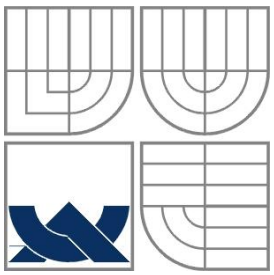
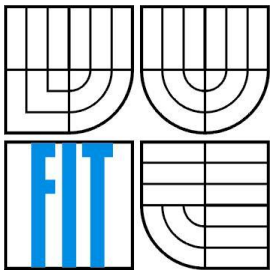


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

PŘEVOD VÝRAZŮ V C DO DIMACS FORMÁTU

TRANSLATION OF C EXPRESSIONS TO DIMACS FORMAT

DIPLOMOVÁ PRÁCE
MASTER‘S THESIS

AUTOR PRÁCE
AUTHOR

BC. PAVEL GRIM

VEDOUCÍ PRÁCE
SUPERVISOR

ING. ALEŠ SMRČKA, PH.D.

BRNO 2015

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2014/2015

Zadání diplomové práce

Řešitel: **Grim Pavel, Bc.**
Obor: Inteligentní systémy
Téma: **Převod výrazů v C do DIMACS formátu**
Translation of C Expressions to DIMACS Format
Kategorie: Návrh číslicových systémů

Pokyny:

1. Nastudujte DIMACS formát (formát pro popis formulí v konjunktní normální formě, CNF). Podrobně nastudujte základními operace jazyka C nad bitovými vektory.
2. Navrhněte převod mezi výrazy v jazyku C nad vícebitovými proměnnými do formátu DIMACS.
3. Navržený převod implementujte v jazyku C/C++.
4. Vytvořte testovací sadu převodu. Vytvořte demonstraci použití vašeho nástroje.
5. Diskutujte další možná rozšíření.

Literatura:

1. Satisfiability Suggested Format (DIMACS), 1993, URL: http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf
2. D.S. Johnson, D.S.; Trick, M.A. Cliques, Coloring, and Satisfiability, volume 26 of DIMACS Series on Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 1996.

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2014
Datum odevzdání: 27. května 2015

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Tato práce se věnuje návrhu převodu výrazů zapsaných v programovacím jazyce C do formátu DIMACS a realizaci programu v jazyce C++ provádějící tento převod. V textu práce se nachází popis programovacího jazyka C a jeho operátorů. Dále obsahuje popis konjunktivní normální formy a popis formátu DIMACS. Následuje návrh vytvoření programu pro uskutečnění převodu z výrazu v programovacím jazyce C do formátu DIMACS a popis realizace programu provádějící tento převod.

Abstract

This work focuses on proposition of transfer of the expressions entered in the C programming language into DIMACS format and creation of program in programming language C++ making this transfer. This work contains a description of the C programming language and its operators. It also contains a description of the conjunctive normal form and a description of the DIMACS format. Following is a proposal for a program for the transfer of expression in the C programming language to the DIMACS format and description of realization of program performing this transfer.

Klíčová slova

C, jazyk C, operátory, bitové operátory, celočíselné operátory, výraz, výrazy jazyka C, převod, konverze, konjunktivní normální forma, CNF, DIMACS, DIMACS formát.

Keywords

C, C language, operators, bitwise operators, integer operators, expression, C expression, transfer, translation, conversion, conjunctive normal form, CNF, DIMACS, DIMACS format.

Citace

Grim Pavel: Převod výrazů v C do DIMACS formátu, diplomová práce, Brno, FIT VUT v Brně, 2015

Převod výrazů v C do DIMACS formátu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením p. Ing. Aleše Smrčky, Ph.D.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Pavel Grim
27. května 2015

Poděkování

Tímto bych chtěl poděkovat p. Ing. Aleši Smrčkovi, Ph.D. za poskytnuté rady při řešení problémů a za cenné rady a pomoc při psaní práce.

Obsah

1 Úvod

Existuje mnoho způsobů, jak popsat chování určitého systému a každá tato možnost je různě náročná pro manuální zápis, ale také pro různorodé zpracování v automatizovaném výpočetním prostředí. Chování systému může být definováno slovně, vývojovým diagramem, zápisem v programovacím jazyku nebo jinak. Různé účely práce s popisem systému mohou vyžadovat, aby existující zápis v jedné formě byl konvertován do jiného zápisu. Pro tyto účely mohou existovat programy, které provedou převod z jednoho formátu popisu systému do jiného.

Uživatel, který si je vědom existence těchto programů pro převod, je může využít. V některých situacích může být potřeba popsat systém ve formátu, který je příliš těžký nebo náročný pro přímý zápis. Pokud by existoval program pro převod do tohoto formátu z nějakého lehčího formátu pro zápis, může uživatel využít dostupných prostředků a ulehčit si práci.

Tato práce se věnuje návrhu převodu z výrazu zapsaného v programovacím jazyce C do formátu DIMACS a vytvoření programu v C++, který by tento navržený převod prováděl.

Kapitola 2 obsahuje formy popisy systémů, které se týkají této práce. První popisovanou formou popisu systémů je programovací jazyk C společně s jeho operátory. Tato kapitola obsahuje detailní popis operátorů, které budou podporovány v rámci této práce. Dále tato kapitola obsahuje popis konjunktivní normální formy a formátu DIMACS. Další kapitola, kapitola 3, popisuje návrh uskutečnění převodu z výrazu zapsaného v programovacím jazyku C do formátu DIMACS a návrh programu, který by tento převod realizoval. V kapitole 4 je popis provedené implementace, struktura napsané aplikace a její některé implementační detaily. Nachází se tam také popis způsobu testování aplikace, ukázka užití aplikace a vyhodnocení. Poslední 5. kapitolou je závěr shrnující obsah práce a dosažených výsledků z předchozích kapitol.

2 Formy popisu systémů

Tato kapitola se věnuje formám popisu systémů, které budou dále použity v této práci a případnému vysvětlení souvisejících témat. Obsahuje popis programovacího jazyka C, jeho výrazů a operátorů, a také popis formátu DIMACS.

V podkapitole 2.1 je popsán programovací jazyk C. Operátory programovacího jazyka C, které budou použity v této práci, jsou popsány v podkapitole 2.2. Podkapitola 2.3 obsahuje popis logických formulí a popis konjunktivní normální formy. Popis formátu DIMACS je v podkapitole 2.4.

2.1 Programovací jazyk C

Programovací jazyk C byl vyvinut v letech 1969 – 1973 a je založen na programovacím jazyku B, který byl založen na programovacím jazyku BCPL [1][2][3]. V roce 1983 byla stanovena komise s cílem standardizace programovacího jazyka C a tuto komisi sestavil Americký národní standardizační institut, ANSI. Na konci roku 1989 byl publikován standard, který je znám jako ANSI-C, nebo také jako C89. V dalším roce byl standard přijat mezinárodní organizací pro normalizaci, ISO, jako ISO/IEC 9899-1990, znám také jako standard C90 [1][2][3]. Další standardy následovaly (C95, C99, C11) a tento programovací jazyk je stále ve vývoji. Tento projekt se bude řídit podle standardu C99.

Jedná se o všestranně použitelný imperativní programovací jazyk, který není vázán na žádné konkrétní technické vybavení. Poskytuje možnost pro strukturované programování, datové struktury, rekurzi a další [3][4]. Díky nezávislosti na konkrétní technické vybavení existuje možnost přeložit programy psané v tomto jazyce na různých strojích podporujících tento programovací jazyk. Programovací jazyk C není vysoko úrovňový jazyk, není ani nijak specializovaný, ale nemá také žádná omezení. Díky těmto vlastnostem je možné ho použít pro různé úkoly. Konstrukce napsané v tomto programovacím jazyku lze efektivně nahradit za typické strojové instrukce, čímž se stal náhradou pro aplikace, které byly původně napsány ve strojovém kódu [3][4]. Dennis Ritchie vytvořil tento programovací jazyk pro operační systém UNIX, ve kterém byl také implementován, ale nebyl vytvořen jen pro tento operační systém. Operační systém UNIX i všechny jeho části byly napsány právě v programovacím jazyku C.

Jednou z konstrukcí jazyka C jsou výrazy. Pomocí výrazů se vypočítávají nové hodnoty, volají funkce, vytvářejí objekty anebo vytvářejí vedlejší účinky [3][4][5][6]. Výraz je sekvence operátorů a operandů, kde operand může být konstanta, proměnná nebo výsledek funkce. Pořadí vyhodnocení jednotlivých operátorů ve výrazu určuje jejich priorita a asociativita [3][4][6][7][8]. Záporné celočíselné proměnné a hodnoty mohou být kódovány v přímém kódu, jedničkovém doplňku nebo v dvojkovém doplňku [5]. V rámci této práce bude uvažován pouze dvojkový doplněk.

2.2 Vybrané operátory programovacího jazyka C

Programovací jazyk C má mnoho operátorů, ale ne všechny budou podporovány v rámci této práce a v této podkapitole budou rozebrány pouze ty operátory, které podporovány budou. Protože práce bude podporovat pouze celočíselné hodnoty a proměnné, operátory pro přístup k datům zde nebudou dále zmíněny. Mimo dalších jsou vynechané operátory násobení, dělení a zůstatek po dělení.

2.2.1 Priorita a asociativita operátorů

Operátory jsou rozděleny podle jejich priorit a více operátorů může mít stejnou prioritu. Více prioritní operátory jsou ve výrazu vyhodnoceny před operátory s nižší prioritou. Jednotlivé skupiny operátorů podle jejich priority v sestupném pořadí [4][5][7][8]:

- Explicitní přetypování, logická negace, bitová negace, unární plus a unární mínus.
- Sčítání a odčítaní.
- Bitový posun vlevo a bitový posun vpravo.
- Menší, menší nebo roven, větší a větší nebo roven.
- Porovnání rovnosti a porovnání nerovnosti.
- Bitový součin.
- Bitový exkluzivní součet.
- Bitový součet.
- Logický součin.
- Logický součet.
- Ternární operátor.
- Přiřazení.

Priorita operátorů je důležitá, aby bylo možné vyhodnotit výraz vždy stejným způsobem. Právě kvůli prioritám operátorů je pořadí vyhodnocení operátorů v rovnici (2.1) totožné s pořadím vyhodnocení operátorů v rovnici (2.2).

$$a = b + 12 \ll c - 2 \tag{2.1}$$

$$a = ((b + 12) \ll (c - 2)) \tag{2.2}$$

Další důležitou vlastností operátorů je jejich asociativita. Většina operátorů je asociativní zleva doprava [4][5][7][8]. Operátory explicitní přetypování, logická negace, bitová negace, unární plus, unární mínus, ternární operátor a přiřazení jsou asociativní zprava doleva. Asociativita operátorů určuje pořadí vyhodnocení operátorů, pokud mají operátory stejnou prioritu [4][5][8].

Vyhodnocení operátorů v rovnici (2.3) je určeno asociativitou operátoru přiřazení, a protože je operátor přiřazení asociativní zprava doleva, rovnice (2.3) má totožné pořadí vyhodnocení operátorů jako rovnice (2.4).

$$a = b = c = 5 \tag{2.3}$$

$$a = (b = (c = 5)) \tag{2.4}$$

2.2.2 Typové konverze

Operátory mohou vyžadovat implicitní konverzi operandů na společný datový typ [4][5]. Pravidla pro implicitní konverzi většinou převádějí datový typ na širší typ, aby nedošlo ke ztrátě informací. Výrazy, které jsou implicitně převáděny na typ, kde může dojít ke ztrátě informací, mohou vyvolat varování od překladače, ale nejsou zakázané [4][5]. Výsledek implicitní konverze je totožný s explicitní konverzí. Tato práce uvažuje pouze celočíselné datové typy, a proto budou uvedeny pouze konverze, které se k nim vztahují a budou vynechány konverze pro ostatní datové typy.

Nad datovými typy může být provedena operace nazývaná zvýšení úrovně celého čísla, *integral promotions* [4][5]. Tato operace převádí datové typy pro znak, krátké celé číslo, celočíselné bitové pole nebo výčtový typ na datový typ reprezentující znaménkové nebo bezznaménkové celé číslo. Pokud datový typ pro celá znaménková čísla může reprezentovat všechny hodnoty původního typu, tak je hodnota převedena na celé znaménkové číslo. Jinak je hodnota převedena na celé bezznaménkové číslo [4], podkapitola 6.3.1.1 normy C99 [5].

