zobrazuje základnú sadu varovaní, ktoré sa týkajú najmä zdrojového kódu a jeho technickej validity. Súčasťou tohto prepínača sú ďalšie prepínače, ktoré sa dajú použiť individuálne a zobrazujú varovania len pre určitý typ chyby. K týmto prepínačom patria napríklad prepínače:

- **-Wformat** – zobrazuje varovania, ktoré sa týkajú funkcií *printf* a *scanf*;
- **-Wunused** – zobrazuje varovania pri nevyužitých premenných;
- **-Wreturn-type** – zobrazuje varovania u funkcií, ktoré sú definované bez návratového typu a neobsahujú ani *void* v deklarácií.

K prepínačom v GCC, ktoré zobrazujú varovania, ktoré prepínač -Wall neodhalí patria napríklad:

- **-Wextra** alebo tiež **-W** – na rozdiel od -Wall zobrazuje napríklad varovania, ktoré sú spôsobené porovnávaním znamienkových a bezznamienkových premenných;
- **-Wconversion** – zobrazuje varovania, ktoré upozorňujú na implicitnú konverziu, ktorá môže mať neočakávané výsledky;
- **-Wshadow** – zobrazuje varovania, ktoré upozorňujú na redeklaráciu už deklarovanej premennej.

K prepínačom týkajúcich sa varovaní patrí aj:

- **-w** – zakazuje zobrazenie akýchkoľvek varovaní;
- **-Werror** – mení varovania na chybové hlásenia;
- **-pedantic** – zobrazuje varovania striktne podla zvolenej normy.

Všetky prepínače týkajúce sa varovaní v GCC sú uvedené v dokumentácií [1].

Podoba varovaní sa u jednotlivých verzií prekladača GCC celkom výrazne mení. Majme napríklad časť zdrojového kódu, ktorá po preklade prekladačom GCC zobrazí varovanie, ktoré je spôsobené nesprávnym použitím funkcie `printf()`:

```
#include<stdio.h>
int i;
int main(){
    printf("Skuska vypis",i);
    return 0;
}
```

Po preklade tohto kódu s prepínačom `-Wall` vo verzií prekladača `gcc-4.4` sa nám zobrazí následujúce varovanie:

```
$ gcc-4.4 -Wall test0.c
test0.c: In function ‘main’:
test0.c:4: warning: too many arguments for format
```

Varovania vo verzií `gcc-4.4` obsahujú vo výstupe prekladača na začiatku meno prekladaného súboru. Za menom súboru sa nachádza číslo riadku, v ktorom varovanie nastalo. Tieto údaje sú od seba oddelené dvojbodkou. Za číslom riadku sa nachádza dvojbodkou oddelené kľúčové slovo `warning` a popis situácie, ktorá dané varovanie spôsobila.

Po preklade rovnakého zdrojového kódu, ale vo verzii prekladača `gcc-4.5` sa nám zobrazí následujúce varovanie:

```
gcc-4.5 -Wall test0.c
test0.c: In function ‘main’:
test0.c:4:5: warning: too many arguments for format
```

Ako je vidieť v príklade varovanie vo verzií `gcc-4.5` sa líši od varovania vo verzií `gcc-4.4` tým, že do výpisu varovania je pridaná hodnota, ktorá udáva pozíciu varovania vrámci riadku, v ktorom sa varovanie nachádza. Táto pozícia je umiestnená za hodnotou čísla riadku a je od nej oddelená dvojbodkou, v našom príklade je to hodnota `5`.

Po preklade rovnakého zdrojového kódu, ale vo verzii prekladača `gcc-4.6` sa nám zobrazí následujúce varovanie:

```
gcc-4.6 -Wall test0.c
test0.c: In function ‘main’:
test0.c:4:5: warning: too many arguments for format [-Wformat-extra-args]
```

Ako je vidieť na tomto príklade, u varovaní od verzie `gcc-4.6` pribudla v popise varovania informácia o formáte daného varovania, ktorá sa nachádza v hranatých zátvorkách a v našom prípade je to `-Wformat-extra-args`

Po preklade rovnakého zdrojového kódu, ale vo verzii prekladača `gcc-4.8` sa nám zobrazí následujúce varovanie:

```
test0.c: In function ‘main’:
test0.c:4:5: warning: too many arguments for format [-Wformat-extra-args]
     printf("Skuska vypis",i);
     ^
```

Ako je vidieť v tomto príklade, varovania vo verzií `gcc-4.8` obsahujú navyše vo výstupe niekoľko riadkov, v ktorých sa nachádzajú doplňujúce informácie o varovaní. Konkrétne je to ukážka riadku, v ktorom varovanie nastalo a pomocou znaku ^ je presne znázornená pozícia varovania v danom riadku.

Varovania v príklade potom oznamujú, že chyba nastala v súbore `test2.c` v riadku číslo `4`, na pozícií `5` a príčinou chyby je priveľa argumentov v použitej funkcií.

3.2 Clang

Pri písaní tejto podkapitoly som čerpala z [4] a [9]. *Clang* slúži ako prekladač pre programovacie jazyky rodiny C ako C, C++ alebo Objective-C. Cieľom tohto prekladača je vytvoriť

konkurenčný prekladač pre GCC. Clang rovnako ako GCC je taktiež *slobodný softvér* a medzi sponzorov vývoja patrí napríklad aj firma Apple.

Oproti GCC ponúka Clang rýchlejší preklad a menšie vyťaženie pamäte. Veľkou výhodou sú aj expresívnejšie a rozsiahlejšie chybové hlásenia a varovania. Clang je napísaný v podmnožine jazyka C++ a je plne kompatibilný s GCC. Clang taktiež umožňuje tzv. *statickú analýzu*, ktorá dokáže identifikovať niektoré chyby bez spustenia programu, čo zjednodušuje kontrolu pri písaní programov.

3.2.1 Štruktúra

Pri preklade zdrojového kódu pracuje Clang ako predná časť prekladača. Súčasťou prekladu je taktiež LLVM (Low Level Virtual Machine), ktorý slúži ako zadná časť prekladača.

Pri preklade Clang prevádza zdrojový kód do LLVM byte kódu. LLVM je framework pre tvorbu prekladačov, ktorý prevádza kód v LLVM byte kóde do inštrukcií pre konkrétne počítačové architektúry a zároveň tu prebiehajú rôzne optimalizácie kódu.

3.2.2 Varovania

V Clangu rovnako ako v GCC sa nachádzajú prepínače, ktoré ovplyvňujú výpis varovaní. Tieto prepínače sú však vo väčšine prípadov zhodné s GCC.

```
#include <stdio.h>
int main(){
   printf("%.*d");
   return 0;
}
```

Po preklade tohto zdrojového kódu vo verzií `2.9` sa nám zobrazí následovné varovanie:

```
$ clang test.c
test.c:3:12: warning: '.*' specified field precision is missing a matching
    'int' argument
        printf("%.*d");
                  ^
```

Všetky vygenerované varovania obsahujú na začiatku meno prekladaného súboru. Za týmto menom nasleduje číslo riadku, v ktorom sa nachádza potenciálna chyba a zároveň sa tu nachádza aj číslo, ktoré udáva umiestnenie potenciálnej chyby v danom riadku. Nasleduje kľúčové slovo `warning` a popis chyby, ktorá dané varovanie vyvolala. Rovnako ako v GCC sú jednotlivé prvky varovania oddelené dvojbodkou. Pod týmto popisom sa nachádza konkrétny riadok zo zdrojového kódu, ktorý vyvolal dané varovanie a pomocou znaku `^` je určená pozícia daného varovania vrámci riadku.

Varovanie v tomto prípade potom oznamuje, že chyba nastala v súbore `test.c` v riadku číslo `3` a umiestnenie chyby v tomto riadku je na mieste číslo `12`. Príčinou chyby je chýbajúci argument typu `int` vo funkcií.

Niektoré varovania prekladača Clang môžu navyše v časti popisu varovania obsahovať formát tohto varovania, ako je vidieť na následujúcom príklade varovania, ktoré bolo vyvolané po dosadení neinicializovanej premennej do funkcie:

```
test2.c:15:19: warning: variable 'u' is uninitialized when used here
           [-Wuninitialized]
    pocitaj(i, o, u);
                  ^
```

Ako je vidieť formát varovania sa nachádza rovnako ako u prekladača GCC v hranatých zátvorkách. V našom prípade je to `-Wuninitialized`

Od verzie `3.0` sa vo výpise prekladača spolu s varovaniami môžu zobrazovať dodatočné poznámky k varovaniam. Tieto poznámky vo výpise obsahujú kľúčové slovo `note`, ako je vidieť na následujúcom príklade rovnakého varovania vo verzií `3.2`:

```
test2.c:15:19: warning: variable 'u' is uninitialized when used here
           [-Wuninitialized]
    pocitaj(i, o, u);
                  ^
test2.c:12:14: note: initialize the variable 'u' to silence this warning
    int i,o,u;
             ^
              = 0
```

Poznámka rovnako ako varovanie obsahuje vo výpise meno prekladaného súboru, za ním sa nachádza dvojbodkou oddelené čísla riadku a pozícia v rámci riadku, ku ktorým sa poznámka vzťahuje. Nasleduje kľúčové slovo `note` a za ním opäť dvojbodkou oddelená poznámka, ktorá popisuje ako je možné dané varovanie odstrániť. Poznámka rovnako ako varovanie obsahuje na ďalších riadkoch dodatočné informácie, ktoré ukazujú konkrétny riadok, v ktorom je možné dané varovanie „umlčať“, prípadne spôsob ako toto varovanie umlčať.

V našom príklade poznámka ukazuje na riadok číslo `12` s pozíciou `14`, v ktorom bola daná premenná inicializovaná. Prekladač nám rovnako radí inicializovať premennú pomocou priradenia hodnoty nula tejto premennej.

3.3 GHC

Pri písaní tejto podkapitoly som čerpala z [13] a [14]. *GHC* alebo *Glasgow Haskell Compiler* je prekladač pre funkcionálny jazyk Haskell. Rovnako aj tento prekladač je *slobodný softvér* a jeho hlavnými vývojármi sú Simon Peyton Jones a Simon Marlow. Jednotlivé verzie GHC sú dostupné pre rôzne platformy ako Windows, väčšina distribúcií Linuxu, MAC OS X, či FreeBSD. Momentálne je samotné GHC napísané v jazyku Haskell.

3.3.1 Štruktúra

Preklad zdrojového kódu prostredníctvom GHC prebieha v niekoľkých fázach. Prvá fáza (angl. parsing) predstavuje časť prekladu, v ktorej je zdrojový kód rozložený na malé časti. Výstupom tejto fázy je abstraktný syntaktický strom. Lexikálny analyzátor v GHC je čisto funkcionálny. Ďalšou fázou je premenovávanie (angl. renaming), ktorej úlohou je rozpoznanie jednotlivých entít a vymedzenie vzťahov medzi nimi. Vstupným prvkom tejto fázy je abstraktný syntaktický strom a ako výstup je produkovaný pozmenený syntaktický strom. Ďalšou fázou je kontrola dátových typov (angl. typechecking), v ktorej prebieha kontrola, či daný program je typovo korektný. V ďalšej fáze (angl. desugaring) sa jednotlivé konštrukcie

v jazyku Haskell prevádzajú na jednoduchšie konštrukcie, ktoré sú v jazyku *Core*, ktorý tvorí medzikód. Tieto prekladané konštrukcie sa nazývajú aj *„syntactic sugar“* a proces prekladu týchto konštrukcií je potom *„desugaring“*. Tieto fázy prekladu predstavujú prednú časť prekladača.