Při provádění konverze datových typů operandů operátoru může být využit proces nazývaný obvyklé aritmetické konverze, *usual arithmetic conversions* [4], podkapitola 6.3.1.8 normy C99 [5]. Tato konverze je definována jako sada šesti pravidel. Pravidla vypadají následovně:

- Nad datovými typy je provedena operace známá jako zvýšení úrovně celého čísla, *integral promotions*.
- Pokud jsou oba typy operandů stejného typu, žádný další převod se neprovádí.

- Pokud jsou oba typy operandů znaménkové nebo jsou oba bezznaménkové, tak se typ s menší přesností převede na typ s vyšší přesností.
- Pokud má bezznaménkový typ operandu vyšší přesnost než operand se znaménkovým typem, je operand se znaménkovým typem převeden na stejný typ, jako má operand s bezznaménkovým typem.
- Pokud má znaménkový typ operandu vyšší přesnost než operand s bezznaménkovým typem, je operand s bezznaménkovým typem převeden na stejný typ, jako má operand se znaménkovým typem.
- V ostatních případech jsou oba operandy převedeny na bezznaménkový typ, který koresponduje se znaménkovým typem.

Převod na logickou hodnotu, má výsledek logickou hodnotu nepravda, pokud je hodnota rovna nule. V ostatních případech je výsledek logická pravda [5].

Pokud datový typ pro znak nemá explicitní označení, jestli se jedná o znaménkový nebo bezznaménkový datový typ, tak programovací jazyk C nedefinuje, jestli je tento typ znaménkový nebo ne. Interpretace a tedy i konverze tohoto typu se může lišit na různých strojích [4], podkapitola 6.2.5 normy C99 [5]. Je doporučeno explicitně specifikovat, jestli je tento typ se znaménkem nebo bez znaménka.

2.2.3 Charakteristika operátorů

V této podkapitole je popis jednotlivých podporovaných operátorů v rámci práce. Sémantika jednotlivých operátorů je čerpána z podkapitoly 6.5 normy C99 [5].

Unární operátory

Unární operátor plus, „+“, provádí operaci zvýšení úrovně celého čísla, *integral promotions* [4][5]. Tato operace je popsána v kapitole 2.2.2. Operátor unární mínus, „-“, provede zvýšení úrovně celého čísla a vrací jeho zápornou hodnotu [4][5].

Operátor bitové negace, „~“, provede zvýšení úrovně celého čísla a vrací hodnotu, která odpovídá původní hodnotě, kde každý bit byl negován [4][5]. Operátor logická negace, „!“, má výsledek hodnotu jedna, pokud původní hodnota byla rovna nule. V opačném případě je výsledkem hodnota nula. Výsledná hodnota má datový typ celého čísla se znaménkem [4][5].

Operátor přetypování, „`(typ)`“, provádí explicitní datovou konverzi hodnoty na zadaný datový typ [4][5].

Binární aritmetické operátory

Výsledkem binárního operátoru sčítání, „+“, je součet vstupních operandů. Binární operátor mínus, „-“, má výsledkem rozdíl operandů. Pro obě operace platí, že na svých operandech provedou obvyklé aritmetické konverze, *usual arithmetic conversions* [4][5].

Bitové operátory posunu

Operátor bitového posunu vlevo, „<<“, provede posun bitů v proměnné doleva. Operátor bitového posunu vpravo, „>>“, provede posun bitů v proměnné doprava. Volné bity jsou zaplněny nulami [4][5]. U obou operací je na obou operandech provedena operace zvýšení úrovně celého čísla, *integral promotions*. Výsledný typ závisí ale pouze na levém operandu [4][5]. Pokud je levý operand znaménkového typu a jeho hodnota je záporná, tak výsledek není definován [4][5]. V případě, že pravý operand má větší hodnotu, než je počet bitů zvýšeného levého operandu, nebo že pravý operand je záporné číslo, tak je výsledek nedefinován [4][5].

Bitové operátory

Binární operátory pro bitové operace jsou bitový součin, „&“, bitový součet, „|“, a bitový exkluzivní součet, „^“. Pro všechny tyto operátory dojde k provedení obvyklé aritmetické konverze, *usual arithmetic conversions*, na obou operandech [4][5]. Sémantika těchto operátorů je znázorněna v tabulce níže, viz Tabulka 2.1.

Tabulka 2.1 - Tabulka výsledků pro binární operace nad jedním bitem

A	*B*	*A & B*	*A | B*	*A ^ B*
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Logické operátory

Logickými operátory jsou logický součin, „&&“, a logický součet, „||“. Hodnota výsledku logického součinu je jedna, právě tehdy, když oba operandy nejsou rovny nule, a hodnota výsledku pro logický součet je jedna, pokud alespoň jeden z operandů není roven nule [4][5]. Výsledek má datový typ celého znaménkového čísla.

Tyto operátory jsou také výjimečné tím, že definují pořadí vyhodnocení operandů zleva doprava [4][5]. Pokud u logického součinu je první operand vyhodnocen jako nula, logická hodnota nepravda, tak výsledek operátoru je také nula a druhý operand není ani vyhodnocován. Obdobné je to u operátoru logického součtu, pokud je první operand vyhodnocen jako jedna, je výsledek operátoru jedna a druhý operand není vyhodnocován [4][5].

Relační operátory

Další skupinou jsou operátory pro různá porovnání hodnot. Jedná se o operátory rovnosti, „==“, nerovnosti, „!=“, menší, „<“, menší nebo rovno, „<=“, větší, „>“ a větší nebo rovno, „>=“. Pro všechny operátory jsou na vstupních operandech provedeny obvyklé aritmetické konverze, *usual arithmetic conversions* [4][5]. Pokud testovaná relace neplatí, tak je výsledkem nula, v opačném případě je výsledek jedna. Výsledek má datový typ celého znaménkového čísla [4][5].

Ternární operátor

Ternární operátor, „?:“, nejdříve vyhodnotí první operand, a pokud je výsledek roven nule, tak je výsledkem tohoto operátoru třetí operand, jinak je výsledkem druhý operand [4][5]. Výsledný datový typ je určen výsledkem po provedení obvyklých aritmetických konverzí, *usual arithmetic conversions*, na druhém a třetím operandu [4][5].

Operátor přiřazení

Operátor přiřazení, „=“, slouží pro uložení hodnoty do proměnné specifikované v levém operandu [4][5]. Hodnota pravého operandu je přetypována na datový typ levého operandu.

2.3 Logické formule v Konjunktivní normální formě

Logická formule je tvořena konstantami a proměnnými. Logické formule mohou být spojeny logickou spojkou a tvoří tak jinou logickou formuli. Pokud konstanty i proměnné nabývají nebo mohou nabývat pouze hodnot logické pravdy a nepravdy, jedná se o booleovskou logickou formuli [9][13]. V rámci této práce budou uvažovány dále pouze booleovské logické formule.

Konjunktivní normální forma, CNF, označuje logickou formuli, který splňuje určité podmínky. CNF se skládá z klauzulí spojených konjunkcí. Každá klauzule se skládá z proměnných spojených disjunkcí. Proměnné mohou být negovány [10][13]. Aby logická formule byla v CNF, musí splňovat právě tyto požadavky. Rovnice (2.5) na straně 9 je logickou formulí v CNF.

2.4 Formát DIMACS

Formát DIMACS slouží pro standardní reprezentaci logické formule pro řešení problému splnitelnosti [10]. DIMACS definuje dva různé formáty uložení. První formát slouží pro reprezentaci logické formule v konjunktivní normální formě, CNF, a je označován jako CNF formát, nebo také DIMACS CNF. Druhý formát slouží pro reprezentaci obecné logické formule a je označován jako SAT [10].

Soubor ve formátu DIMACS je rozdělen logicky na dvě části. První část slouží jako preambule a obsahuje komentáře a definici o jaký konkrétní formát se jedná. Druhá část soboru obsahuje logické formule [10][11][12].

V preambuli začínají řádky jedním znakem a mezerou. Znak určuje, co je na daném řádku uloženo. Prvním znakem mohou být pouze znaky `c` a `p` [10][11][12]. Úvodní znak `c` definuje, že je na řádku komentář pro člověka a má tedy být ignorován programem, který tento soubor zpracovává. Znak `p` značí, že se jedná o řádek s definicí problému. Řádek s definicí problému ve formátu CNF je definován jako „`p cnf VARIABLES CLAUSES`“, kde `VARIABLES` je počet proměnných ve výrazu a `CLAUSES` udává počet řádků s klauzulemi [10][11][12]. Definice problému pro formát SAT je obdobný, ale místo označení problému `cnf` obsahuje označení `sat` a neuvádí počet klauzulí [10].

Při použití formátu CNF jsou logické formule definovány jako čísla proměnných, které se vyskytují v odpovídající formuli, oddělených mezerou. Za konce jednotlivých formulí lze volitelně zapsat navíc nulu, jako ukončovač řádku. Negace je značena záporným znaménkem před číslem [10][11][12]. Formát SAT zapisuje formuli, jako výraz podobný výrazům z programovacího jazyka C. Používá závorky, `-` značí negaci, `+` logický součet a `*` logický součin [10]. Navíc lze využít „`*()`“ pro označení hodnoty logické pravdy a „`+()`“ pro hodnotu logické nepravdy.

$$(A \vee B \vee \neg \mathrm{C}) \wedge (B \vee D) \wedge (\neg \mathrm{A} \vee \neg \mathrm{D}) \tag{2.5}$$

Logický výraz v rovnici (2.5) je zapsán ve formátu CNF jako:

```
c Ukazka formatu cnf
p cnf 4 3
1 2 -3
2 4
-1 -4
```

A ve formátu SAT jako:

```
c Ukazka formatu sat
p sat 4
(*(+(1 2-3)
+(2 4)+(-1-4)))
```

3 Návrh realizace převodu

V této kapitole bude navržena realizace převodu z výrazu zapsaného v programovacím jazyce C do formátu DIMACS CNF. U návrhu tohoto převodu se bude vycházet z poznatků popsaných v kapitole 2. Bude také navržen program, který by tento převod prováděl.

Rozbor a návrh převodu jednotlivých operátorů programovacího jazyka C do konjunktivního normálního formátu se nachází v podkapitole 3.1. Podkapitola 3.2 obsahuje návrh programu, který realizuje převod výrazu v programovacím jazyce C, přes CNF, do formátu DIMACS CNF.

3.1 Převod operátorů programovacího jazyka C do CNF

Převod jednotlivých operátorů do CNF zachycuje vztahy jednotlivých operandů a výsledků na bitové úrovni. V článku od p. M. N. Veleva a p. R. E. Bryanta [14] je realizován převod logických operátorů bitový součet, bitový součin, ternární operátor a bitová negace na logické formule v CNF. Tato práce má ale podporovat mnohem více operátorů a proto je důležité zjistit, jak byly vytvořeny převody pro tyto operátory a pokusit se stejným způsobem navrhnout převod i pro ostatní operátory.

V podkapitole 3.1.1 jsou popsány převody operátorů pro bitovou negaci, bitový součet, bitový součin a bitový exkluzivní součet. Podkapitola 3.1.2 obsahuje popis převodu pro ternární operátor. Popis převodu operátorů pro sčítání a odčítání je v podkapitole 3.1.3. V podkapitole 3.1.4 se nachází popis převodu relačních operátorů. Jedná se o operátory porovnání rovnosti a nerovnosti, a o operátory menší, menší nebo rovno, větší a větší nebo rovno. Podkapitola 3.1.5 obsahuje popis převodu pro logický operátor negace, součinu a součtu, unární operátory plus a mínus, přetypování a operátor přiřazení. V podkapitole 3.1.6 je popis převodu operátorů pro bitový posun vlevo a vpravo.

3.1.1 Bitové operátory

Bitová negace

V článku od p. M. N. Veleva a p. R. E. Bryanta [14] je operátor bitové negace pro jeden vstupní bit *i* a výstupní bit *o*, definován v CNF v rovnici (3.1).

$$(\neg i \vee \neg o) \wedge (i \vee o) \tag{3.1}$$

Rovnice (3.1) vychází z omezení rovnosti, *equivalent constrains*, definovaných v rovnici (3.2) [14].

$$i \Rightarrow \neg o \tag{3.2}$$
$$\neg i \Rightarrow o$$

Lze vidět, že rovnice (3.1) je odvozená od výrazů v rovnici (3.2) za pomoci nahrazení implikace disjunkcí, která je uvedena v rovnici (3.3) a spojení výsledných výrazů konjunkcí.

$$a \Rightarrow b \Leftrightarrow \neg a \vee b \tag{3.3}$$

Pravdivostní tabulka pro logickou formuli v rovnici (3.1) je níže, viz Tabulka 3.1. Na této tabulce lze vidět, že formule v rovnici (3.1) je pravdivá právě tehdy, když výstupní hodnota *o* nabývá správné hodnoty pro vstupní hodnotu i, vzhledem k operaci bitové negace. Rovnice (3.1) tedy správně definuje logický výraz v CNF pro jednobitovou negaci.

Tabulka 3.1 - Pravdivostní tabulka pro logickou formuli v rovnici (3.1)

i	*o*	Rovnice (3.1)
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

Jelikož je tato operace nad jedním bitem nezávislá na okolních bitech, nebude ji problém aplikovat na proměnnou s libovolným celočíselným datovým typem, a tedy libovolným počtem bitů. Následné vygenerované formule pro každý bit se spojí konjunkcí. Před převodem samotného operátoru se nesmí zapomenout na provedení konverze operandů, jak jsou definovány v kapitole 2.2.2.

Bitový součin

V článku od p. M. N. Veleva a p. R. E. Bryanta [14] se nachází logická formule v CNF pro n-ární bitový součin. Jelikož je tato operace v programovacím jazyku C binární, tak bude zde uvedena také jen pro dva vstupní operandy. Binární součin jako logická formule v CNF je uvedena v rovnici (3.4) [14], pro operandy i a j, a výsledek o.

$$(i \vee \neg o) \wedge (j \vee \neg o) \wedge (\neg i \vee \neg j \vee o) \tag{3.4}$$

Rovnice (3.4) vychází z omezení rovnosti definovaných v rovnici (3.5) [14].

$$\begin{aligned} \neg i &\Rightarrow \neg o \\ \neg j &\Rightarrow \neg o \\ i \wedge j &\Rightarrow o \end{aligned} \tag{3.5}$$

Z rovnice (3.5) lze vidět, jak jsou tyto vazby vytvářeny. Je nutné je určit pro pozitivní i pro negativní výsledek. Nepravdivost jakéhokoliv operandu má za následek nepravdivost výsledku. Musí ale platit oba operandy, aby platil i výsledek.

Provedení této operace je nezávislé na okolních bitech a lze ji tedy bez dalších úprav možné aplikovat na proměnnou s libovolným celočíselným datovým typem. Následné vygenerované formule pro každý bit se spojí konjunkcí. Před převodem samotného operátoru se nesmí zapomenout na provedení konverze operandů, jak jsou definovány v kapitole 2.2.2.

Bitový součet

Bitový součet je také uveden v článku od p. M. N. Veleva a p. R. E. Bryanta [14]. Tato operace je velmi podobná operaci bitového součinu. Omezení rovnosti jsou uvedeny v rovnici (3.6) [14].

$$\begin{aligned} i &\Rightarrow o \\ j &\Rightarrow o \\ \neg i \wedge \neg j &\Rightarrow \neg o \end{aligned} \tag{3.6}$$

Výsledný logický výraz v CNF je v rovnici (3.7) [14].

$$(\neg i \vee o) \wedge (\neg j \vee o) \wedge (i \vee j \vee \neg o) \tag{3.7}$$

Pro tuto operaci platí stejné požadavky na konverzi operandů, jako u bitového součinu a lze ji také využít s libovolným celočíselným datovým typem, protože tato operace nad jedním bitem je nezávislá na okolních bitech. Před převodem samotného operátoru se nesmí zapomenout na provedení konverze operandů, jak jsou definovány v kapitole 2.2.2.

Bitový exkluzivní součet

Binární operace exkluzivního bitového součtu má výsledek logickou pravdu právě tehdy, když je právě jeden operand pravdivý. Pro vstupní operandy *i* a *j*, a výsledek *o*, je tento vztah zachycen výrazy v rovnici (3.8).

$$\begin{aligned} \neg i \wedge j &\Rightarrow o \\ i \wedge \neg j &\Rightarrow o \end{aligned} \tag{3.8}$$

V opačném případě, tedy když jsou oba operandy pravdivé nebo nepravdivé, tak je výstup nepravdivý. Rovnice (3.9) zachycuje toto omezení.

$$i \wedge j \Rightarrow \neg o \qquad (3.9)$$
$$\neg i \wedge \neg j \Rightarrow \neg o$$

Nahrazením implikace ve výrazech z rovnic (3.8) a (3.9) pomocí rovnice (3.3) (strana 12) a spojením výsledných výrazů konjunkcí by měl vzniknout výraz v CNF, který odpovídá operaci exkluzivního bitového součtu. Výsledný výraz je uveden v rovnici (3.10).

$$(i \vee \neg j \vee o) \wedge (\neg i \vee j \vee o) \wedge (\neg i \vee \neg j \vee \neg o) \wedge (i \vee j \vee \neg o) \qquad (3.10)$$

Pravdivostní tabulka pro výraz v rovnici (3.10) je níže, viz Tabulka 3.2. Správnost tabulky byla ověřena pomocí internetové aplikace *WolframAlpha*[1]. Z pravdivostní tabulky lze vidět, že výraz v rovnici (3.10) odpovídá operaci exkluzivního bitového součtu.

Tabulka 3.2 – Pravdivostní tabulka pro výraz v rovnici (3.10)

i	j	o	Rovnice (3.10)
0	0	0	**1**
0	0	1	**0**
0	1	0	**0**
0	1	1	**1**
1	0	0	**0**
1	0	1	**1**
1	1	0	**1**
1	1	1	**0**

Jelikož je tato operace nad jedním bitem nezávislá na okolních bitech, nebude ji problém aplikovat na proměnné s libovolným celočíselným datovým typem, a tedy libovolným počtem bitů. Následné vygenerované formule pro každý bit se spojí konjunkcí. Před převodem samotného operátoru se nesmí zapomenout na provedení konverze operandů, jak jsou definovány v kapitole 2.2.2.

[1] Aplikace je dostupná na http://www.wolframalpha.com/. Použitý příkaz je `truth table`.

3.1.2 Ternární operátor

Podle článku od p. M. N. Veleva a p. R. E. Bryanta [14] je ternární operátor dán výrazy uvedených v rovnici (3.11), pro vstupní operandy *a*, *b* a *c*, a výstup *o*.

$$\begin{gathered} a \wedge b \Rightarrow o \\ a \wedge \neg b \Rightarrow \neg o \\ \neg a \wedge c \Rightarrow o \\ \neg a \wedge \neg c \Rightarrow \neg o \end{gathered} \tag{3.11}$$

Odpovídající formule v CNF je uvedena v rovnici (3.12) [14].

$$(\neg a \vee \neg b \vee o) \wedge (\neg a \vee b \vee \neg o) \wedge (a \vee \neg c \vee o) \wedge (a \vee c \vee \neg o) \tag{3.12}$$

Takto popsaný operátor, ale neodpovídá definici operátoru v programovacím jazyce C. První operand se nejdříve musí porovnat s nulou a až poté je možné aplikovat převod operátoru na výsledek porovnání podle rovnice (3.12).

Před převodem samotného operátoru se nesmí zapomenout na provedení konverze druhého a třetího operandu, jak jsou definovány v kapitole 2.2.2. Tato operace není závislá na okolních bitech vzhledem k druhému a třetímu operandu, a také není závislá na počtu bitů druhého a třetího operandu, protože tyto operandy budou mít vždy stejný datový typ. Nebude ji tedy problém aplikovat na proměnné s libovolným celočíselným datovým typem. V každém vzniklém výrazu se bude vyskytovat tentýž bit prvního operandu, protože pouze s ním se bude pracovat. Následné vygenerované formule pro každý bit se spojí konjunkcí.

3.1.3 Součet a rozdíl

Operátor binárního součtu

U tohoto operátoru již neplatí, že by výraz nebyl závislý na okolních bitech, protože při součtu může vzniknout přetečení, které se musí přenést do součtu vedlejších bitů. Doposud byly také převáděny bitové operátory, které nezohledňují, jestli se jedná o celé číslo se znaménkem nebo bez znaménka, protože to pro tyto operátory nebylo podstatné. Na strojích, kde je v programovacím jazyku C záporné číslo reprezentováno jako dvojkový doplněk, je provedení operátoru součtu identické pro znaménková i bezznaménková čísla.

Znamená to tedy, že operátor nemusí řešit, jestli jsou operandy záporné či nikoliv a vždy provede stejný výpočet. U předcházejících operátorů také nezáleželo, jestli se převod prováděl od nejdůležitějšího bitu nebo od nejméně důležitého. U operátoru součtu se ale musí postupovat od nejméně důležitých bitů, protože následné bity berou v potaz vygenerovaný příznak od méně důležitých bitů.

Součet dvou jednobitových operandů by byl totožný s operací exkluzivního bitového součtu. Výraz je uveden v rovnici (3.10) (strana 15). Tento součet by generoval přetečení, které by odpovídalo operátoru binárního součinu. Odpovídající výraz je v rovnici (3.4) (strana 13).

Pokud by ale operandy byly více bitové, tak by tyto výrazy pokryly pouze součet prvních bitů operandů. Pro ostatní bity vstupuje do součtu bit indikující přetečení součtu předchozích bitů. Tabulka 3.3 zobrazuje pravdivostní tabulku pro součet bitů z prvního operandu a_i, z druhého operandu b_i a bitu přetečení z předchozího součtu, c_{i-1}, a výsledku součtu o_i a bit přetečení pro další součet c_i.

Tabulka 3.3 – Pravdivostní tabulka pro součet bitů včetně zohlednění přetečení

a_i	b_i	c_{i-1}	o_i	c_i
0	0	0	**0**	**0**
0	0	1	**1**	**0**
0	1	0	**1**	**0**
0	1	1	**0**	**1**
1	0	0	**1**	**0**
1	0	1	**0**	**1**
1	1	0	**0**	**1**
1	1	1	**1**	**1**

Generování přetečení do dalšího součtu zachycují výrazy v rovnici (3.13). Tyto výrazy jsou odvozeny z pravdivostní tabulky výše, viz Tabulka 3.3. Pokud jsou alespoň dva vstupní bity kladné, je generováno přetečení, pokud jsou alespoň dva vstupní bity záporné, přetečení není generováno.

$$\begin{aligned} a_i \wedge b_i &\Rightarrow c_i & \neg a_i \wedge \neg b_i &\Rightarrow \neg c_i \\ a_i \wedge c_{i-1} &\Rightarrow c_i & \neg a_i \wedge \neg c_{i-1} &\Rightarrow \neg c_i \\ b_i \wedge c_{i-1} &\Rightarrow c_i & \neg b_i \wedge \neg c_{i-1} &\Rightarrow \neg c_i \end{aligned} \tag{3.13}$$

Odpovídající formule v CNF je uvedena níže v rovnici (3.14).

$$(\neg a_i \vee \neg b_i \vee c_i) \wedge (\neg a_i \vee \neg c_{i-1} \vee c_i) \wedge (\neg b_i \vee \neg c_{i-1} \vee c_i) \wedge (a_i \vee b_i \vee \neg c_i) \wedge (a_i \vee c_{i-1} \vee \neg c_i) \wedge (b_i \vee c_{i-1} \vee \neg c_i) \tag{3.14}$$

Tabulka 3.3 také umožňuje odvodit výrazy pro výstupní bit. Bohužel tento výpočet neobsahuje žádné nápomocné pravidlo pro ulehčení výrazů a je tedy potřeba pokrýt výrazy každý řádek pravdivostní tabulky. Tyto výrazy jsou uvedeny v rovnici (3.15).