V zadnej časti prekladača potom nastávajú rôzne optimalizácie kódu, po ktorých nasleduje generovanie cieľového kódu. V tejto fáze môže byť program buď prevedený do *byte kódu* a následne vykonaný interaktívnym interpretom alebo sa dostane do *generátoru kódu* (angl. code generator) pre preklad do strojového kódu.

Generátor kódu prevedie program do jazyka *STG*, čo je jazyk Core s väčším množstvom informácií. Následne je kód preložený do *C--*, čo je nízkoúrovňový jazyk s explicitným zásobníkom. Odtiaľto môže byť program prevedený buď pomocou natívneho generátoru kódu do programu pre konkrétnu architektúru alebo môže byť prevedený do LLVM kódu a spracovaný LLVM prekladačom. Poslednou možnosťou je, že program bude prevedený do programu v jazyku C.

3.3.2 Varovania

GHC taktiež obsahuje sadu prepínačov na zobrazovanie varovaní, ktoré sú generované počas prekladu programu. Nachádza sa tu aj sada štandardných varovaní, ktoré obyčajne indikujú potencionálne chyby v programe. Medzi takéto varovania patria napríklad:

- **-fwarn-missing-fields** – zobrazuje varovania v prípade, ak konštruktor daného poľa nie je kompletný;
- **-fwarn-missing-methods** – zobrazuje varovania v prípade, ak chýba deklarácia niektorej metódy v triede a daná trieda nemá pre túto metódu automatickú deklaráciu;
- **-fwarn-duplicate-exports** – zobrazuje varovania v prípade duplicity vstupov z externých zoznamov.

K prepínačom, ktoré zobrazujú varovania patria napríklad:

- **-Wall** – zobrazuje varovania, ktoré upozorňujú na podozrivý kód. Rovnako ako v GCC aj Clangu obsahuje tento prepínač sadu prepínačov, ktoré zobrazujú konkrétne varovania;
- **-W** – obsahuje štandardné varovania plus je doplnené o ďalšie rozširujúce prepínače;
- **-Werror** – mení varovania na chybové správy. Opakom tohto prepínača je **-Wwarn**, ktorý znení chybové hlásenia späť na varovania;
- **-w** – zakazuje zobrazenie akýchkoľvek varovaní.

Všetky prepínače, ktoré GHC podporuje sú vidieť v dokumentácií [2].

```
main =
    do putStrLn "Do you like Haskell? [yes/no]"
       answer <- getLine
       case answer of
         _     -> putStrLn "say what???"
         "yes" -> putStrLn "yay!"
         "no"  -> putStrLn "I am sorry to hear that :("
```

Po preklade tohto kódu GHC prekladačom vo verzií `7.6` sa nám zobrazí nasledujúce varovanie:

```
test3.hs:4:8:
    Warning: Pattern match(es) are overlapped
      In a case alternative:
   "yes" -> ...
   "no" -> ...
```

Varovania v GHC obsahujú na začiatku meno súboru, v ktorom nastala potenciálna chyba. Nasleduje číslo riadku s touto chybou a umiestnenie chyby v danom riadku. Na ďalšom riadku sa nachádza kľúčové slovo `Warning` a popis chyby, ktorá dané varovanie vyvolala. Pod uvedeným varovaním je zobrazené, čo daná chyba spôsobí. Rovnako aj v GHC sú všetky tieto prvky oddelené dvojbodkou.

V našom prípade varovanie oznamuje, že chyba nastala v súbore `test3.hs` na riadku číslo `4` a umiestnenie chyby v danom riadku je na pozícií číslo `8`. Príčinou chyby je, že príkazy vo vetvách „*yes*“ a „*no*“ vo funkcií `case` sa nikdy nevykonajú, pretože sú prekryté vetvou „_“.

3.4 Javac

Pri písaní tejto podkapitoly som čerpala z [7]. *Javac* alebo aj *Java compiler* je prekladač pre objektovo orientovaný programovací jazyk Java, ktorý číta jednotlivé triedy (angl. class) a rozhrania (angl. interface) a prekladá ich do bytekódových súborov. Zdrojový súbor v jazyku Java má príponu *.java* a po preklade do bytekódu má súbor príponu *.class*. Meno prekladaného súboru musí obsahovať meno, ktoré identifikuje danú triedu, pričom každá verejná (angl. public) trieda musí byť umiestnená v osobitnom súbore. Rovnako aj tento prekladač je *slobodný softvér* a je súčasťou OpenJDK projektu. Javac je napísaný v programovacom jazyku C s využitím knižníc C++.

3.4.1 Štruktúra

Pri preklade zdrojového kódu pracuje javac vo viacerých fázach. V prvej fáze (angl. parse), ktorá predstavuje lexikálny analyzátor je čítaný zdrojový kód, ktorý sa následne rozdelí na menšie časti, z ktorých sa vytvorí abstraktný syntaktický strom. V ďalšej fáze (angl. enter) sú jednotlivé symboly identifikované a priraďované do tabuľky symbolov, kde sú následné vymedzené vzájomné vzťahy medzi symbolmi a jednotlivými triedami. V nasledujúcich fázach (angl. attribute a flow) sú vykonávané rôzne kontroly ako typové kontroly, analýza jednotlivých výnimiek, ktoré sú v programe použité alebo kontrola jednotlivých premenných a ich použitie. V ďalšej fáze (angl. desugar) je abstraktný syntaktický strom transformovaný do jednoduchšej konštrukcie (angl. syntactic sugar), z ktorej je následne generovaný bytekódový súbor s príponou .class, ktorý slúži ako medzikód. Tieto fázy tvoria prednú časť prekladača.

Zadnú časť potom tvorí tzv. *Java Virtual Machine* (JVM), ktorý využíva virtuálny stroj, pomocou ktorého interpretuje medzikód, čím umožňuje spúšťanie programu na rôznych platformách.

3.4.2 Varovania

Rovnako aj v Javacu existujú prepínače, ktoré ovplyvňujú výpisy varovaní. Tieto prepínače patria do skupiny prepínačov, ktoré sa označujú ako neštandardné (angl. non-standard options). K neštandardnému prepínaču, ktorý zobrazuje varovania chýb, ktoré nastanú počas kompilácie patrí prepínač **-Xlint**.

Tento prepínač môže byť doplnený o špecifické stavy, pomocou ktorých si užívateľ určuje, ktoré varovania majú byť zobrazené. Medzi tieto stavy napríklad patria:

- **-Xlint:cast** – zobrazuje varovania, ktoré upozorňujú na nadbytočné a nepotrebné časti v zdrojovom kóde;
- **-Xlint:deprecation** – zobrazuje varovania v prípadoch, kedy je volaná tzv. „zastaralá“ metóda, ktorá by sa v nových programoch už nemala používať;
- **-Xlint:divzero** – zobrazuje varovania pri delení nulou;
- **-Xlint:empty** – zobrazuje varovania v prípade, keď podmienka „if“ je prázdna, čo znamená, že za podmienkou sa nenachádza žiadny príkaz;
- **-Xlint:fallthrough** – zobrazuje varovania pri nesprávnom použití príkazu *switch*;
- **-Xlint:finally** – zobrazuje varovania v prípade ak má návestie *finally* chybný zápis (napríklad nachádza sa v ňom príkaz *return)*;
- **-Xlint:overrides** – zobrazuje varovania v prípade ak sa zdedená trieda pokúša pokúša prepísať niektorú metódu rodičovskej triedy;
- **-Xlint:none** – zakazuje zobrazenie všetkých varovaní.

Všetky prepínače prekladača Javac sa nachádzajú v dokumentácií [5].

```
private static void Warning()
{
    final Set<Person> people = new HashSet<Person>();
    people.add(fred);
    people.add(wilma);
    people.add(barney);
    for (final Person person : people)
    {
      out.println("Person: " + ((Person) person).getFullName());
    }
}
```

Po preklade tohto kódu prekladačom Javac vo verzií `1.7` s prepínačom `-Xlint` alebo `-Xlint:cast` nám prekladač zobrazí následujúce varovanie:

```
Main.java:37: warning: [cast] redundant cast to dustin.examples.Person
        out.println("Person: " + ((Person) person).getFullName());
```

Varovania v Javacu obsahujú na začiatku meno prekladaného súboru, nasleduje číslo riadku v ktorom nastala potenciálna chyba a kľúčové slovo `warning`, za ktorými sa nachádza popis chyby, ktorá varovanie vyvolala. Všetky prvky varovania sú opäť oddelené dvojbodkou. Pod popisom varovania sa rovnako ak v Clangu nachádza časť zdrojového kódu, v ktorej je znakom `^` ukázané konkrétne umiestnenie chyby.

Varovanie v príklade potom oznamuje, že chyba nastala v súbore `Main.java` na riadku číslo `37` a príčinou varovania je zbytočne uvedený typ pri objekte `person`, pretože typ tohto objektu je určený automaticky.

Kapitola 4

Varovania, ktoré je vhodné filtrovať

V tejto kapitole je ukázaných niekoľko druhov varovaní jednotlivých prekladačov, ktoré sa dajú v istých situáciach považovať za bezproblémové. Zároveň sú tu popísané situácie, v ktorých tieto varovania nepredstavujú riziko a neohrozujú ani beh programu, preto môžu byť následne z výstupu prekladača filtrované.

4.1 Varovania spôsobené knižnicami tretích strán

Tieto varovania sa zobrazujú pri používaní knižníc hlavičkových súborov, ktoré nemôžeme meniť a ktorých používaniu sa nedá vždy vyhnúť. Tieto súbory však môžu spôsobovať rôzne druhy varovaní, ktoré je vhodné z výstupu prekladača filtrovať.