$$\begin{aligned} \neg a_i \wedge \neg b_i \wedge \neg c_{i-1} &\Rightarrow \neg o_i & a_i \wedge \neg b_i \wedge \neg c_{i-1} &\Rightarrow o_i \\ \neg a_i \wedge \neg b_i \wedge c_{i-1} &\Rightarrow o_i & a_i \wedge \neg b_i \wedge c_{i-1} &\Rightarrow \neg o_i \\ \neg a_i \wedge b_i \wedge \neg c_{i-1} &\Rightarrow o_i & a_i \wedge b_i \wedge \neg c_{i-1} &\Rightarrow \neg o_i \\ \neg a_i \wedge b_i \wedge c_{i-1} &\Rightarrow \neg o_i & a_i \wedge b_i \wedge c_{i-1} &\Rightarrow o_i \end{aligned} \tag{3.15}$$

Odpovídající formule v CNF je uvedena v rovnici (3.16).

$$(a_i \vee b_i \vee c_{i-1} \vee \neg o_i) \wedge (\neg a_i \vee b_i \vee c_{i-1} \vee o_i) \wedge (a_i \vee b_i \vee \neg c_{i-1} \vee o_i) \wedge (\neg a_i \vee b_i \vee \neg c_{i-1} \vee \neg o_i) \wedge (a_i \vee \neg b_i \vee c_{i-1} \vee o_i) \wedge (\neg a_i \vee \neg b_i \vee c_{i-1} \vee \neg o_i) \wedge (a_i \vee \neg b_i \vee \neg c_{i-1} \vee \neg o_i) \wedge (\neg a_i \vee \neg b_i \vee \neg c_{i-1} \vee o_i) \tag{3.16}$$

Pomocí rovnice (3.14) je možné zapsat generování přetečení a pomocí rovnice (3.16) je možné zapsat součet dvou čísel pro libovolný celočíselný datový typ. Součet posledních bitů nemusí generovat přetečení a součet prvních bitů je možné zjednodušit, jak je uvedeno na začátku popisu tohoto operátoru. Vzniklé výrazy pro všechny bity se spojí konjunkcí. Před převodem samotného operátoru se nesmí zapomenout na provedení konverze operandů, jak jsou definovány v kapitole 2.2.2.

Operátor binárního rozdílu

Tento operátor je velice podobný operátoru binárního součtu, vzhledem k jeho vlastnostem. Výpočet rozdílu dvou bitů závisí na přetečení rozdílu předchozích bitů. Pokud je záporné číslo reprezentováno jako dvojkový doplněk, tak je operace totožná pro znaménková i bezznaménková čísla.

Výsledek rozdílu prvního bitu operandů je totožný s operací exkluzivního bitového součtu, viz Tabulka 3.4 níže. Výraz je uveden v rovnici (3.10) (strana 15). Tabulka 3.4 ale také ukazuje, že pro výpočet přetečení u výpočtu rozdílu prvního bitu je potřeba sestrojit nový výraz v CNF.

Tabulka 3.4 – Pravdivostní tabulka pro jednobitový rozdíl. o je výsledek rozdílu a c je generované přetečení.

a	b	o	c
0	0	**0**	**0**
0	1	**1**	**1**
1	0	**1**	**0**
1	1	**0**	**0**

Výrazy pro přetečení jsou uvedeny v rovnici (3.17).

$$\neg b \Rightarrow \neg c \qquad (3.17)$$
$$\neg a \wedge b \Rightarrow c$$
$$a \wedge b \Rightarrow \neg c$$

Odpovídající výraz v CNF je v rovnici (3.18).

$$(b \vee \neg c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \qquad (3.18)$$

Tabulka 3.5 níže zobrazuje pravdivostní tabulku pro rozdíl bitů z prvního operandu a_i, z druhého operandu b_i a bitu přetečení z předchozího rozdílu, c_{i-1}, a výsledku rozdílu o_i a bit přetečení pro další rozdíl c_i.

Tabulka 3.5 – Pravdivostní tabulka pro rozdíl bitů včetně zohlednění přetečení

a_i	b_i	c_{i-1}	o_i	c_i
0	0	0	**0**	**0**
0	0	1	**1**	**1**
0	1	0	**1**	**1**
0	1	1	**0**	**1**
1	0	0	**1**	**0**
1	0	1	**0**	**0**
1	1	0	**0**	**0**
1	1	1	**1**	**1**

Z této tabulky lze vidět, že výstup je totožný s výstupem pro součet a tedy výraz v CNF pro výsledek rozdílu je v rovnici (3.16) (strana 18). Výrazy pro přetečení jsou v rovnici (3.19).

$$\begin{aligned} \neg a_i \wedge b_i &\Rightarrow c_i & a_i \wedge \neg b_i &\Rightarrow \neg c_i \\ \neg a_i \wedge c_{i-1} &\Rightarrow c_i & a_i \wedge \neg c_{i-1} &\Rightarrow \neg c_i \\ b_i \wedge c_{i-1} &\Rightarrow c_i & \neg b_i \wedge \neg c_{i-1} &\Rightarrow \neg c_i \end{aligned} \tag{3.19}$$

Odpovídající výraz v CNF je v rovnici (3.20).

$$\begin{aligned} &(a_i \vee \neg b_i \vee c_i) \wedge (\neg a_i \vee b_i \vee \neg c_i) \wedge \\ &(a_i \vee \neg c_{i-1} \vee c_i) \wedge (\neg a_i \vee c_{i-1} \vee \neg c_i) \wedge \\ &(\neg b_i \vee \neg c_{i-1} \vee c_i) \wedge (b_i \vee c_{i-1} \vee \neg c_i) \end{aligned} \tag{3.20}$$

Pomocí rovnice (3.20) je možné zapsat generování přetečení a pomocí rovnice (3.16) je možné zapsat rozdíl dvou čísel pro libovolný celočíselný datový typ. Rozdíl posledních bitů nemusí generovat přetečení a rozdíl prvních bitů je možné zjednodušit, jak je uvedeno na začátku popisu tohoto operátoru. Vzniklé výrazy pro všechny bity se spojí konjunkcí. Před převodem samotného operátoru se nesmí zapomenout na provedení konverze operandů, jak jsou definovány v kapitole 2.2.2.

3.1.4 Relační operátory

Operátor rovnosti a nerovnosti

Operátor porovnávaní rovnosti i operátor nerovnosti porovnává operandy na bitové úrovni, a tak nemusí zohledňovat, jestli se jedná o znaménková nebo bezznaménková celá čísla. Před převodem samotného operátoru se nesmí zapomenout na provedení konverze operandů tak, jak jsou definovány v kapitole 2.2.2.

Princip převodu těchto operátorů se může podobat principu sčítání a odčítání. Jednotlivé bity se musí porovnat, jestli jsou stejné, a výsledek tohoto porovnání se jako příznak přenese do dalšího porovnání. Pokud příznak značí, že hodnoty stejné nejsou, tak už se nemusí porovnávat zbylé bity operandů. Porovnávat se může od nejdůležitějších bitů nebo od nejméně důležitých bitů. Po dokončení porovnávání všech bitů příznak obsahuje, jestli jsou operandy stejné nebo rozdílné.

Tabulka 3.6 níže ukazuje pravdivostní tabulku pro porovnání operátorů se vstupními bity a_i a b_i, příznakem předcházejícího porovnání f_{i-1} a výsledným příznakem tohoto porovnání f_i. Hodnota příznaku jedna značí, že jsou hodnoty stejné, a nula, že rozdílné. Takto navrhnutou operaci je tedy možné použít, jak pro operátor rovnosti, tak nerovnosti. Lišit se budou pouze v reprezentaci výsledného příznaku.

Tabulka 3.6 – Pravdivostní tabulka pro generování příznaku pro operaci porovnávání

f_{i-1}	a_i	b_i	f_i
0	x	x	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Z této tabulky lze vidět, že pokud příznak značí nerovnající se operandy, tak se tento příznak jen pošle dál. Pokud příznak značí rovnající se operandy, obsahuje tabulka v podstatě bitovou operaci ekvivalence. Výsledné výrazy z této tabulky jsou níže v rovnici (3.21).

$$\neg f_{i-1} \Rightarrow \neg f_i \qquad (3.21)$$

$$f_{i-1} \wedge \neg a_i \wedge \neg b_i \Rightarrow f_i \qquad f_{i-1} \wedge a_i \wedge \neg b_i \Rightarrow \neg f_i$$

$$f_{i-1} \wedge \neg a_i \wedge b_i \Rightarrow \neg f_i \qquad f_{i-1} \wedge a_i \wedge b_i \Rightarrow f_i$$

Odpovídající výraz v CNF je v rovnici (3.22).

$$(f_{i-1} \vee \neg f_i) \wedge \tag{3.22}$$
$$(\neg f_{i-1} \vee a_i \vee b_i \vee f_i) \wedge (\neg f_{i-1} \vee \neg a_i \vee b_i \vee \neg f_i) \wedge$$
$$(\neg f_{i-1} \vee a_i \vee \neg b_i \vee \neg f_i) \wedge (\neg f_{i-1} \vee \neg a_i \vee \neg b_i \vee f_i)$$

Výsledný příznak f_i poté obsahuje výsledek operátoru rovnosti. Pro operátor nerovnosti je třeba nad tímto příznakem provést navíc jeho negaci podle kapitoly 3.1.1.

Nabízí se také odlišný přístup. Je možné provést porovnání pro každou dvojici bitů z operandů zvlášť a nad tímto výsledkem poté provést binární součin. Výsledek poté obsahuje, jestli jsou operandy stejné nebo rozdílné. Nebo je možné také odečíst od sebe oba operandy a porovnat výsledek s nulou.

Operátory menší, menší nebo roven, větší a větší nebo roven

Převod operátorů porovnávajících hodnotu bude proveden podobným způsobem, jako je navržen převod operátorů pro porovnání rovnosti a nerovnosti výše. Před převodem samotného operátoru se nesmí zapomenout na provedení konverze operandů, jak jsou definovány v kapitole 2.2.2. Převod musí zohlednit, jestli porovnává znaménková nebo bezznaménková čísla. Pokud jsou operandy znaménkové, tak pro nejdůležitější bit platí opačná relace, to je jednička je menší než nula. Zbylé bity se již dále porovnávají nehledě na užitý datový typ.

Porovnávat se musí od nejdůležitějších bitů k těm méně důležitým, zprava doleva. Mezi jednotlivými porovnáními se ale musí přenášet dva příznaky. Jeden značící, jestli jsou si porovnávaná čísla rovna a druhý, jestli je větší první nebo druhý operand. Dokud budou čísla stejná, tak druhý příznak nemá žádnou vypovídající hodnotu. V momentě, kdy budou porovnávané bity různé, musí se nastavit první příznak, aby značil, že jsou čísla různá, a druhý příznak, jestli je větší první nebo druhý operand.

Tabulka 3.7 níže ukazuje pravdivostní tabulku pro porovnání operátorů se vstupními bity a_i a b_i, dvěma předcházejícími příznaky f_{i-1}^1 a f_{i-1}^2, a výslednými příznaky tohoto porovnání f_i^1 a f_i^2. Hodnota prvního příznaku jedna značí, že jsou hodnoty stejné a nula, že rozdílné. Jednička v druhém příznaku značí, že první operand je větší. Takto navrhnutou operaci je tedy možné použít pro všechny operátory porovnávající velikost operandů. Lišit se budou pouze v reprezentaci výsledných příznaků.