4.2 Varovania prekladačov GCC a Clang

Oba prekladače GCC aj Clang prekladajú zdrojový kód napísaný v jazykoch rodiny C a oba prekladače obsahujú veľmi podobné sady prepínačov, ktoré zobrazujú varovania, preto aj popis chýb, na ktoré varovania poukazujú sú u oboch prekladačov vo väčšine prípadov zhodné, prípadne veľmi podobné. Medzi varovania, ktoré sa na určitých miestach dajú považovať za bezproblémové patria napríklad:

`warning: unused parameter 'foo'` – toto varovanie nastáva v situácií, kedy v definícií funkcie nie sú použité všetky parametre danej funkcie. V následujúcom príklade je vidieť, že volaná funkcia má 2 parametre `param1` a `param2`, ale v definícií funkcie je použitý práve jeden parameter. Táto situácia je následne prekladačom oznámená ako varovanie, pričom priebeh programu nie je ovplyvnený a program pracuje korektne.

```
#include<stdio.h>

void funkcia(int param1, int param2){
     printf("parameter param1 ma hodnotu %d", param1);
                       //miesto, ktoré vyvolalo varovanie
     return;
}
```

```
int main(void){
    int cislo1, cislo2;
    funkcia(cislo1, cislo2);
    return 0;
}
```

`warning: comparsion between signed and unsigned integer expressions` – toto varovanie nastáva pri porovnávaní premennej so znamienkom s premennou bez znamienka, pretože rozsah hodnôt znamienkových a bezznamienkových typov sú rôzne. Premenná bez znamienka nemôže nadobúdať záporné hodnoty a naopak, pri použití rovnakých veľkostí typov so znamienkom a bez znamienka môžu premenné typu bez znamienka nadobudnúť väčšie hodnoty ako je maximálna možná hodnota znamienkového typu. V uvedenom príklade dochádza porovnanie bezznamienkovej premennej so znamienkovou, pričom beh zostáva korektný.

```
int main(void){
  unsigned int unsig_num1 = 5;
  unsigned int unsig_num2 = 0;
  int num = 0;

  if (unsig_num1 > num){ //miesto, ktoré vyvolalo varovanie
     unsig_num1 = 0;
  }

  if (unsig_num1 == unsig_num2){
     unsig_num1 = 0
  }
  return 0;
}
```

`warning: variable 'c' set but not used` – toto varovanie upozorňuje na situáciu, kedy je do nejakej premennej dosadená korektná hodnota, ale táto premenná nie je v programe použitá. V našom prípade je do premennej `c` dosadená hodnota premennej `a`, ale táto premenná nie je ďalej v kóde použitá, pričom beh programu touto situáciou nie je ovplyvnený.

```
int main(void){
   int a,b,c;
   a = 10;
   b = 15;

   if (a != b){
       c = a;
       return 0;
   }
   else{
      return 1;
   }
}
```

4.3 Varovania prekladača GHC

K varovaniam prekadača GHC, ktoré sa dajú na určitých miestach považovať za bezproblémové a je vhodné ich filtrovať, patria napríklad:

`Warning: Pattern match(es) are non-exhaustive` – toto varovanie upozorňuje na situáciu, kedy sa v programe nachádza príkaz, ktorý nie je úplný. V uvedenom príklade varovanie oznamuje, že príkaz `case` chýba štandardná vetva, ktorá zahŕňa všetky neuvedené stavy. Pokiaľ v danom programe očakávame len stavy, pre ktoré máme vytvorené vetvy, pracuje daný program korektne.

```
main =
   do putStrLn "Do you like Haskell? [yes/no]"
      answer <- getLine
      case answer of
         "yes" -> putStrln "yay!"
         "no"  -> putStrln "I'm sorry to hear that"
                  -- miesto, ktoré vyvolalo varovanie
```

`Warning: Top-level binding with no type signature` – toto varovanie upozorňuje na situáciu, kedy nie su definované dátové typy u použitých výrazov. V uvedenom príklade nie sú uvedené typy pri výrazoch `x^2` a `y^2`, pričom program napriek tomu pracuje korektne.

```
norm2 x y = sqrt(x^2 + y^2) --miesto, ktoré vyvolalo varovanie
main = print $ norm2 1 1
```

4.4 Varovania prekladača Javac

Medzi varovania prakladača Javac, ktoré sa dajú na určitých miestach považovať za bezproblémové patria napríklad:

`warning: [unchecked] unchecked cast` – toto varovanie upozorňuje na situáciu, kedy prekladač ešte v dobe prekladu nemá dostatočné informácie o všetkých typoch použitých v programe, preto sa potencionálne nebezpečenstvá, ktoré môžu vzniknúť nekorektným použitím typov oznamujú varovaním. V našom prípade program kontroluje, či objekt vrátený zo `super.clone()` je typu `Warning`. Táto informácia je však počas prekladu neznáma. Avšak za behu programu je v tejto časti vykonávaná iba opätovná kontrola typu, čím nie je ohrozený beh programu.

```
class Warning<T> {
  public Warning (T arg) {
      warning = argument
  }

  public Warning<T> clone() {
     Warning<T> clon = null;
      try {
        clon = (Warning<T>) super.clone(); //miesto, ktoré vyvolalo varovanie
      } catch (CloneNotSupportedException e) {
```

```
        throw new InternalError();
      }
  }
```

`warning: [fallthrough] possible fall-through into case` – toto varovanie upozorňuje na situáciu, kedy vo vetve príkazu `switch` chýba ukončujúci príkaz `break`, čo znamená, že príkaz v tejto vetve sa vykoná a následne sú vykonávané ďalšie príkazy až po návestie `break`. V uvedenom príklade varovanie upozorňuje, že pri zadaní hodnoty `1` sa vykonajú obe vetvy príkazu `switch`, pokiaľ je toto zámerom programu, pracuje tento program korektne.

```
switch(x){
    case 1:
        System.out.println("Výstup"); //miesto,ktoré vyvolalo varovanie
    case 2:
        System.out.println("Výstup");
    default:
        System.out.println("Koniec");
        break;
}
```

Kapitola 5

Návrh programového filtru

Cieľom tejto kapitoly je popis návrhu aplikácie, ktorej úlohou je filtrovať varovania, ktoré si užívateľ neželá zobrazovať. Taktiež tu budú rozobrané problémy, ktoré budú v implementácií programu riešené a možnosť implementovať program tak, aby bolo jednoduché pridávať implementácie filtrovania varovaní aj pre ďalšie prekladače.

5.1 Úloha programového filtru

Filter varovaní je program, ktorého úlohou je selektívne filtrovanie varovaní, čím sa sprehľadní výstup prekladača. Cieľom programu nie je filtrovanie všetkých varovaní, ale len tých, ktoré užívateľ považuje za bezproblémové a neželá si tieto varovania zobrazovať. K filtrovaniu varovaní dochádza len z výstupu prekladača, teda bez zásahu do zdrojového kódu.

5.2 Spracovanie vstupných dát

Program bude prijímať ako vstupné dáta dve povinné zložky. Prvou zložkou bude výsledok prekladu zdrojových súborov, z ktorého sa budú filtrovať neželané varovania. Medzi podporované prekladače budú patriť `GCC`, `Clang`, `Javac` a `GHC`.

Druhou povinnou zložkou vstupných dát programového filtru bude súbor, ktorý bude obsahovať varovania, ktoré si užívateľ neželá zobrazovať.

Z dôvodu, že výstupy a najmä štruktúra varovaní je v jednotlivých prekladačoch odlišná (ako je popísané v kapitole 3) musí program najprv rozpoznať prekladač, ktorého varovania budú filtrované. Po rozpoznaní prekladača bude program prechádzať výstup prekladača a hľadať varovania. Po identifikovaní varovania program porovná nájdené varovanie s varovaniami, ktoré sa nachádzajú v súbore s neželanými varovaniami. Ak dôjde k zhode nájdeného varovania vo výstupe prekladača s varovaním v súbore s neželanými varovaniami, bude toto varovanie z výstupu prekladača odstránené.

5.3 Spôsoby filtrovania varovaní

Po identifikácií varovaní vo výstupe prekladača program rozdelí nájdené varovania podľa regulárnych výrazov na časti, ktoré budú porovnávané s varovaniami, ktoré sú zadané v súbore s neželanými varovaniami. Varovanie môže byť rozdelené až na 3 časti, záleží o aký typ

prekladača, prípadne verziu prekladača ide. Následne porovnávanie varovaní s varovaniami v súbore s neželanými varovaniami môže byť na základe:

- `Pozície varovania v zdrojovom kóde` – v tomto prípade budú porovnávané so súborom s neželanými varovaniami časti varovaní, ktoré obsahujú hodnotu, ktorá udáva aká je pozícia varovaní v zdrojovom kóde. Ako je vidieť na príklade:

  ```
  test.c:5:9: warning: unused variable ‘b’
  ```

  V tomto prípade (varovanie prekladača GCC verzie `gcc-4.5`) je pozícia varovania určená číslom riadku, na ktorom sa varovanie nachádza a hodnotou, ktorá popisuje umiestnenie varovania v rámci riadku. Užívateľ bude môcť určiť pozíciu neželaného varovania buď len hodnotou čísla riadku alebo rozšíreným popisom aj s identifikačným číslom, určujúcim pozíciu.

- `Popisu varovania` – v tomto prípade bude program porovnávať časti varovaní, ktoré popisujú situáciu, na ktorú varovanie poukazuje. Program si taktiež bude vedieť poradiť s varovaniami ako je napríklad:

  ```
  test.c:5:9: warning: unused variable ‘b’
  ```

  V tomto type varovaní je vhodné, aby program filtroval varovania pre všetky premenné a nie len pre konkrétnu (akoby tomu bolo, keby sa v súbore s neželanými varovaniami nachádzal popis varovania v tvare `unused variable ‘b’`), preto v prípade, kedy bude užívateľ požadovať filtrovanie daného varovania pre všetky premenné, nahradí v neželanom varovaní konkrétnu premennú znakom „`*`“. Po rozpoznaní tohto znaku v súbore s neželanými varovaniami nebudú popisy varovaní porovnávané ako celok, ale program rozdelí popis varovania na menšie časti, u ktorých sa bude zisťovať zhoda. V súbore s neželanými varovaniami by potom takto parametrizované varovanie vyzeralo ako:

  ```
  unused variable ’*’
  ```

- `Typu varovania` – v tomto prípade bude program porovnávať časť varovania, ktorá udáva, akého typu dané varovanie je. Ako je vidieť na príklade:

  ```
  test.c:7:2: warning: too many arguments for format [-Wformat-extra-args]
  ```

  Po zadaní formátu varovania do súboru s neželanými varovaniami, budú odstránené z výstupu prekladača všetky varovania daného formátu (ktorý by bol v danom príklade `-Wformat-extra-args`).

5.4 Výsledky programu

Výstupom programu bude výsledok prekladu zdrojových súborov, z ktorého budú odstránené neželané varovania. Program bude taktiež vykonávať rôzne kontroly, ktoré rozhodnú o tom, či program bude pokračovať ďalej alebo sa ukončí chybovým hlásením. Pri situácií, kedy sa programu nepodarí identifikovať prekladač alebo užívateľ zadá prekladač, ktorý nie je podporovaný, bude program ukončený chybovým hlásením. V situácií, kedy sa vo výstupe prekladaných súborov nebude nachádzať žiadne varovanie alebo žiadne z varovaní sa nebude zhodovať s neželanými varovaniami v užívateľskom súbore, bude výstup programu zhodný so vstupom.

5.5 Filtrovanie varovaní prekladača GCC

U prekladača GCC bude program podporovať preklad súborov jazyka C. Ako je vidieť v časti 3.1.2, varovania jednotlivých verzií prekladača GCC sú odlišné, preto je nutné implementovať program tak, aby bol schopný filtrovať varovania viacerých verzií prekladača. Konkrétne program bude filtrovať varovania vo verziách `gcc-4.4` – `gcc-4.8`.

Pri filtrovaní varovaní prekladača GCC bude program prechádzať výstup prekladača a pomocou regulárneho výrazu bude v tomto výstupe hľadať varovania. Po nájdení varovania bude program na jeho základe identifikovať o akú verziu prekladača GCC ide. Pre každú verziu prekladača bude vytvorený regulárny výraz, pomocou ktorého bude prebiehať identifikácia.

Po nájdení varovania vo výstupe prekladača, program toto varovanie rozdelí podľa regulárneho výrazu, ktorý bude príslušný k danej verzií prekladača a následne budú tieto časti porovnávané so súborom, ktorý obsahuje neželané varovania.

Zobrazenie pozícií varovania vo výstupe prekladača GCC sa v jednotlivých verziách líši. Ako je vidieť na príklade varovania z verzie `gcc-4.4` je pozícia varovania definovaná len prostredníctvom riadku, v ktorom sa varovanie nachádza, preto aj filtrovanie varovaní na základe ich pozície v tejto verzií prekladača bude spočívať v porovnaní hodnoty v súbore s neželanými varovaniami s hodnotou vo výstupe.

```
test.c:7: warning: too many arguments for format
```

Od verzie `gcc-4.5` je rozšírená hodnota určujúca pozíciu varovania o časť, ktorá identifikuje nie len riadok, kde sa varovanie nachádza, ale aj pozíciu vrámci tohto riadku.

```
test.c:7:2: warning: too many arguments for format
```

Pri tomto type popisu pozície varovaní sa bude môcť užívateľ rozhodnúť, či pozíciu varovania určí len číslom riadku, alebo oboma hodnotami, pričom program bude umožňovať filtrovanie oboma spôsobmi.

Pri varovaniach, ktoré sú určené prostredníctvom popisu bude program presne porovnávať popis varovania s obsahom súboru s neželanými varovaniami, prípadne bude možné varovania parametrizovať prostredníctvom znaku „`*`“.

Keďže typ varovania sa objavuje vo výstupe prekladača GCC až od verzie `gcc-4.6` bude program filtrovať varovania určené prostredníctvom typu len pri použití prekladača GCC vo verziách `gcc-4.6` – `gcc 4.8`.

Problémy programu pre filter prekladača GCC

Pri implementácií programového filtru pre prekladač GCC bude nutné riešiť niekoľko problémov. Jedným z problémov, ktorý bude v implementácií programu riešený sú časti výstupu prekladača, ktoré sa po filtrovaní varovaní môžu zdať v tomto výstupe nadbytočné. Majme napríklad výstup prekladača:

```
test2.c: In function ‘pocitaj’:
test2.c:5:32: warning: unused parameter ‘b’ [-Wunused-parameter]
test2.c: In function ‘main’:
test2.c:14:5: warning: too many arguments for format [-Wformat-extra-args]
test2.c:14:11: warning: ‘i’ is used uninitialized in this function
       [-Wuninitialized]
test2.c:15:12: warning: ‘o’ is used uninitialized in this function
       [-Wuninitialized]
test2.c:15:12: warning: ‘u’ is used uninitialized in this function
       [-Wuninitialized]
```

V prípade, že užívateľ by požadoval filtrovanie všetkých varovaní „`unused parameter`“, zostal by vo výstupe riadok „`test2.c: In function ‘pocitaj’`“, ktorý sa dá po filtrácií varovania považovať za nadbytočný. Preto je nutné, aby pri filtrovaní varovaní boli odstránené aj „nadbytočné“ riadky informatívneho charakteru. Pri riešení tejto situácie bude program obsahovať regulárny výraz pre identifikáciu týchto častí. Po ich rozpoznaní program bude kontrolovať, či sa za touto časťou budú vypisovať ďalšie informácie. V prípade, že sa na výstup budú vypisovať aj ďalšie informácie budú tieto riadky zobrazené, v opačnom prípade budú z výstupu prekladača filtrované.

Ďalším problémom, ktorý bude program implementujúci filter pre prekladač GCC riešiť, sú varovanie verzie `gcc-4.8`, ktoré môžu ležať na viacerých riadkoch:

```
test2.c:14:5: warning: too many arguments for format [-Wformat-extra-args]
     printf("Skuska vypis",i);
     ^
```

Riešenie tejto situácie sa opiera o fakt, že len nové informácie vo výstupe prekladača začínajú menom prekladaného súboru, pričom dodatočné informácie popisujúce varovanie na ďalších riadkoch túto časť neobsahujú. Preto program po rozpoznaní neželaného varovania kontroluje následujúce riadky prostredníctvom regulárneho výrazu, či obsahujú časť s menom prekladaného súboru, na základe čoho sa zisťuje, či dané riadky sú súčasťou neželaného varovania a budú filtrované.

Posledným problémom, ktorý bude pri prekladači GCC je situácia, kedy niektoré varovania vo výstupe obsahujú dodatočné informácie v podobe poznámky ako je to vidieť v následujúcom prípade:

```
test5.c: In function ‘main’:
test5.c:495:3: warning: passing argument 2 of ‘change_array’ from
               incompatible pointer type [enabled by default]
test5.c:99:5: note: expected ‘char **’ but argument is of type ‘int (*)[2]’
```

Táto situácia bude riešená spôsobom, kedy po označení varovania za neželané sa bude prostredníctvom regulárneho výrazu kontrolovať, či sa za týmto varovaním nenachádza dodatočná poznámka, ktorá s týmto varovaním súvisí. Po jej identifikáciu, bude táto poznámka spolu s varovaním z výstupu prekladača odstránená.

5.6 Filtrovanie varovaní prekladača Clang

U prekladača Clang bude program rovnako podporovať preklad súborov jazyka C a bude umožňovať filtrovať varovania vo verziách `clang 2.9` – `clang 3.2`. Program bude prechádzať výstupom prekladača po riadkoch a pomocou regulárneho výrazu hľadať varovania, ktoré sa v tomto výstupe nachádzajú. Po nájdení varovania program opäť rozdelí toto varovanie podľa regulárneho výrazu na niekoľko častí, ktoré sa budú porovnávať s varovaniami zadanými v súbore s neželanými varovaniami. Rovnako ako v prekladači GCC môže byť varovanie rozdelené až na 3 časti.

Ako bolo spomínané v časti 3.2.2, tak pozícia varovaní v prekladači Clang je v oboch verziách definovaná dvomi hodnotami, ktorými sú číslo riadku, ktoré varovanie spôsobilo a hodnota pozície v tomto riadku. Rovnako ako v prekladači GCC aj tu sa bude môcť užívateľ rozhodnúť, či pozíciu varovania určí iba číslom riadku alebo oboma hodnotami.

Pri filtrovaní varovaní určených prostredníctvom ich popisu bude opäť možné filtrovať tieto varovania na základe ich celého popisu, prípadne môže užívateľ tieto varovania parametrizovať pomocou znaku „`*`“.

Keďže v prekladači Clang neobsahujú všetky varovania vo svojom popise aj typ varovania, program rozdelí varovanie najprv na dve časti a potom dodatočne kontroluje, či sa v popise varovania nachádza aj jeho typ, na základe ktorého bude môcť byť toto varovanie filtrované.

Problémy programu pre filter prekladača Clang

Rovnako aj pri implementácií filtru varovaní pre prekladač Clang bude musieť program riešiť niekoľko problémových situácií. Prvou takouto situáciu je, že varovania prekladača Clang ležia na niekoľkých riadkoch ako je vidieť v časti 3.2.2. Program rovnako ako pri varovaniach prekladača GCC bude využívať faktu, že dodatočné informácie, ktoré sú súčasťou varovania neobsahujú na začiatku riadku informáciu o mene prekladaného súboru, ktorú program bude overovať pomocou regulárneho výrazu a následne zisťovať, ktoré časti výstupu prekladača patria k neželanému varovaniu.

Program sa taktiež bude starať o to, aby v prípade filtrovania posledného varovania, nebol filtrovaný aj posledný riadok výstupu, ktorým je počítadlo varovaní a chybových hlásení, ktoré nezačína menom prekladaného súboru. Táto situácia bude ošetrená spôsobom, že program bude kontrolovať pomocou regulárneho výrazu, či daný riadok, nie je práve daným riadkom obsahujúci dané informácie a podľa toho vyhodnotí, či je riadok súčasťou neželaného varovania.

Ďalším problémom, s ktorým sa bude musieť program vysporiadať je, že niektoré varovania v prekladači Clang od verzie `clang 3.0` obsahujú dodatočné informácie v podobe:

```
test2.c:15:19: warning: variable 'u' is uninitialized when used here
          [-Wuninitialized]
    pocitaj(i, o, u);
                  ^
test2.c:12:14: note: initialize the variable 'u' to silence this warning
    int i,o,u;
             ^
              = 0
```

Po odstránení varovania by bola táto poznámka vo výstupe prekladača nadbytočná, preto bude vhodné tieto poznámky spolu s varovaniami z výstupu filtrovať. Ako je vidieť v prí-

klade poznámka súvisiaca s varovaním odkazuje na inú pozíciu v zdrojovom kóde ako varovanie, takže nebude možné odstrániť varovanie spolu s poznámkou prostredníctvom pozície v zdrojovom kóde. Rovnako poznámka neobsahuje v popise typ ako je to v popise varovania, preto program túto situáciu bude riešiť tak, že po filtrácií varovania bude pomocou regulárneho výrazu kontrolované, či sa bezprostredne za varovaním nenachádza aj poznámka súvisiaca s týmto varovaním. Pokiaľ program túto poznámku identifikuje, bude z výstupu prekladača odstránená.

Posledným problémom, s ktorým si bude musieť program pri filtrovaní varovaní prekladača Clang porovnať je počítadlo chybových hlásení a varovaní na konci výstupu prekladača:

```
test.c:6:12: warning: '.*' specified field precision is missing a matching
           'int' argument
        printf("%.*d");
                ~~~~
test.c:7:24: warning: data argument not used by format string
          [-Wformat-extra-args]
        printf("Skuska vypis",i);
               ~~~~~~~~~~~~~~ ^
test.c:8:9: error: use of undeclared identifier 'neco'
        printf(neco);
               ^
2 warnings and 1 error generated.
```

Po filtrácií nie je vhodné aby sa na konci výstupu zobrazoval pôvodný počet varovaní. Táto situácia bude riešená spôsobom, že v program bude počítať aktuálny počet varovaní, ktoré budú zobrazené a následne hodnotu vo výstupe prepisovať. Program sa rovnako bude starať o prípady, kedy nebude zobrazené žiadne varovanie. V tomto prípade program bude zobrazovať len počet chybových hlásení. Prípadne keď bude zobrazené len jedno varovanie bude prepisovať slovo `warnings` na jeho tvar v jednotnom čísle `warning`. Pri situácií kedy budú filtrované všetky časti výstupu prekladača, nebude ani tento riadok vo výstupe zobrazovaný.

5.7 Filtrovanie varovaní prekladača Javac

U prekladača Javac bude program podporovať preklad súborov jazyka Java do bytekódových súborov. Program bude rovnako prechádzať výstupom prekladača po riadkoch a pomocou regulárneho výrazu hľadať varovania. Po nájdení varovania vo výstupe prekladača, program následne toto varovanie rozdelí na 3 časti, ktoré budú porovnávané s hodnotami zadanými užívateľom v súbore s neželanými varovaniami.

Pozícia varovania je vo výstupe prekladača Javac definovaná len prostredníctvom čísla riadku, čo znamená, že program bude filtrovať varovania na základe ich pozície z výstupu prekladača len pomocou tejto jednej hodnoty.