Tabulka 3.7 – Pravdivostní tabulka pro operaci porovnávání velikosti

f_{i-1}^1	f_{i-1}^2	a_i	b_i	f_i^1	f_i^2
0	0	x	x	0	0
0	1	x	x	0	1
1	x	0	0	1	x
1	x	0	1	0	0
1	x	1	0	0	1
1	x	1	1	1	x

Z této tabulky lze vidět, že pokud příznak značí nerovnající se operandy, tak se oba příznaky jen pošlou dál. Pokud příznak značí rovnající se operandy, tak nezáleží na druhém příznaku a tabulka obsahuje pro první příznak v podstatě bitovou operaci ekvivalence. Hodnota druhého příznaku je počítána pouze v případě, že se operandy nově nerovnají. Výsledné výrazy z této tabulky jsou níže v rovnici (3.23).

$$\neg f_{i-1}^1 \Rightarrow \neg f_i^1 \tag{3.23}$$

$$\neg f_{i-1}^1 \wedge \neg f_{i-1}^2 \Rightarrow \neg f_i^2 \qquad \neg f_{i-1}^1 \wedge f_{i-1}^2 \Rightarrow f_i^2$$

$$f_{i-1}^1 \wedge \neg a_i \wedge b_i \Rightarrow \neg f_i^2 \qquad f_{i-1}^1 \wedge a_i \wedge \neg b_i \Rightarrow f_i^2$$

$$f_{i-1}^1 \wedge \neg a_i \wedge \neg b_i \Rightarrow f_i^1 \qquad f_{i-1}^1 \wedge a_i \wedge \neg b_i \Rightarrow \neg f_i^1$$

$$f_{i-1}^1 \wedge \neg a_i \wedge b_i \Rightarrow \neg f_i^1 \qquad f_{i-1}^1 \wedge a_i \wedge b_i \Rightarrow f_i^1$$

Odpovídající výraz v CNF je v rovnici (3.24).

$$\begin{gathered}\left(f_{i-1}^1 \vee \neg f_i^1\right) \wedge \\ \left(f_{i-1}^1 \vee f_{i-1}^2 \vee \neg f_i^2\right) \wedge \left(f_{i-1}^1 \vee \neg f_{i-1}^2 \vee f_i^2\right) \wedge \\ \left(\neg f_{i-1}^1 \vee a_i \vee \neg b_i \vee \neg f_i^2\right) \wedge \left(\neg f_{i-1}^1 \vee \neg a_i \vee b_i \vee f_i^2\right) \wedge \\ \left(\neg f_{i-1}^1 \vee a_i \vee b_i \vee f_i^1\right) \wedge \left(\neg f_{i-1}^1 \vee \neg a_i \vee b_i \vee \neg f_i^1\right) \\ \left(\neg f_{i-1}^1 \vee a_i \vee \neg b_i \vee \neg f_i^1\right) \wedge \left(\neg f_{i-1}^1 \vee \neg a_i \vee \neg b_i \vee f_i^1\right)\end{gathered} \tag{3.24}$$

Výsledný příznak f^1 poté obsahuje, jestli jsou si operandy rovny. A pokud jsou operandy rozdílné, tak příznak f^2 obsahuje výsledek určující, který operand je větší. Tabulka 3.8 níže obsahuje výsledek operátoru vzhledem k výsledným příznakům.

Tabulka 3.8 – Pravdivostní tabulka pro výsledné příznaky porovnávání operandů

f^1	f^2	$>$	$\geq$	$\leq$	$<$
0	0	0	0	1	1
0	1	1	1	0	0
1	x	0	1	1	0

Výsledné výrazy pro operaci větší než jsou v rovnici (3.25), pro operaci větší nebo rovno v rovnici (3.26), pro operaci menší nebo rovno v rovnici (3.27) a pro operaci menší než v rovnici (3.28).

$$f^1 \Rightarrow \neg o, \neg f^1 \wedge \neg f^2 \Rightarrow \neg o, \neg f^1 \wedge f^2 \Rightarrow o \tag{3.25}$$

$$f^1 \Rightarrow o, \neg f^1 \wedge \neg f^2 \Rightarrow \neg o, \neg f^1 \wedge f^2 \Rightarrow o \tag{3.26}$$

$$f^1 \Rightarrow o, \neg f^1 \wedge \neg f^2 \Rightarrow o, \neg f^1 \wedge f^2 \Rightarrow \neg o \tag{3.27}$$

$$f^1 \Rightarrow \neg o, \neg f^1 \wedge \neg f^2 \Rightarrow o, \neg f^1 \wedge f^2 \Rightarrow \neg o \tag{3.28}$$

Výsledné formule v CNF jsou pro rovnici (3.25) v rovnici (3.29), pro rovnici (3.26) v rovnici (3.30), pro rovnici (3.27) v rovnici (3.31) a pro rovnici (3.28) v rovnici (3.32).

$$\left(\neg f^1 \vee \neg o\right) \wedge \left(f^1 \vee f^2 \vee \neg o\right) \wedge \left(f^1 \vee \neg f^2 \vee o\right) \tag{3.29}$$

$$\left(\neg f^1 \vee o\right) \wedge \left(f^1 \vee f^2 \vee \neg o\right) \wedge \left(f^1 \vee \neg f^2 \vee o\right) \tag{3.30}$$

$$\left(\neg f^1 \vee o\right) \wedge \left(f^1 \vee f^2 \vee o\right) \wedge \left(f^1 \vee \neg f^2 \vee \neg o\right) \tag{3.31}$$

$$\left(\neg f^1 \vee \neg o\right) \wedge \left(f^1 \vee f^2 \vee o\right) \wedge \left(f^1 \vee \neg f^2 \vee \neg o\right) \tag{3.32}$$

3.1.5 Operátor přiřazení, logické a unární operátory

Operátor přiřazení podle kapitoly 2.2.3 v podstatě provede jen přetypování pravého operandu na datový typ levého operandu. Operátor přetypování je popsán níže v této kapitole.

Podle kapitoly 2.2.3 je **operátor logického součinu** kombinací operátorů nerovnosti a bitového součinu. Převod operátoru nerovnosti je v kapitole 3.1.4 a operátor bitového součinu je v kapitole 3.1.1.

Operátor logického součtu je podle kapitoly 2.2.3 kombinací operátorů nerovnosti a bitového součtu. Převod operátoru nerovnosti je v kapitole 3.1.4 a operátor bitového součtu je v kapitole 3.1.1.

Operátor logické negace je podle kapitoly 2.2.3 porovnání rovnosti s nulou. Převod operátoru rovnosti je v kapitole 3.1.4.

Operátor unární plus provádí podle kapitoly 2.2.3 přetypování menších typů než je celé znaménkové číslo na celé znaménkové nebo bezznaménkové číslo. Převod operátoru přetypování je popsán níže v této kapitole.

Operátor unární mínus provede podle kapitoly 2.2.3 přetypování menších typů než je celé znaménkové číslo na celé znaménkové nebo bezznaménkové číslo a vrátí jeho zápornou hodnotu. Záporná hodnota z čísla se získá podle reprezentace záporných hodnot na stroji. Pokud jsou záporná čísla reprezentována dvojkovým doplňkem, tak se provede bitová negace a k výsledku se přičte jednička. Převod operátoru přetypování je popsán níže v této kapitole, operátor bitové negace je popsán v kapitole 3.1.1 a operátor součtu je popsán v kapitole 3.1.3.

Operátor explicitního přetypování se podle kapitoly 2.2.2 chová stejně jako implicitní přetypování. Pokud je provedeno přetypování na datový typ, který je menší než typ ze kterého se přetypovává, tak se přebývající bity jednoduše zahodí. Pokud se provádí přetypování na datový typ, který je stejně široký, jako datový typ ze kterého se přetypovává, tak se neprovádí žádná operace. A pokud se provádí přetypování na větší typ, tak záleží, jestli se přetypování provádí z typu se znaménkem nebo ne a případně, jak jsou reprezentovány záporné hodnoty.

Při přetypování na větší typ z datového typu bez znaménka se nové bity nastaví na nulu. Pokud se přetypovává z datového typu se znaménkem a záporná čísla jsou reprezentována dvojkovým doplňkem, tak nové bity se nastaví na hodnotu, která je v nejdůležitějším bitu původního datového typu.

3.1.6 Bitový posun vpravo a vlevo

Operátor bitového posunu vlevo a operátor bitového posunu vpravo nemá definovaný výsledek v případě, že levý anebo pravý operand je záporné číslo, viz kapitola 2.2.3. Z tohoto důvodu bude navržen převod, který bude pracovat s levým i pravým operandem, jako s datovými typy bez znaménka a nebude dále upravovat vstupní operandy.

Pokud bude u těchto operátorů pravý operand konstanta, tak je možné tyto operátory převést jako jednoduché přiřazení na bitové úrovni. V případě ale, že pravý operand je proměnná, tak u převodu bitového posunu vlevo nebo vpravo bohužel nestačí ani pomocné příznaky, které by se přesouvaly u jednotlivého vyhodnocování jednotlivých bitů. Tyto operátory se musí převést pro každý bit zvlášť. Mějme dva čtyřbitové operandy $a = a_4a_3a_2a_1$ a $b = b_4b_3b_2b_1$, a výsledek $o = o_4o_3o_2o_1$, a uvažujme výraz uvedený v rovnici (3.34).

$$o = a \ll b \tag{3.33}$$

Pro výpočet prvního bitu pak platí výrazy uvedené v rovnici (3.34).

$$\begin{aligned} &\neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge \neg b_1 \wedge a_1 \Rightarrow o_1 && b_1 \Rightarrow \neg o_1 \\ &\neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge \neg b_1 \wedge \neg a_1 \Rightarrow \neg o_1 && b_2 \Rightarrow \neg o_1 \\ & && b_3 \Rightarrow \neg o_1 \\ & && b_4 \Rightarrow \neg o_1 \end{aligned} \tag{3.34}$$

Pro výpočet druhého bitu platí výrazy uvedené v rovnici (3.35).

$$\begin{aligned} &\neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge \neg b_1 \wedge a_2 \Rightarrow o_2 && b_2 \Rightarrow \neg o_2 \\ &\neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge \neg b_1 \wedge \neg a_2 \Rightarrow \neg o_2 && b_3 \Rightarrow \neg o_2 \\ &\neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge b_1 \wedge a_1 \Rightarrow o_2 && b_4 \Rightarrow \neg o_2 \\ &\neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge b_1 \wedge \neg a_1 \Rightarrow \neg o_2 \end{aligned} \tag{3.35}$$

Pro výpočet třetího bitu platí výrazy uvedené v rovnici (3.36).

$$\begin{aligned} \neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge \neg b_1 \wedge a_3 &\Rightarrow o_3 \qquad & b_1 \wedge b_2 &\Rightarrow \neg o_3 \\ \neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge \neg b_1 \wedge \neg a_3 &\Rightarrow \neg o_3 & b_3 &\Rightarrow \neg o_3 \\ \neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge b_1 \wedge a_2 &\Rightarrow o_3 & b_4 &\Rightarrow \neg o_3 \\ \neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge b_1 \wedge \neg a_2 &\Rightarrow \neg o_3 \\ \neg b_4 \wedge \neg b_3 \wedge b_2 \wedge \neg b_1 \wedge a_1 &\Rightarrow o_3 \\ \neg b_4 \wedge \neg b_3 \wedge b_2 \wedge \neg b_1 \wedge \neg a_1 &\Rightarrow \neg o_3 \end{aligned} \tag{3.36}$$

A pro výpočet posledního, čtvrtého, bitu platí výrazy uvedené v rovnici (3.37).

$$\begin{aligned} \neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge \neg b_1 \wedge a_4 &\Rightarrow o_4 \qquad & & \\ \neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge \neg b_1 \wedge \neg a_4 &\Rightarrow \neg o_4 & b_3 &\Rightarrow \neg o_4 \\ \neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge b_1 \wedge a_3 &\Rightarrow o_4 & b_4 &\Rightarrow \neg o_4 \\ \neg b_4 \wedge \neg b_3 \wedge \neg b_2 \wedge b_1 \wedge \neg a_3 &\Rightarrow \neg o_4 \\ \neg b_4 \wedge \neg b_3 \wedge b_2 \wedge \neg b_1 \wedge a_2 &\Rightarrow o_4 \\ \neg b_4 \wedge \neg b_3 \wedge b_2 \wedge \neg b_1 \wedge \neg a_2 &\Rightarrow \neg o_4 \\ \neg b_4 \wedge \neg b_3 \wedge b_2 \wedge b_1 \wedge a_1 &\Rightarrow o_4 \\ \neg b_4 \wedge \neg b_3 \wedge b_2 \wedge b_1 \wedge \neg a_1 &\Rightarrow \neg o_4 \end{aligned} \tag{3.37}$$