U varovaní, ktoré bude chcieť užívateľ filtrovať na základe ich popisu je možné zapísať do súboru s varovaniami opäť dvomi spôsobmi. Prvým spôsobom je zadať do súboru celý popis varovania, ktorý bude porovnávaný ako celok s nájdeným varovaním vo výstupe prekladača. Ďalším spôsobom bude možnosť parametrizovať popis varovania pomocou znaku „*“, kedy bude možné filtrovať všetky varovania rovnakého typu, ktoré v tele popisu obsahujú rôzne premenné.

Varovania prekladača Javac obsahujú vo svojom tele položku, ktorá popisuje formát daného varovania. Pomocou tejto zložky bude možné filtrovať viacero varovaní, ktoré zdieľajú rovnaký formát.

Problémy programu pre filter prekladača Javac

Pri implementácií filtru varovaní pre prekladač Javac je taktiež niekoľko situácií, s ktorými sa bude musieť program vysporiadať. Jednou z týchto situácií je fakt, že varovania vo výstupe opäť ležia na niekoľkých riadkoch. Riešenie tejto situácie sa obdobne ako u prekladačov GCC a Clang bude opierať o fakt, že doplňujúce informácie k varovaniu, ktoré ležia na ďalších riadkoch neobsahujú informácie o názve prekladaného súboru. Program taktiež pomocou regulárneho výrazu bude kontrolovať, či daný riadok neobsahuje údaje o počte chybových hlásení a varovaní, ktorý sa nachádza na konci výstupu prekladača a neobsahuje údaje o mene prekladaného súboru, aby nedošlo k situácií, že tento riadok bude filtrovaný spolu s nechceným varovaním.

Ďalším problémom, s ktorým si bude musieť program poradiť je, že vo výstupe prekladača sa na konci výstupu nachádza informácia o počte varovaní prípadne chybových hlásení odhalených pri preklade súborov ako je vidieť na príklade:

```
User.java:3: warning: [deprecation] deprecatedMethod() in Main has been
  deprecated
      Main.deprecatedMethod();
          ^
1 warning
```

Opäť je nežiadúce aby po filtrovaní varovaní bol na konci programu zobrazovaný pôvodný počet varovaní. Program preto počas behu bude priebežne počítať koľko varovaní bude zobrazených a táto výsledná hodnota bude následne zobrazená vo výstupe. Program taktiež bude kontrolovať, či sa zobrazí jedno alebo viacej varovaní a na základe toho vyhodnotí, či kľúčové slovo `warning` bude v jednotnom alebo množnom čísle. V prípade, že vo výstupe sa nebude nachádzať žiadne varovanie, prípadne chybové hlásenie, bude tento údaj z výstupu odstránený úplne.

5.8 Filtrovanie varovaní prekladača GHC

U prekladača GHC bude program podporovať preklad súborov jazyka Haskell. Program bude prechádzať jednotlivé riadky výstupu prekladača a pomocou regulárneho výrazu hľadať varovania.

```
TestHaskel3.hs:9:8:
    Warning: Pattern match(es) are non-exhaustive
             In a case alternative:
                 Patterns not matched:
                     []
                     (GHC.Types.C# #x) : _ with #x ‘notElem‘ [’y’, ’n’]
                     [GHC.Types.C# ’y’]
                     (GHC.Types.C# ’y’) : ((GHC.Types.C# #x) : _)
                     with
                     #x ‘notElem‘ [’e’]
```

...

Ako je vidieť na príklade varovania prekladača GHC môžu mať informácie o pozícií varovania umiestnené na inom riadku ako je popis varovania, či samotné kľúčové slovo `Warning`, čo znamená, že program v čase keď hodnotí riadok s pozíciou varovania, ešte nevie či daná pozícia súvisí s varovaním. Preto program pri prechode výstupom prekladača bude primárne hľadať pomocou regulárneho výrazu pozície v zdrojovom kóde. Po ich identifikácií program následne skontroluje celý, prípadne ďalší riadok, či sa jedná o varovanie. Pokiaľ bude varovanie identifikované, bude toto varovanie analyzované a porovnané so súborom s neželanými varovaniami.

Jednou z možností filtrovania varovaní prekladača GHC môže byť filtrovanie na základe pozície v zdrojovom kóde, ktorá je u tohto prekladača popísaná hodnotou riadku, v ktorom bolo varovanie spôsobené a hodnotou pozície vrámci tohto riadku. Užívateľ bude môcť opäť varovania určiť buď iba hodnotou riadku alebo kombináciou oboch hodnôt.

Filtrovanie varovaní bude taktiež možné prostredníctvom ich popisu, ktorý sa nachádza za kľúčovým slovom `Warning`. Opäť bude možné filtrovať varovanie, buď na základe celého popisu alebo sa budú filtrovať všetky varovania rovnakého typu bez ohľadu na to, aké obsahujú premenné pri parametrizácií prostredníctvom znaku „`*`“.

Varovania prekladača GHC vo výstupe neobsahujú hodnotu určujúcu formát daného varovania, preto filtrovanie na základe formátu v tomto prekladači nebude možný.

Problémy programu pre filter prekladača GHC

Rovnako aj pri prekladači GHC bude program musieť riešiť niekoľko problémových situácií. Majme výstup prekladača:

```
[1 of 1] Compiling Main             ( TestHaskell3.hs, TestHaskell3.o )

TestHaskell3.hs:5:1: Warning: Defined but not used: ‘norm2’

TestHaskell3.hs:5:1:
    Warning: Top-level binding with no type signature:
               norm2 :: forall a. Num a => a -> a -> a

TestHaskell3.hs:10:8:
    Warning: Pattern match(es) are non-exhaustive
             In a case alternative:
                 Patterns not matched:
                     []
                     (GHC.Types.C# #x) : _ with #x ‘notElem‘ [’y’, ’n’]
                     [GHC.Types.C# ’y’]
                     (GHC.Types.C# ’y’) : ((GHC.Types.C# #x) : _)
                     with
                     #x ‘notElem‘ [’e’]
                     ...
Linking TestHaskell3 ...
```

Ako je vidieť na danom príklade vo výstupe prekladača GHC sa objavuje varovanie, ktoré má kľúčové slovo `Warning` na rovnakom riadku ako je pozícia. A rovnako sa tu objavuje

varovanie, ktoré obsahuje toto kľúčové slovo na ďalšom riadku. Program bude túto situáciu riešiť, že pre oba typy varovaní bude obsahovať regulárne výrazy, pomocou ktorých bude tieto varovanie identifikovať a následne rozdeľovať na jednotlivé časti.

Ďalším problémom je fakt, že varovania, prekladača GHC obsahujú vo výstupe rôzny počet riadkov. Program sa bude riadiť faktom, že jednotlivé varovania sú oddelené prázdnymi riadkami, takže po identifikácií neželaného varovania bude toto varovanie filtrované až po najbližší prázdny riadok, vrátane neho, ktorý bude identifikovaný prostredníctvom regulárneho výrazu.

Program rovnako dbá o to, aby pri prípadnom filtrovaní posledného riadku nebol spolu s týmto varovaním filtrovaný aj posledný riadok výstupu `Linking...`, ktorý od posledného varovania nie je oddelený medzerou. Túto situáciu program rieši tým, že jednotlivé riadky výstupu kontroluje, či sa nejedná o posledný riadok. Táto kontrola prebieha opäť pomocou regulárneho výrazu.

5.9 Rozšíriteľnosť programu

Program bude implementovaný tak, aby bolo možné a najmä jednoduché kedykoľvek pridať implementáciu filtrovania varovaní pre ďalšie prekladače. Program bude implementovaný tak, že funkcie, ktorých úlohou je filtrácia varovaní pre konkrétny prekladač, budú implementované v externom súbore a do hlavného programu budú importované. Úlohou hlavného programu bude zistiť výstup, ktorého prekladača má na vstupe a na základe jeho identifikácie rozhodne v podmienke, ktorá funkcia bude volaná.

Program bude rozpoznávať prekladač, ktorého výstup sa nachádza na jeho vstupe dvomi spôsobmi. Prvým spôsobom rozpoznávania bude zo súboru s varovaniami, ktoré si užívateľ neželá zobraziť. Tento spôsob bude z hľadiska rozšíriteľnosti vhodnejší, keďže použitý prekladač bude zadaný užívateľom. V súbore s neželanými varovaniami bude prvý riadok vyhradený pre možnosť zadania použitého prekladača, program bude po spustení tento riadok kontrolovať a pri identifikácií mena prekladača v tomto riadku skontroluje, či je tento prekladač podporovaný v podmienkovej vetve. Potom v situácií keď užívateľ bude chcieť rozšíriť program o ďalšie prekladače, bude stačiť do hlavného programu naimportovať funkcie, ktoré implementujú filtrovanie a rozšíriť podmienku podporovaných prekladačov o vetvu, ktorá sa vykoná po identifikácií prekladača.

V prípade, že program neidentifikuje prekladač zo súboru s neželanými varovaniami, bude možné rozpoznať prekladač ná základe obsahu jeho výstupu, ktorý bude implementovaný v externom súbore. Pri identifikácií prekladača sa bude vychádzať zo skutočnosti, že výstup každého prekladača obsahuje unikátne znaky na základe, ktorých je možné daný prekladač identifikovať. Konkrétne u prekladačov použitých v tejto práci bude identifikácia nasledovná. U prekladača Javac, ktorý prekladá ako jediný z použitých prekladačov súbory s koncovkou `.java`, bude program identifikovať prekladač pomocou mena prekladaného súboru, ktorý obsahuje túto koncovku. Program bude kontrolovať túto skutočnosť pomocou regulárneho výrazu. Rovnaký postup je aj pri prekladači GHC, ktorý ako jediný prekladá súbory jazyka Haskell, ktoré majú koncovku `.hs`.

U prekladačov GCC a Clang však tento postup nie je možný, pretože oba prekladače prekladajú súbory rodiny jazyka C. Keďže u prekladača GCC sa od verzie gcc-4.8 nachádzajú varovania, ktoré ležia na niekoľkých riadkoch, nie je možné rozlíšenie prekladačov ani na základe ich varovaní. Preto bude program rozlišovať tieto prekladače na základe reťazca `In function`, ktorý sa nachádza len vo výstupe prekladača GCC a oznamuje, ktorým funkciám dané varovania, prípadne chybové hlásenia prislúchajú. Takže v prípade, že

program pomocou regulárneho výrazu identifikuje, že prekladaný súbor, je súborom jazyka C, následne bude hľadať pomocou regulárneho výrazu reťazec `In function` a na základe zhody, prípadne nezhody identifikuje, či sa jedná o prekladač Clang alebo GCC.

V prípade, že užívateľ bude chcieť pridať nový prekladač, je nutné aby v tomto prípade do externého súboru implementoval funkciu, pomocou ktorej bude možné identifikovať pridaný prekladač.

Kapitola 6

Implementácia programu

V tejto kapitole je popísaná implementácia jednotlivých častí programu, ktorý filtruje užívateľsky neželané varovania z výstupu prekladačov `GCC`, `Clang`, `Javac` a `GHC`.