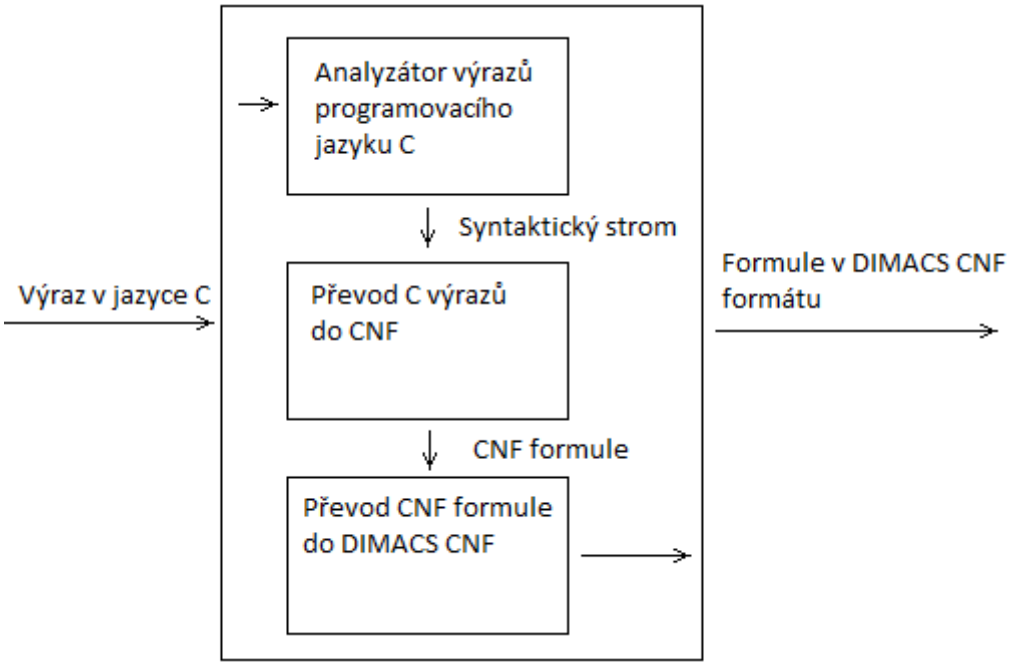
To je celkem třicet dva výrazů pro popis operátoru bitového posunu vlevo s čtyřbitovými operandy. Výrazy z rovnic (3.34), (3.35), (3.36) a (3.37) lze již lehce následně převést do CNF.

3.2 Návrh programu pro převod

Pro uskutečnění převodu se v prvním kroku načte výraz, který je zapsán v programovacím jazyce C. Tento výraz se následně nechá zpracovat lexikálním, syntaktickým a sémantickým analyzátorem programovacího jazyka C. Pokud vstupní výraz neobsahuje chyby, tak výstupem tohoto převodu bude syntaktický strom. Nad syntaktickým stromem se provede převod na logické formule v CNF. Tyto formule se následně převedou do formátu DIMACS CNF. Schéma odpovídající návrh takovéhoto programu je znázorněno na obrázku níže, viz Obrázek 3.1.

Analyzátor bude podporovat pouze podmnožinu programovacího jazyka C. V rámci této práce nemají být podporovány všechny datové typy. Podporované budou pouze celočíselné datové typy, a tak by se měl chovat i tento analyzátor. Toto omezení je popsané v podkapitole 2.2.2. Navíc by měl podporovat datové typy, které jsou definovány v hlavičkovém souboru `stdint.h`. V rámci této práce nemají být podporovány ani všechny operátory programovacího jazyka C. Analyzátor by měl proto podporovat pouze operátory, které jsou rozepsány v kapitole 2.2.3. Další omezení analyzátoru bude podpora zpracování pouze výrazů programovacího jazyka C.

Obrázek 3.1 - Návrh programu pro provedení převodu



V rámci sémantické analýzy mohou být do syntaktického stromu přidány implicitní převody datových typů. Popis chování převodů datových typů programovacího jazyka C je v kapitole 2.2.2.

Před převodem na logickou formuli v CNF je potřeba rozložit jednotlivé proměnné a konstanty na nové proměnné, které budou reprezentovat jednotlivé bity původních proměnných nebo konstant. Navíc, jelikož formát DIMACS CNF požaduje proměnné jako celá kladná čísla, viz kapitola 2.4, tak nové proměnné při rozkladu by mohli dostat rovnou unikátní identifikační číslo, které bude uplatněno při převodu do formátu DIMACS CNF.

Převod samotných operátorů v syntaktickém stromu na formuli v CNF je popsán v podkapitole 3.1. Převod lze provést podobně jako interpretace. Výsledky jednotlivých částí výrazu tvoří nové dočasné proměnné. Výsledný výraz obsahuje tyto proměnné navíc.

Převod z formule v CNF do formátu DIMACS CNF bude proveden podle popisu tohoto formátu v kapitole 2.4. Pro další práci s tímto výstupem bude vhodné, aby program generoval také komentáře. Tyto komentáře by měli obsahovat proměnné z výrazů v programovacím jazyku C a k nim mohou být uvedeny odpovídající proměnné z klauzulí v DIMACS CNF formátu.

4 Implementace

Tato kapitola obsahuje popis implementace aplikace pro realizaci převodu z výrazů v programovacím jazyce C do formátu DIMACS. Kromě samotného popisu implementace se v této kapitole nachází i způsob ovládání a popis provedeného testování a použitých nástrojů pro testování aplikace.

Jako typ aplikace byla zvolena konzolová aplikace. Aplikace byla vyvíjena pod operačním systémem Microsoft Windows 8.1 a k vývoji a kompilaci bylo použito vývojové prostředí Microsoft Visual Studio 2013. Aplikace byla také testována pod operačním systémem Lubuntu 14.10[1] a pro překlad byly použity překladače gcc verze 4.9.1 a clang verze 3.5.0. Při vývoji aplikace byl použit systém správy verzí git[2].

V podkapitole 4.1 se nachází popis struktury aplikace a hrubá funkčnost jednotlivých částí. Podkapitola 4.2 obsahuje některé detaily implementace, použité algoritmy a detaily funkčnosti jednotlivých částí aplikace. V podkapitole 4.3 je popis použitého testování a přístupu k testování aplikace. Podkapitola 4.4 obsahuje ukázku použití aplikace a v podkapitole 4.5 se nachází vyhodnocení aplikace.

4.1 Struktura aplikace

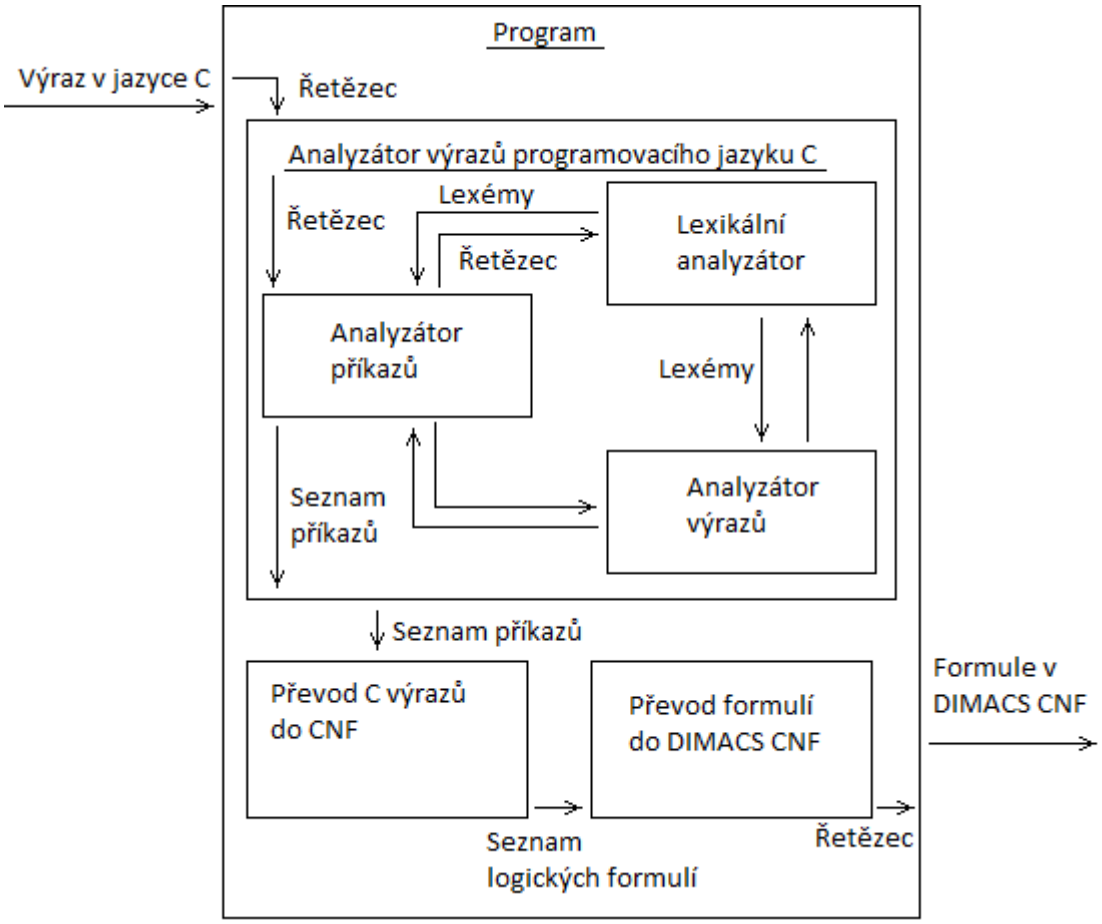
Aplikace byla vytvořena, aby vstupní výrazy pro převod byly načítány ze standardního vstupu. Jakákoliv chyba, která nastane při běhu aplikace, je vypsána na standardní chybový výstup. Výstup aplikace je vypsán na standardní výstup. Aplikace nepřijímá a ani nikterak nekontroluje vstupní parametry. Pro přeložení výrazu zadaného v souboru je třeba přesměrovat tento soubor na vstup, jako standardní vstup aplikace.

Vstup aplikace je prvně zpracován analyzátorem programovacího jazyka C. Tento analyzátor podporuje požadovanou podmnožinu jazyka C, potřebnou v rámci této práce. Výstupem tohoto analyzátoru je seznam příkazů, které se mají vykonat. Tento seznam příkazů je následně po jednom převeden na logické formule v CNF. Převod probíhá podle navržených převodů v kapitole 3.1. Jednotlivé výsledné formule jsou ukládány do jiného seznamu. Po ukončení převodu na formule v CNF přichází na řadu převod do DIMACS formátu. Převod do DIMACS formátu probíhá přímým vypisováním do standardního výstupu. Podrobnější popis jednotlivých výše popsaných částí programu se nachází v podkapitole 4.2.

[1] Lubuntu je větev operačního systému Ubuntu a Linuxu, více informací na http://lubuntu.net/.
[2] Více informací o vezovacím systému git na http://git-scm.com/.

Popsané chování aplikace je zachyceno na hrubém schématu programu zobrazeného na obrázku níže, viz Obrázek 4.1. Tento program zhruba odpovídá návrhu z předchozí kapitoly, viz Obrázek 3.1 na straně 28, ale oproti návrhu obsahuje více detailů a také zachycuje skutečnou realizaci uvnitř programu. Obrázek 4.2 na straně 32, dále v této podkapitole, zachycuje podrobnější popis použitých struktur a objektů v programu a Obrázek 4.3 na straně 33 zobrazuje vyobrazený průběh zpracování vstupního řetězce a následných formulí a také celkový průběh běhu programu.

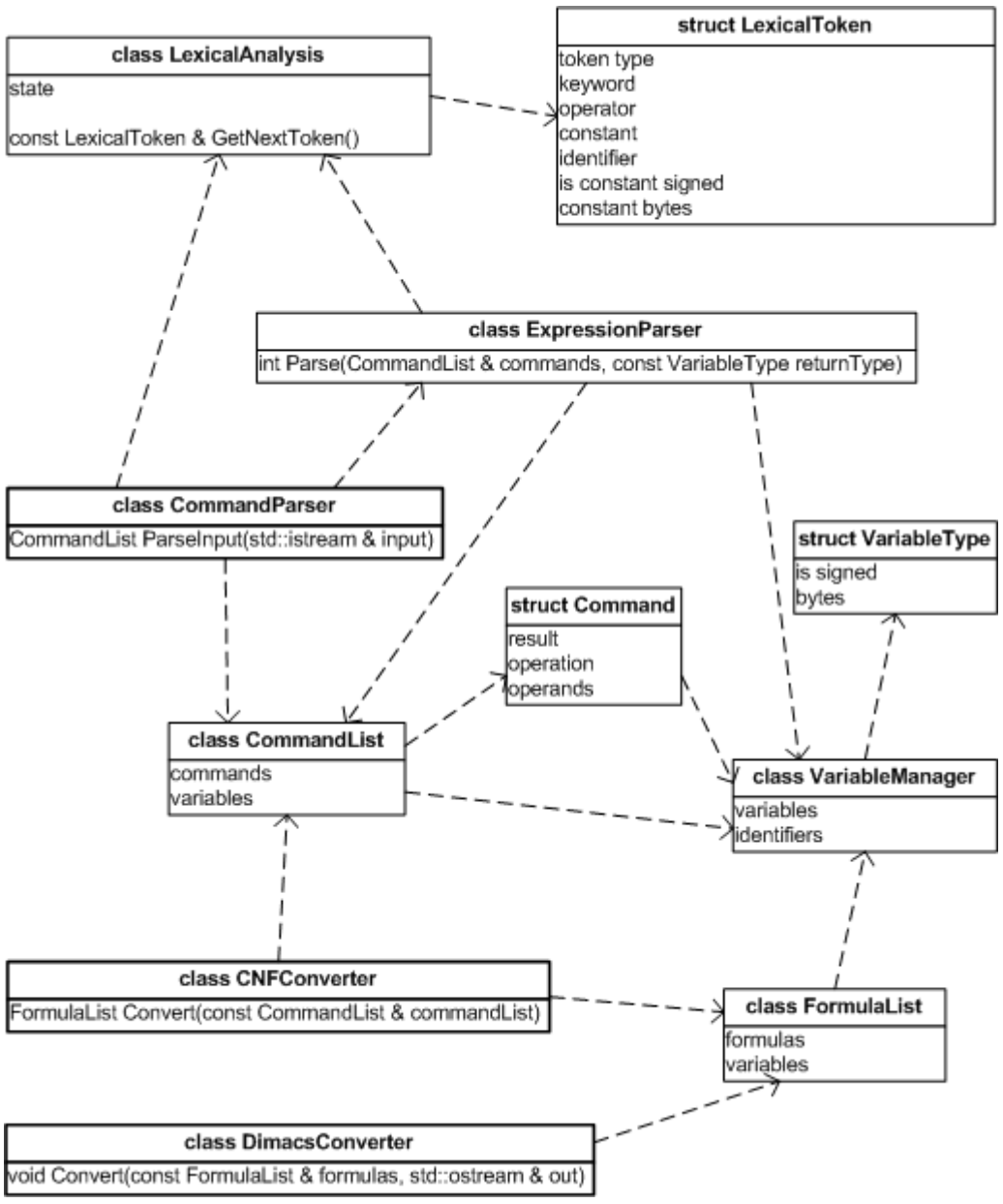
Obrázek 4.1 – Schéma provedené implementace programu



Obrázek 4.2 níže, zobrazuje detailnější strukturu programu. Tento obrázek obsahuje jednotlivé objekty pod názvy, tak jak jsou definovány v programu. Na obrázku jsou také zobrazeny některé jejich funkce, které představují důležitou funkcionalitu, nebo proměnné reprezentující jejich vnitřní stav či podstatnou informaci, kterou nesou. Jednotlivé šipky reprezentují závislosti jednotlivých tříd na sobě. Pro přehlednost nejsou na obrázku vyobrazeny veškeré závislosti, neboť se předpokládá automatická tranzitivní závislost.

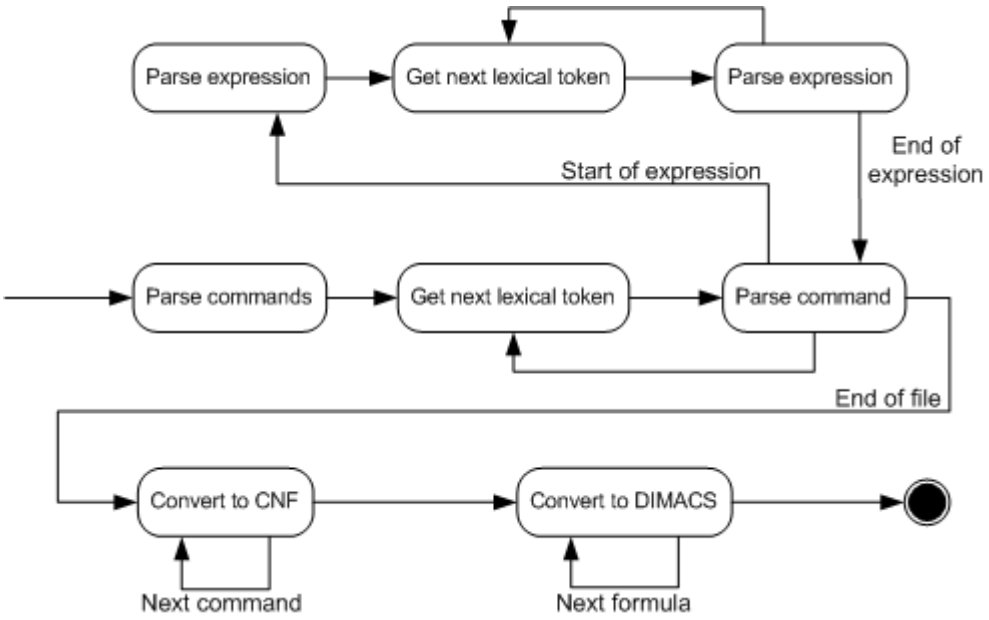
Tři nejdůležitější části programu, analyzátor jazyka C, převodník do CNF a převodník do DIMACS, jsou zvýrazněny. Analyzátor jazyka C, který sám se skládá ze tří dílčích částí, je zvýrazněn pouze jako analyzátor příkazů, neboť ostatní části jsou jeho nedílnou součástí a obrázek zachycuje jeho závislost na nich. Podrobnější popis se nachází v podkapitole 4.2.

Obrázek 4.2 – Strukturální schéma aplikace se závislostmi



Podrobnější přehled a posloupnost vykonávání jednotlivých částí je zachycena níže, viz Obrázek 4.3. Tento obrázek zachycuje podrobněji pořadí a způsob analýzy vstupního řetězce analyzátorem jazyka C jednotlivými jeho částmi. Z tohoto obrázku lze i částečně vidět jednotlivé závislosti jednotlivých částí, které jsou jinak vyobrazeny výše, viz Obrázek 4.2.

Obrázek 4.3 – Průběh zpracování vstupního řetězce a celkový průběh programu



4.2 Implementační detaily

Analyzátor jazyka C je členěn na tři velké části, viz Obrázek 4.1. Jednou z částí je lexikální analyzátor, který rozpracovává vstupní řetězec na zanalyzované části řetězce, lexémy. Tento analyzátor čte vstup znak po znaku a podle jednoduchých pravidel v podobě stavového automatu se rozhoduje, co právě zanalyzoval. Druhou částí je analyzátor výrazů. Pro svou funkcionalitu využívá lexikálního analyzátoru a analyzuje výrazy jazyka C. Tato část analyzátoru využívá pro svou funkcionalitu algoritmus seřaďovacího nádraží, *shunting-yard algorithm* [15][16]. Tento algoritmus je blíže popsán v podkapitole 4.2.2 na straně 35. Třetí částí je analyzátor příkazů. Tato část zpracovává deklarace a definice proměnných. Pro tuto činnost využívá funkcionality lexikálního i výrazového analyzátoru. Obrázek 4.2 i Obrázek 4.3 zachycuje tuto závislost. Analyzátor výrazů i analyzátor příkazů používají stejnou vnitřní reprezentaci proměnných a konstant. Tato reprezentace je podrobněji popsána v podkapitole 4.2.1 na straně 35.

V rámci této práce jsou podporovány pouze výrazy pro převod, a proto implementovaný program a analyzátor jazyka C podporuje pouze podmnožinu programovacího jazyka C. V rámci analyzátoru příkazů jsou očekávány pouze deklarace a definice proměnných, a přiřazení do proměnných. Program také nepodporuje všechny operátory jazyka C, ale pouze ty, které jsou rozepsány v podkapitole 2.2.3. Jedná se o ternární operátor, bitové a logické operátory, operátory plus, mínus a unární operátory bez vedlejšího efektu. Navíc jsou podporovány pouze celočíselné proměnné.

Analyzované výrazy jsou **převáděny do CNF** podle návrhu z podkapitoly 3.1, která začíná na straně 11. Většina operátorů uvedených v podkapitole 3.1.5 na straně 25, tzv. logické operátory negace, součtu a součinu, přiřazení, přetypování a unární operátory plus a mínus, a ternární operátor z podkapitoly 3.1.2 na straně 16, jsou převedeny odlišným způsobem od ostatních operátorů. Tyto operátory jsou nějakým způsobem upraveny nebo nahrazeny jiným operátorem, nebo více operátory na úrovni analyzátoru jazyka C. Operátory unárního mínusu a logické operátory negace, součtu a součinu jsou převedeny podle popisu z podkapitoly 3.1.5 na odpovídající jiný operátor, nebo kombinaci více operátorů v rámci analyzátoru jazyka C a v převodu do CNF formulí už nevystupují. Ternární operátor je upraven a vyhodnocení podmínky prvního operandu je předpřipraveno z analyzátoru. Takto předpřipravený operátor je dále převeden s ostatními operátory do CNF.

Operátor přetypování je jediným operátorem z podkapitoly 3.1.5, který se neupravuje v analyzátoru. Tento operátor je přímo převeden a v podstatě jen přiřazuje bity z jedné proměnné do jiné proměnné. Navíc, ale musí zohledňovat, jestli je výsledná proměnná znaménkového typu nebo ne, protože pokud je datový typ výsledné proměnné větší a je znaménkový, musí se do nových bitů nastavit nejdůležitější bit původní proměnné. V opačném případě jsou nové bity nastaveny na nulu. Operátory přiřazení a unární plus jsou převedeny v rámci analyzátoru na operátor přetypování, protože ve výsledku je jejich funkcionalita stejná. Operátor přetypování společně s operátory porovnávajících velikost čísel jsou jediné operátory, které zohledňují, jestli pracují se znaménkovým nebo bezznaménkovým datovým typem.

Jedinou výjimkou při převodu operátorů podle podkapitoly 3.1 jsou operátory bitového posunu, které se v implementaci liší od jejich původního návrhu. Hlavní příčinou rozdílnosti je jejich nedefinované chování pro některé hodnoty jejich operandů. Kdy je výsledek operátoru nedefinován je popsáno v kapitole 2.2.3 i u návrhu jejich převodu v podkapitole 3.1.6. Všechny případy nedefinovaného chování byly otestovány na všech zmíněných systémech i překladačích, uvedených na začátku kapitoly 4. V případě záporného čísla v levém operandu všechny kompilátory pracovaly s číslem, jako by bylo bezznaménkové a takto to je implementováno také v programu. Stejný přístup byl otestován také u pravého operandu. U pravého operandu je navíc omezení, že pokud je hodnota větší než velikost levého operandu v bitech, tak je výsledek také nedefinován. Zde se již implementace jednotlivých překladačů mírně lišily. Překladače clang a gcc upravují pravý operand modulem velikostí levého operandu v bitech. Oproti tomu Microsoft Visual Studio upravuje pravý operand modulo číslem 256, nehledě na velikost levého operandu. V tomto případě se program řídí podle implementaci překladačů clang a gcc.

Převaděč do DIMACS formátu vypíše před převodem komentáře, které obsahují důležité informace ohledně výstupu. Obsahuje informace o jednotlivých mapováních proměnných na příslušné bity, a popis, jakým způsobem jsou bity na proměnné mapovány. Tato funkcionalita odpovídá návrhu z podkapitoly 3.2. Výstup obsahuje pouze vstupní a výstupní proměnné. Po vypsání komentářů jsou vypsány samotné logické formule v DIMACS formátu, který je popsán v kapitole 2.4. Navíc za každou vypsanou formuli je vypsána volitelná nula, protože některé programy řešící SAT problémy ji očekávají[1].