6.1 Štruktúra programu

Programový filter je implementovaný v skriptovacom programovacom jazyku *Python 3* a je zložený z niekoľkých súborov:

- `parser.py` – hlavný súbor, ktorým je program spustený a z ktorého sú volané ďalšie súbory;
- `gccFilter.py` – súbor, v ktorom je implementované filtrovanie varovaní prekladača `GCC`;
- `clangFilter.py` – súbor, v ktorom je implementované filtrovanie varovaní prekladača `Clang`;
- `javacFilter.py` – súbor, v ktorom je implementované filtrovanie varovaní prekladača `Javac`;
- `ghcFilter.py` – súbor, v ktorom je implementované filtrovanie varovaní prekladača `GHC`;
- `recognizer.py` – súbor, v ktorom je implementované rozpoznávanie prekladača na základe jeho výstupu.

6.2 Formát užívateľského súboru

Súčasťou vstupných parametrov je užívateľský súbor, ktorý obsahuje všetky varovania, ktoré užívateľ považuje za bezproblémové a neželá si tieto varovania zobrazovať vo výstupe prekladača. Jednotlivé varovania sú od seba v súbore oddelené prázdnym riadkom. Program nie je case sensitive, čiže pri zápise varovaní nezáleží na tom, či sú zapísané veľkými alebo malými písmenami.

V implementácií programu musí byť tento súbor vo formáte `.txt` a je zadaný pri spustení programu ako povinný parameter. Z dôvodu jednoduchej rozšíriteľnosti programu o ďalšie prekladače, môže prvý riadok súboru obsahovať reťazec „`compiler: [meno prekladača]`“,

na základe ktorého program jednoducho zistí, aký prekladač užívateľ použil. Následne program môže jednoducho zavolať funkciu, ktorá filtruje varovania pre zvolený prekladač.

Jednotlivé varovania, ktoré je neželané zobrazovať, môžu byť v užívateľskom súbore zapísané niekoľkými spôsobmi. Prvým spôsobom zápisu je, že užívateľ do súboru zadá časť varovania, ktorá sa nachádza vo výpise za kľúčovým slovom „`warning:`“. Napríklad ak chceme filtrovať varovanie z výstupu prekaldača GCC ako:

```
test2.c:4:2: warning: too many arguments for format [-Wformat]
```

zadáme do užívateľského súboru následujúcu časť varovania:

```
too many arguments for format
```

Program taktiež umožňuje filtrovanie varovaní, ktoré obsahujú premenné. Príkladom je následujúce varovanie:

```
test3.c:4:6: warning: unused variable ‘a’ [-Wunused-variable]
```

V prípade, že užívateľ bude chcieť filtrovať všetky varovania `unused variable`, ktoré obsahujú rôzne premenné, stačí zadať do užívateľského súboru popis varovania následne:

```
unused variable '*'
```

Ďalším spôsobom zápisu neželaných varovaní je možnosť zadať do užívateľského súboru polohu varovania, na základe čísla riadku, ku ktorému sa varovania vzťahujú. Tento spôsob je vhodný ak užívateľ chce filtrovať varovania, ktoré sa vzťahujú k určitému miestu v zdrojovom kóde a zároveň užívateľ považuje toto miesto za korektné. Do súboru je možné zadať len číslo riadku, pričom program bude filtrovať všetky varovania, ktoré sa k tomuto riadku viažu. U prekladačov, ktoré okrem čísla riadku zobrazujú aj pozíciu v rámci riadku, môže, ale nemusí užívateľ zadať obe hodnoty oddelené bodkočiarkou, ako je to vo výstupe prekladača. Napríklad opäť vo varovaní prekladača GCC:

```
test2.c:4:2: warning: too many arguments for format [-Wformat]
```

Ak by užívateľ chcel odstrániť všetky varovania, ktoré sa nachádzajú na štvrtom riadku stačilo by do užívateľského súboru zadať hodnotu `4`. Pokiaľ by užívateľ chcel filtrovať len varovania na pozícií `4:2`, zadá do súboru hodnotu v tomto tvare.

U prekladačov, ktoré u jednotlivých varovaní zobrazený aj typ varovania, je možné filtrovať jednotlivé varovania na základe ich typu. Opäť na príklade varovania prekladača GCC:

```
test2.c:4:2: warning: too many arguments for format [-Wformat]
```

Toto varovanie je typu `Wformat`, ak by si užívateľ želal filtrovať všetky varovania daného typu, zadá do súboru danú hodnotu typu varovania.

6.3 Priebeh programu

Program je spustený súborom `parser.py` a spúšťací príkaz má tvar:

```
[preklad súboru/ov] 2>&1 | ./parser.py [súbor s neželanými varovaniami]
```

(napríklad: `gcc -Wall -Wextra subor.c 2>&1 | ./parser.py varovania`)

Program prijíma ako vstup dva povinné parametre. Prvým povinným parametrom je výstup prekladača, ktorý program prijíma ako vstup cez „rúru“ (angl. pipe). Druhým povinným parametrom je textový súbor vo formáte `.txt`, ktorý obsahuje varovania, ktoré si užívateľ neželá vo výstupe prekladača zobrazovať.

Po spustení program najprv skontroluje, či boli zadané oba vstupné parametre, teda či užívateľ zadal ako parameter súbor s nechcenými varovaniami, a či na vstup programu prichádzajú dáta z „rúry“. Následne program otvorí súbor s neželanými varovaniami, prečíta prvý riadok a pokúsi sa identifikovať prekladač. V prípade, že program úspešne identifikuje prekladač, porovná ho s podporovanými prekladačmi. Ak sa identifikovaný prekladač zhoduje s niektorým s podporovaných prekladačov, program následne volá funkciu, ktorej úlohou je filtrovanie neželaných varovaní pre daný prekladač, a ktorá je implementovaná v externom súbore. Ak sa identifikovaný prekladač z užívateľského súboru nezhoduje s žiadnym z podporovaných prekladačov, program sa skončí chybovým hlásením.

V prípade, že sa programu nepodarí identifikovať prekladač z užívateľského súboru s varovaniami, je volaná funkcia `recognize_compiler()`, ktorá je implementovaná v externom súbore `recognizer.py`, a ktorej úlohou je rozpoznať prekladač, ktorého výstup je vstupom programu.

6.4 Dodatočná identifikácia prekladača

V situácií, kedy užívateľ nezadá do súboru s neželanými varovaniami identifikáciu prekladača, je následne z hlavného programu volaná funkcia `recognize_compiler()`. Úlohou tejto funkcie je rozpoznať použitý prekladač, na základe jeho výstupu.

Funkcia „`recognize_compiler()`“ v implementácií identifikuje prekaldače `GCC, Clang, Javac a GHC`. Pre jednotlivé prekladače sú vytvorené regulárne výrazy, aby bolo možné tieto prekladače podľa ich výstupov identifikovať. Pre prekladač Javac je tento regulárny výraz v tvare:

```
(.+).java:(.+)
```

Pre identifikáciu prekladača GHC má tento regulárny výraz tvar:

```
(.+).hs:(.+)
```

U prekladačov GCC a Clang je najprv kontrolované pomocou regulárneho výrazu, či je prekladaný súbor súborom rodiny jazyka C. Tento regulárny výraz je v tvare:

```
(.+).c(.*):(.+)
```

Pokiaľ je prekladaný súbor súborom rodiny jazyka C, je vo výstupe prekladača pomocou regulárneho výrazu kontrolované, či sa vo výstupe nachádza reťazec prekladača GCC `In function`, kedy je regulárny výraz v tvare:

```
(.+).c: In function (.+)
```

Po úspešnej identifikácií je identifikátor pre konkrétny prekladač vrátený do hlavného programu, odkiaľ je následne volaná príslušná funkcia pre filtrovanie varovaní. V situácií, kedy sa funkcií prekladač identifikovať nepodarilo, je program ukončený chybovým hlásením.

6.5 Filtrovanie varovaní prekladača GCC

Implementácia filtrovania varovaní pre prekladač GCC sa nachádza v súbore `gccFilter.py` a je implementovaná vo funkcií „`parseGccWarning()`“, ktorá je volaná z hlavného súboru po identifikácií prekladača.

Funkcia má na vstupe dva parametre. Prvým parametrom je súbor, ktorý obsahuje neželané varovania a druhým parametrom je výstup prekladača. Program prechádza jednotlivé riadky výstupu prekladača a zisťuje, či daný riadok nie je varovanie. Následne pomocou regulárnych výrazov identifikuje verziu prekladača. Pre každú verziu prekladača má program vytvorené regulárne výrazy. Pre verziu `gcc-4.4`, ktorej varovania vo výstupe obsahujú názov prekladaného súboru, číslo riadku, ktorý varovanie spôsobil, kľúčové slovo `warning` a popis varovania je regulárny výraz v tvare:

```
(.+):(\d+): warning: (.+)
```

U verzie `gcc-4.5`, ktorá je vo výpise varovaní doplnená o hodnotu určujúcu pozíciu varovania vrámci riadku, má regulárny výraz tvar:

```
(.+):(\d+):(\d+): warning: (.+)
```

Pre verzie `gcc-4.6` - `gcc-4.8` má prvý riadok varovania rovnaký tvar a oproti predchádzajúcim varovaniam je doplnená o formát varovania. Regulárny výraz pre tieto verzie má následne tvar:

```
(.+):(\d+):(\d+): warning: (.+) \[(.+)\]
```

Pri identifikácií verzie `gcc-4.8`, ktorá vo výstupe môže obsahovať niekoľko riadkov, sa prostredníctvom regulárnych výrazov určuje, či na ďalšom riadku pokračuje varovanie. Program teda overuje, že na následujúcom riadku sa nenachádza nová informácia, začínajúca menom prekladaného súboru regulárnym výrazom:

```
(.+).c(.*):(.+)
```

Po identifikácií prekladača sú varovania na základe príslušného regulárneho výrazu rozdelené na jednotlivé časti pomocou funkcie `group()`. Jednotlivé časti sú následne porovnávané s obsahom užívateľského súboru a v prípade zhody nie sú tieto varovania zobrazované na štandardný výstup.

Po filtrovaní varovaní program následne kontroluje, či sa za týmito neželanými varovaniami nenachádza aj poznámka, ktorá toto varovanie dopĺňa a ktorej zobrazovanie po filtrácií varovania nie je žiadúce. Táto poznámka sa taktiež identifikuje pomocou regulárneho výrazu v tvare:

```
(.+): note: (.+)
```

Program identifikuje potencionálne nadbytočné informácie vo formáte `In function [meno funkcie]` prostredníctvom regulárneho výrazu, ktorý ma tvar:

```
(.+).c(.*): In function (.+)
```

Po jej identifikácií program túto informáciu nezobrazuje, ale prechádza výstupom ďalej. V prípade, že v danej sekcií bude zobrazená nejaká informácia, zobrazí sa najprv riadok s potencionálnou nadbytočnou informáciou a následne ostatné informácie súvisiace s touto sekciou.

6.6 Filtrovanie varovaní prekladača Clang

Filtrovanie varovaní pre prekladač Clang je implementované vo funkcií `parseClangWarning()`, ktorá sa nachádza v súbore `clangFilter.py` a je importovaná do hlavného programu.

Rovnako ako vo funkcií, ktorá filtrovala varovania prekladača GCC, aj funkcia pre filtrovanie varovaní prekladača Clang má ako vstupné parametre výstup prekladača a súbor s neželanými varovaniami. Program rovnako prechádza po riadkoch výstup prekladača a hľadá v ňom varovania pomocou regulárneho výrazu, ktorý je v tvare:

```
(.+):(\d+):(\d+): warning: (.+)
```

Po identifikácií varovania je následne toto varovanie rozdelené na jednotlivé časti pomocou funkcie `group`. V popise varovania sa potom kontroluje, či dané varovanie obsahuje v tomto popise formát. Táto kontrola prebieha opäť pomocou regulárneho výrazu, ktorý má tvar:

```
(.+) \[(.+)\]
```

Jednotlivé časti varovaní sa porovnávajú s obsahom užívateľského súboru a v prípade zhody sú varovania filtrované. Keďže varovania prekladača Clang obsahujú niekoľko riadkov, koniec varovania sa určuje prostredníctvom regulárnych výrazov. Program v prípade identifikácie neželaného varovania najprv kontroluje, či sa za týmto varovaniam nenachádzajú dodatočné poznámky. Tieto sú identifikované regulárnym výrazom, ktorý má tvar:

```
(.+): note: (.+)
```

V prípade, že táto poznámka nebola nájdená, je použitý regulárny výraz, ktorý overuje, či daný riadok obsahuje meno prekladaného súboru, čím identifikuje koniec varovania. Tento regulárny výraz má tvar:

```
(.+).c(.*):(.+)
```

Program taktiež pomocou regulárneho výrazu kontroluje, či práve skúmaný riadok, nie je riadkom posledným. Tento regulárny výraz má tvar:

```
(\d+) warning(.?) (.+)
```

Po nájdení tohto riadku je tento riadok upravený podľa aktuálneho počtu varovaní, ktoré budú zobrazené. Tento údaj je získaný prostredníctvom počítadla, ktoré je v programe implementované a pre každé zobrazené varovanie je jeho hodnota zvýšená.

6.7 Filtrovanie varovaní prekladača Javac

Filtrovanie varovaní pre prekladač Javac je implementovaný vo funkcií `parseJavacWarning`, ktorá sa nachádza v súbore `javacFilter.py`.

Rovnako aj táto funkcia obsahuje dva vstupné parametre, ktorými sú výstup prekladača a súbor s neželanými varovaniami. Program taktiež prebieha výstupom prekladača po jednotlivých riadkoch a hľadá varovania pomocou regulárneho výrazu, ktorý je v tvare:

```
(.+):(\d+): warning: \[(.+)\] (.+)
```

Toto varovanie je rovnako ako v predchádzajúcich prípadoch rozdelené pomocou funkcie `group` a jednotlivé časti sú porovnávané s obsahom užívateľského súboru. V prípade zhody varovania s varovaním nachádzajúcim sa v užívateľskom súbore, je toto varovanie z výstupu prekladača odstránené. Varovania prekladača Javac taktiež ležia na niekoľkých riadkoch, preto je koniec varovania opäť identifikovaný pomocou regulárneho výrazu, ktorý kontroluje, či daný riadok obsahuje meno prekladaného súboru a je v tvare:

```
(.+).java:(.+)
```

Program taktiež pomocou regulárneho výrazu overuje, či aktuálny riadok nie je riadkom posledným. Tento regulárny výraz je v tvare:

```
(\d+) warning(.?)(.+)
```

Po identifikácií posledného riadku je rovnako ako v prekladači Clang, tento riadok upravený podľa aktuálneho počtu zobrazených varovaní.

6.8 Filtrovanie varovaní prekladača GHC

Filtrovanie varovaní pre GHC je implementované vo funkcií „`parseGhcWarning`“, ktorá sa nachádza v súbore „ `ghcFilter`“.

Rovnako aj táto funkcia má ako parametre výstup prekladača a súbor s neželanými varovaniami. Program prechádza výstup prekladača, v ktorom hľadá riadky s menom prekladaného súboru a pozíciou v rámci zdrojového súboru. Tieto informácie sú identifikované pomocou regulárneho výrazu, ktorý je v tvare:

```
(.+).hs:(\d+):(\d+)
```

Po identifikácií tohto riadku program skontroluje, či ďalší riadok identifikuje varovanie pomocou regulárneho výrazu v tvare:

```
(.+) Warning: (.+)
```

V prípade, že varovanie bolo identifikované je z prvého riadku použitá pozícia varovania a z druhého riadku popis varovania. Tieto údaje sú následne porovnané s obsahom užívateľského súboru, kde v prípade zhody príde k filtrovaniu varovania z výstupu prekladača.

Okrem tohto spôsobu, kde program hľadá varovania, ktoré obsahujú kľúčové slovo `Warning` na inom riadku ako je popis, hľadá program aj varovania, ktoré toto kľúčové slovo obsahujú na rovnakom riadku, ako je popis. Táto identifikácia prebieha pomocou regulárneho výrazu v tvare:

```
(.+).hs:(\d+):(\d+): Warning: (.+)
```

Program zároveň u jednotlivých riadkov kontroluje, či sa nejedná o posledný riadok výstupu, aby v prípade filtrovania varovania, nebol odstránený aj tento riadok. Identifikujúci regulárny výraz je v tvare:

```
Linking (.+)
```

6.9 Implementácia rozšíriteľnosti

V prípade, že by užívateľ chcel v budúcnosti rozšíriť program o ďalšie prekladače, je možné do programu tieto prekladače doimplementovať. Je vhodné samotnú implementáciu filtrovania varovaní implementovať do externého súboru, ktorý bude importovaný do hlavného súboru `parser.py`.

V prípade, že užívateľ nebude mať záujem pri pridávaní nového prekladača implementovať dodatočnú identifikáciu prekladača, ale bude prekladač identifikovať v rámci užívateľského súboru, je nutné v súbore `parser.py` vo funkcií `recognizeFunctionForParsing()` pridať vetvu k príkazu `if`, v ktorej bude identifikovaný prekladač podľa názvu uvedeného v užívateľskom súbore a následne bude volaná funkcia z externého súboru, v ktorej je implementované filtrovanie varovaní.

V prípade, že spolu s prekladačom bude užívateľ chcieť implementovať aj dodatočnú identifikáciu prekladača na základe jeho výstupu, je nutné zmeniť implementáciu súboru `recognizer.py`. V tomto súbore sa nachádza funkcia `recognize_compiler()`, do ktorej je nutné pridať regulárny výraz, na základe ktorého bude možné jednoznačne určiť nový prekladač. Po implementovaní identifikácie stačí doplniť vetvu príkazu `if` v súbore `parser.py`, ako tomu bolo v predchádzajúcom prípade.

Kapitola 7

Testovanie programu

Táto kapitola sa zaoberá testovaním programu. Jej cieľom je ukázať na príkladoch, že program pracuje správne. V uvedených prípadoch je testované, že program filtruje z výstupov jednotlivých prekladačov varovania, ktoré si užívateľ neželá zobrazovať.

Program bol implementovaný v programovacom jazyku `Python 3.2` a následne testovaný na operačnom systéme `Kubuntu` vo vezií `12.10` a taktiež na školskom serveri `Merlin`. V tejto kapitole sú názorne ukázané štyri demonštračné testy. Na priloženom CD sa nachádza 15 testovacích súborov, na ktorých je ukázaná funkčnosť programu.

7.1 Prekladač GCC

Pre testovanie funkcie, ktorá obsahuje implementáciu filtra varovaní pre prekladač GCC bolo vytvorených niekoľko demonštračných testov. V jednom z testov majme následovný výstup prekladača GCC verzie `gcc-4.7`:

```
test.c: In function ‘main’:
test.c:6:2: warning: field precision specifier ‘.*’ expects a matching
         ‘int’ argument [-Wformat]
test.c:6:2: warning: format ‘%d’ expects a matching ‘int’ argument
         [-Wformat]
test.c:8:9: error: ‘neco’ undeclared (first use in this function)
test.c:8:9: note: each undeclared identifier is reported only once for each
         function it appears in
test.c:5:12: warning: unused variable ‘c’ [-Wunused-variable]
test.c:5:9: warning: unused variable ‘b’ [-Wunused-variable]
test.c:5:6: warning: unused variable ‘a’ [-Wunused-variable]
test.c:4:6: warning: unused variable ‘o’ [-Wunused-variable]
```

Ako je vidieť na príklade, vo výstupe prekladača sa nachádza niekoľko varovaní, kde prvé dve varovania vznikli nesprávnym použitím funkcie `printf()` a ďalšie varovania deklarovaním premenných, ktoré nie sú v zdrojovom súbore ďalej použité. Predpokladajme, že užívateľ si neželá zobrazovať varovania, ktoré upozorňujú na nepoužité premenné. V tomto prípade má užívateľ niekoľko spôsobov ako tieto varovania filtrovať. Jedným z efektívnejších spôsobov je filtrovanie týchto varovaní prostredníctvom ich formátu, ktorý je rovnaký. Pri tejto alternatíve stačí zapísať do súboru s neželanými varovaniami reťazec `-Wunused-variable`. Následne po spustení programu príkazom:

```
gcc -Wall -Wextra test1.c 2>&1 | ./filter.py warnings.txt
```

Bude výstup prekladača následovný:

```
test.c: In function ‘main’:
test.c:6:2: warning: field precision specifier ‘.*’ expects a matching
         ‘int’ argument [-Wformat]
test.c:6:2: warning: format ‘%d’ expects a matching ‘int’ argument
 [-Wformat]
test.c:8:9: error: ‘neco’ undeclared (first use in this function)
test.c:8:9: note: each undeclared identifier is reported only once for each
         function it appears in
```

Vo verzií gcc-4.5 a starších sa však formát varovaní nenachádza, preto je vhodné tieto varovania filtrovať tak aby boli u varovania s popisom `unused variable` odstránené všetky bez ohľadu na to, akú obsahujú premennú. Táto alternatíva je možná v prípade, keď užívateľ zadá do súboru s neželanými varovaniami reťazec `unused variable '*'`. Výstup prekladača je potom rovnaký ako v predchádzajúcom prípade.

7.2 Prekladač Clang

Keďže prekladač Clang prekladá rovnako ako prekladač GCC súbory rodiny jazyka C, je možné použiť na oba prekladače rovnaké testy. U jedného z testov funkcie, v ktorej je implementované filtrovanie varovaní prekladača Clang, majme výstup vo verzií `3.2`:

```
test2.c:5:32: warning: unused parameter 'b' [-Wunused-parameter]
void pocitaj(int i, int o, int b){
                               ^
test2.c:15:16: warning: variable 'o' is uninitialized when used here
          [-Wuninitialized]
    pocitaj(i, o, u);
               ^
test2.c:12:12: note: initialize the variable 'o' to silence this warning
    int i,o,u;
           ^
            = 0
test2.c:14:27: warning: variable 'i' is uninitialized when used here
          [-Wuninitialized]
    printf("Skuska vypis",i);
                          ^
test2.c:12:10: note: initialize the variable 'i' to silence this warning
    int i,o,u;
         ^
          = 0
test2.c:15:19: warning: variable 'u' is uninitialized when used here
          [-Wuninitialized]
    pocitaj(i, o, u);
                  ^
test2.c:12:14: note: initialize the variable 'u' to silence this warning
```

```
    int i,o,u;
            ^
              = 0
4 warnings generated.
```

Tento výstup obsahuje niekoľko varovaní, kde prvé varovanie opäť upozorňuje na situáciu, kedy definícia funkcie obsahuje parameter, ktorý nie je ďalej zdrojovom súbore použitý. Zvyšné varovania upozorňujú na situáciu, kedy premenné boli definované a použité ako parametre funkcie, ale neboli im priradené žiadne hodnoty.

Predpokladajme, že užívateľ si neželá zobraziť varovanie, ktoré upozorňuje na nevyužitý parameter. V tomto prípade, keďže vo výstupe sa nachádza len jedno varovanie tohto typu má užívateľ na výber viacero možností filtrovania. jedným so spôsobom je zápis pozície varovania do súboru s neželanými varovaniami, ktorý by v tomto prípade mohol mať tvar buď `5:32` alebo iba `5`. V prípade ak by chcel užívateľ filtrovať toto varovanie na základe jeho popisu môže do súboru zadať buď reťazec `unused parameter 'b'`, prípadne `unused parameter '*'`, pre filtrovanie prípadne ďalších varovaní tohto typu. Posledným spôsobom filtrovania tohto varovania je zadanie formátu varovania do súboru s neželanými varovaniami v tvare `-Wunused-parameter`. Po zadaní jednej z možností do súboru s neželanými varovaniami a spustením programu príkazom

```
clang -Wall -Wextra test1.c 2>&1 | ./filter.py warnings.txt
```

Bude výstup prekladača následovný:

```
test2.c:15:16: warning: variable 'o' is uninitialized when used here
          [-Wuninitialized]
    pocitaj(i, o, u);
               ^
test2.c:12:12: note: initialize the variable 'o' to silence this warning
    int i,o,u;
           ^
             = 0
test2.c:14:27: warning: variable 'i' is uninitialized when used here
          [-Wuninitialized]
    printf("Skuska vypis",i);
                          ^
test2.c:12:10: note: initialize the variable 'i' to silence this warning
    int i,o,u;
         ^
           = 0
test2.c:15:19: warning: variable 'u' is uninitialized when used here
          [-Wuninitialized]
    pocitaj(i, o, u);
                  ^
test2.c:12:14: note: initialize the variable 'u' to silence this warning
    int i,o,u;
             ^
               = 0
3 warnings generated.
```

Ako je vidieť z výstupu prekladača, nechcené varovanie bolo filtrované a zároveň aj u posledného riadku výstupu, ktorý obsahuje počítadlo varovaní, prípadne chybových hlásení, bola hodnota zmenená na aktuálny počet zobrazených varovaní.

V prípade, že užívateľ by si neželal zobrazovať varovania, ktoré upozorňuje na situáciu, kedy definovaná premenná nebola inicializovaná, je vhodné filtrovať tieto varovania buď prostredníctvom ich formátu, prostredníctvom zadania reťazca `-Wuninitialized` do súboru s neželanými varovaniami, prípadne zadaním do súboru reťazec `variable '*' is uninitialized when used here` pre odstránenie varovaní prostredníctvom ich popisu. Následne po spustení programu bude výstup prázdny.

7.3 Prekladač Javac

Rovnako aj pre testovanie filtrovania prekladača Javac bolo vytvorenie niekoľko demonštračných testov. U jedného z vytvorených testov majme výstup prekladača:

```
User.java:3: warning: [deprecation] deprecatedMethod() in Main has been
         deprecated
       Main.deprecatedMethod();
           ^
1 warning
```

Vo výstupe prekladača sa nachádza varovanie, ktoré upozorňuje na situáciu, kedy je v zdrojovom súbore použitá zastaralá metóda, ktorej použitie môže spôsobiť neočakávané situácie pri behu programu.

Predpokladajme, že užívateľ si dané varovanie neželá zobraziť. Opäť je tu niekoľko možností zápisu varovania do súboru s neželaným varovaním. Prvým spôsobom je možnosť zadanie pozície varovania, ktorá by bola v tomto prípade vo formáte `3`. Ďalšou možnosťou je zadanie popisu varovania do súboru, ktorý by mal v tomto prípade tvar `deprecatedMethod() in Main has been deprecated`. Poslednou možnosťou je filtrovanie varovania prostredníctvom jeho formátu, kde by mal zápis do súboru tvar `deprecation`. Po zadaní jedného z týchto zápisov do súboru s neželanými varovaniami a spustením programu pomocou príkazu:

```
javac -Xlint User.java 2>&1 | ./parser.py warnings.txt
```

bude výstup prekladača prázdny, pretože okrem varovania bude odstránená aj informácia o aktuálnom počte varovaní, ktorá by bola v tomto prípade informáciou nadbytočnou.

7.4 Prekladač GHC

Pre testovanie funkcie, ktorá obsahuje filtrovanie varovaní prekladača GHC bolo taktiež vytvorených niekoľko testov. Majme výstup prekladača pri preklade jedného z testov:

```
[1 of 1] Compiling Main             ( TestHaskel3.hs, TestHaskel3.o )

TestHaskel3.hs:6:1:
    Warning: Top-level binding with no type signature: main :: IO ()

TestHaskel3.hs:9:8:
```

```
    Warning: Pattern match(es) are non-exhaustive
             In a case alternative:
                 Patterns not matched:
                     []
                     (GHC.Types.C# #x) : _ with #x ‘notElem‘ [’y’, ’n’]
                     [GHC.Types.C# ’y’]
                     (GHC.Types.C# ’y’) : ((GHC.Types.C# #x) : _)
                     with
                     #x ‘notElem‘ [’e’]
                     ...
Linking TestHaskel3 ...
```

V tomto prípade prvé varovanie upozorňuje na situáciu, kedy v zdrojovom súbore nie sú definované dátové typy u použitých výrazov. Ďalšie varovanie upozorňuje zasa na situáciu, kedy sa v zdrojovom kóde nachádza príkaz `case`, ktorý neobsahuje vetvu, ktorá zahŕňa všetky neuvedené stavy.

Povedzme, že užívateľ si želá filtrovať prvé varovanie vo výstupe prekladača. V tomto prípade má opäť niekoľko možností. Jednou z možností je zápis varovania do súboru s neželanými varovaniami prostredníctvom pozície varovania. V tomto prípade by mal zápis tvar buď `6:1`, prípadne iba `6`. Ďalšou možnosťou zápisu je prostredníctvom popisu varovaniam ktorý by mal v tomto prípade tvar `Top-level binding with no type signature: main :: IO ()`. Po zadaní jednej z týchto možností do súboru s neželanými varovaniami a následným spustením programu prostredníctvom príkazu:

```
ghc -Wall TestHaskel3 2>&1 | ./parser.py warnings.txt
```

Bude výstup programu nasledujúci:

```
[1 of 1] Compiling Main             ( TestHaskel3.hs, TestHaskel3.o )

TestHaskel3.hs:9:8:
    Warning: Pattern match(es) are non-exhaustive
             In a case alternative:
                 Patterns not matched:
                     []
                     (GHC.Types.C# #x) : _ with #x ‘notElem‘ [’y’, ’n’]
                     [GHC.Types.C# ’y’]
                     (GHC.Types.C# ’y’) : ((GHC.Types.C# #x) : _)
                     with
                     #x ‘notElem‘ [’e’]
                     ...
Linking TestHaskel3 ...
```

V prípade, že užívateľ bude filtrovať obe varovania, ostane vo výstupe programu len prvý a posledný riadok programu, ktorý informuje užívateľa o prekladaných súboroch, preto je vhodné tieto riadky v programe zanechať.

Kapitola 8

Záver

Úlohou tejto práce bolo analyzovať rôzne druhy a formáty varovaní prekladačov GCC, Clang, Javac a GHC a uviesť čitateľa do tejto problematiky. Na základe analýzy jednotlivých prekladačov a ich varovaní boli následne vyhodnotené situácie, v ktorých tieto varovania nepredstavujú potencionálne nebezpečenstvo. Po tejto analýze bolo možné vytvoriť program, ktorý filtruje užívateľmi neželané varovania z výstupu jednotlivých prekladačov.

V ďalších kapitolách bola postupne rozobraná tvorba programu ako návrh tohto program a jeho implementácia v jazyku Python. V práci sú taktiež spomínané problémy, ktoré bolo nutné pri implementácií aplikácie riešiť a následne boli rozobrané implementačné postupy použité pri tvorbe programu.

Dôležitou súčasťou bolo aj testovanie programu, s ktorého výsledkami je čitateľ oboznámený v záverečnej časti práce. Na základe testov, ktoré boli ukázané v práci, prípadne uložené na priloženom CD bolo ukázané, že program je schopný filtrovať varovania pre prekladače GCC a jeho verzie `gcc-4.4` – `gcc-4.8`, Clang a jeho verzie `2.9` – `3.2`, Javac vo verzií `1.7` a GHC vo verzií `7.6`, ako aj ďalšie časti výstupu, ktoré súvisia s varovaniami a po ich odstránení sa môžu zdať nadbytočnými.

V práci bola taktiež uvedená možnosť rozšíriteľnosti programu o ďalšie prekladače a popis ako je možné pridať do programu implementáciu filtrovania varovaní pre tieto prekladače.

Literatura

[1] Options to Request or Suppress Warnings.
URL <http://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html>

[2] Warnings and sanity-checking.
URL <http:
//www.haskell.org/ghc/docs/7.4.1/html/users_guide/options-sanity.html>

[3] Structure of GCC. 2008.
URL <http://gcc.gnu.org/wiki/StructureOfGCC>

[4] Clang - Features and Goals. 2013.
URL <http://clang.llvm.org/>

[5] javac - Java programming language compiler. 2013.
URL <http:
//docs.oracle.com/javase/7/docs/technotes/tools/solaris/javac.html>

[6] Beneš, M.: *Prekladače*. Vysoké učení technické v Brne.
URL <http://www.fit.vutbr.cz/~meduna/fjp/skripta.pdf>

[7] D. Erni, A. K.: *The Hacker's Guide to Javac*. University of Bern, 2008.

[8] D. Grunde, H. E. B.: *Modern compiler design*. New York: J. Wiley, 2001, ISBN 0-471-97697-0, 736 s.

[9] Divicky, R.: Clang/LLVM: potenciolni konkurent prekladace gcc. 2009.
URL <http:
//www.root.cz/clanky/clang-llvm-potencialni-konkurent-prekladace-gcc/>

[10] Habiballa, H.: *Prekladače*. Ostravská univerzita v Ostrave, 2005.

[11] Kollár, J.: *Návrh a interpretácia programovacích jazykov*. Elta, 2002.

[12] Muller, K.: *Programovací jazyky*. České vysoké učení v Praze, 2001.

[13] S. Marlow, S. P. J.: *The Glasgow Haskell Compiler*. 2012.

[14] Shaw, J.: *LEARN HASKELL*. 2007.
URL <http://learnhaskell.blogspot.com/>

[15] Stallman, R. M.: *An Introduction to GCC*. Network Theory Limited, 2004, ISBN 0-9541617-9-3.