4.2.1 Vnitřní reprezentace proměnných

Při zpracování v analyzátoru jsou konstanty i proměnné převedeny na vnitřní reprezentaci proměnných. Tato reprezentace proměnných je také použita pro reprezentaci dočasných proměnných a konstant potřebných při analýze výrazů a generování mezivýsledků. Důležitou vlastností vnitřní reprezentace proměnných je použití každé proměnné k přiřazení nanejvýše jednou. Toto omezení plyne z výstupu programu ve formě jedné logické formule. Výstupní logická formule musí mít vyhodnocení nezávislé na pořadí vyhodnocení jednotlivých výrazů uvnitř. Z tohoto důvodu i každé přiřazení do proměnné generuje novou proměnnou v rámci vnitřní reprezentace. Tento způsob vnitřní reprezentace proměnných se nazývá jedinou statickou formu přiřazení, *static single assignment form*, *SSA*[2].

Proměnné jsou děleny na vstupní, výstupní a dočasné v rámci převodu do CNF. Vstupní proměnná reprezentuje neznámé proměnné. Vstupní proměnné jsou ty, které mají samostatnou deklaraci bez definice. Tato proměnná může nabývat libovolné hodnoty, kterou může daný datový typ reprezentovat. Výstupní proměnné jsou ty, které byly deklarovány nebo definovány a konkrétně z vnitřní reprezentace to je ta proměnná, do které bylo naposledy přiřazena nějaká hodnota. Při mapování jednotlivých bitů proměnných do DIMACS formátu jsou nejmenší čísla bitů rezervována pro vstupní a následně pro výstupní proměnné. Tato konvence má za cíl usnadnit následnou práci s výstupním souborem nebo výstupem ze SAT solveru.

4.2.2 Algoritmus seřaďovacího nádraží

Algoritmus seřaďovacího nádraží, *shunting-yard algorithm*, analyzuje výraz v infixové notaci a jeho výstupem je výraz v postfixové notaci [15][16]. V této podkapitole je popsán tento algoritmus a jeho modifikace, která byla použita při implementaci, pro analyzování výrazů.

---

[1] Konkrétně použitý `2clseq` SAT solver, http://www.cs.toronto.edu/~fbacchus/sat.html

[2] Více o SSA na wikipedii http://en.wikipedia.org/wiki/Static_single_assignment_form nebo v knize o SSA, http://ssabook.gforge.inria.fr/latest/book.pdf.

Algoritmus pracuje s jednou vstupní frontou, jednou výstupní frontou a zásobníkem. Při zpracování vstupní fronty jsou všechny konstanty a proměnné přesunuty rovnou do výstupní fronty. Pokud algoritmus narazí na operátor, tak ten pošle do zásobníku. Ale ještě před tím, než ho přidá na vrchol zásobníku, tak porovná prioritu a asociativitu aktuálního operátoru a operátoru z vrcholu zásobníku. Pokud je aktuální operátor asociativní zprava a má menší prioritu než operátor z vrcholu zásobníku, nebo pokud je operátor levě asociativní a má menší nebo stejnou prioritu, tak je operátor z vrcholu zásobníku přesunut do výstupní fronty. Pokud je na vrcholu zásobníku otevírací závorka, tak se operátor umístí na vrchol zásobníku. Toto porovnávání se děje tak dlouho, dokud podmínka platí anebo dokud je na vrcholu zásobníku nějaký operátor. Poté se aktuálně zpracovávaný operátor umístí na vrchol zásobníku [15][16]. Při nálezu otevírací závorky je tato závorka umístěna na vrchol zásobníku. Při nalezení uzavírací závorky jsou operátory přesunovány ze zásobníku do výstupní fronty, dokud se nenarazí na otevírací závorku. Až se narazí na otevírací závorku, tak jsou obě závorky odstraněny. Při nalezení konce výrazu jsou postupně odebrány všechny operátory ze zásobníku a přesunuty do výstupní fronty [15][16]. Takto popsaný algoritmus lze poupravit a použít pro analyzování výrazu v infixovém formátu do vnitřní reprezentace abstraktního syntaktického stromu uvnitř programu.

Pro potřeby použití tohoto algoritmu v programu je nutné drobné předzpracování. Problémem jsou binární a unární operátory plus a mínus. V době, kdy se porovnávají operátory, je třeba vědět jejich prioritu i asociativitu. Lexikální analyzátor ovšem nemůže vědět, jestli se jedná o binární či unární operátor a předpokládá, že se jedná vždy o binární operátory plus a mínus. Při samotném porovnávání tato neznalost nepředstavuje největší problém, ten leží při následném vyjmutí tohoto operátoru z vrcholu zásobníku a přesunutí ho do výstupní fronty, kde již prakticky nelze určit, jestli je operátor binární nebo unární. Předzpracování leží v analyzátoru výrazů. Ještě před tím, než se začne operátor porovnávat s operátorem na vrcholu zásobníku, je zkontrolována možnost převedení binárního plus a mínus na unární, podle posledního zpracovaného lexému. Pokud byl posledním lexémem operátor, otevírací závorka anebo je to první zpracovávaný lexém ve výrazu, je operátor převeden na unární variantu. Takto upravený algoritmus si již dokáže poradit se všemi vstupy podporovanými v této práci, ale zatím je výstupem stále postfixová notace.

Cílem modifikace tohoto algoritmu bude získání vnitřní reprezentace zanalyzovaného výrazu. Modifikace spočívá ve změně výstupní fronty na výstupní zásobník a změnou chování při přesunu operátoru z vrcholu zásobníku do výstupní fronty. Když je přesouván operátor z vrcholu zásobníku do výstupní fronty, tak již nebude umístěn do výstupního zásobníku, ale do speciálního výstupu, seznamu příkazů. K tomuto operátoru jsou následně přidány operandy z vrcholu výstupního zásobníku podle arity operátoru. K nově přidanému příkazu je vygenerována nová proměnná reprezentující výsledek tohoto příkazu. Tento operátor, operandy a výsledek reprezentují jeden příkaz a tuto strukturu lze vidět ve struktuře programu, viz Obrázek 4.2 na straně 32. Proměnná reprezentující výsledek tohoto příkazu je umístěn na vrchol výstupního zásobníku. Po úspěšném ukončení analyzování zůstane ve výstupním zásobníku jediná proměnná reprezentující celkový výsledek analyzovaného výrazu.

4.3 Testování

Po celou dobu vývoje byl kladen důraz na testovatelnost aplikace. Z tohoto důvodu byly při vývoji psány a užívány unit testy a celý vývoj probíhal agilní metodikou, programování řízené testy, *test driven development*. Pro účely testování byl použit testovací framework, *Google Test*[1]. Tento framework funguje a je v programu používán pod oběma používanými operačními systémy. Použití testovacího frameworku, programování řízené testy a pokrytí celé aplikace testy umožňovalo, kdykoliv puštění všech testů a jednoduchou a rychlou verifikaci stavu aplikace a ověření její funkcionality. Celkově bylo napsáno 174 testů, které pokrývají 85 % napsaného kódu aplikace. Pokrytí kódu testy bylo změřeno nástrojem *OpenCppCoverage*[2], který je napsán pouze pro operační systém Microsoft Windows.

Většina vývoje aplikace probíhala návrhem části architektury a implementací odpovídající kostry této infrastruktury. Nad touto kostrou bylo následně napsáno několik unit testů pro testování funkčnosti, kterou by tato část programu měla umět. Jakmile byla dokončena funkcionalita implementované části a testy potvrdily funkčnost, byly dopsány další testy testující funkcionalitu více do detailů. Jakmile testy opět potvrdily funkčnost nové části aplikace, byla navržena další část architektury a tento model vývoje pokračoval dále.

Jednotlivé části programu byly od počátku testovány izolovaně, ale jakmile obsahovali nějakou základní funkcionalitu, byly testovány také v integraci s ostatními částmi. Pokud při vývoji byla nalezena jakákoliv chyba, byl napsán alespoň jeden test, který uměl odhalit tuto chybu. Poté, co byla chyba odstraněna, tak její opravu kontroloval napsaný test.

Vzhledem ke složitosti finálního výstupu aplikace již nestačily k otestování této části ani samotné unit testy. Tyto testy sice mohou kontrolovat, že se generuje výstup a výstupní logická formule obsahuje očekávané části, ale již lehce nezkontrolují opravdovou korektnost výstupní formule. Program může generovat špatný výstup z důvodu špatné implementace, ale také může být chyba už v samotném návrhu převodu nějakého operátoru. Z těchto důvodů byly do projektu a tedy přímo do testů integrovány dva SAT solvery, které slouží k verifikaci výstupů programu. Používají se dva SAT solvery z důvodu vzájemné kontroly jejich výstupů a zabránění případné další možné chyby.

Integrační testy používající SAT solvery obsahují krátký program obsahující co nejjednodušší výraz, nebo více výrazů, pro ověření funkčnosti. Podrobně testují základní funkčnost, jako například konstanty a operátor přiřazení, a další testy se potom již mohou spolehnout na tuto funkcionalitu a dále ji použít. Tyto integrační testy po převedení výrazů do CNF jsou také schopny vložit vlastní logickou formuli před převodem do formátu DIMACS pro účel testu, a tak změnit význam celého výrazu. Tento postup je vysvětlen na příkladu níže.

[1] Domovská stránku Google Testu https://code.google.com/p/googletest/.
[2] Domovská stránka OpenCppCoverage nástroje https://opencppcoverage.codeplex.com/.

Mějme výrazy:

```
int a;
int b;
int c = a + b;
int d = (a == 45) && (b == -17);
```

Takto zadaný výraz počítá součet vstupních proměnných `a` a `b` do výstupní proměnné `c` a výstupní proměnou `d` nastavuje na jedničku v případě splněných příslušných podmínek. V integračních testech je tento výraz převeden do CNF. Následně se najde, které bity odpovídají proměnné `d` a do logických formulí je vložena formule dávající nejméně významnému bitu hodnotu jedna. Takto upravená CNF formule se přeloží do DIMACS formátu a je dána SAT solverům k řešení. Jelikož je formule upravená a proměnná `d` se musí rovnat hodnotě logické pravdy, tak musí SAT solvery dosadit za proměnné `a` i `b` odpovídající hodnoty, aby vyhověly podmínce, a hodnota proměnné `c` je testovatelná na konkrétní hodnotu, 28 v tomto případě. Aby tento příklad správně fungoval, tak je potřeba, aby fungovala reprezentace konstant a operátory přiřazení, binární plus, logické rovnosti, logického součinu a unárního mínusu. Hodnoty vstupních proměnných `a` a `b` jsou nastavovány takovýmto způsobem, aby se do budoucna zamezilo rozbití funkčnosti testů zavedením optimalizace analyzátoru jazyka C, která bude vypočítávat operátory, které budou mít na vstupu konstanty. Ve zmíněném případě by při rozbití funkčnosti převodu do CNF příslušné testy stále ukazovaly, že fungují, protože se operátor ani nedostal k převodu do CNF a optimalizace ho odstranila ještě před samotným převodem.

4.4 Ukázka

Pro účely demonstrační ukázky programu byl zvolen algoritmus cyklický redundantní součet, *cyclic redundancy check*, CRC. Tento algoritmus převzatý z internetu[1] obsahuje pouze operace podporované v této práci nebo operace, které lze převést na ty podporované. Tento převzatý algoritmus funguje pouze na strojích, které ukládají nejméně důležité bajty, před ty důležitější, tzv. jsou *little-endian*. Funkčnost algoritmu byla ověřena pomocí online CRC kalkulačky[2]. Byla zvolena šestnáctibitová varianta s polynomem `0x8005`, respektive jeho reverzi `0xA001`, a vstupní zpráva „`123456789`“. Algoritmus přepsaný do jazyka C vypadá následovně:

```
unsigned short crc16(char * message, unsigned length)
{
  unsigned short remainder = 0;
  for (unsigned i = 0; i < length; i++, message++) {
        remainder = remainder ^ *message;
        for (unsigned j = 0; j < 8; j++) {
              if (remainder & 0x0001) {
                    remainder = (remainder >> 1) ^ 0xA001;
              } else {
                    remainder = remainder >> 1;
              }
        }
  }
  return remainder;
}
```

Podmínka uvnitř algoritmu lze převést na ternární operátor. Vnitřní cyklus lze rozepsat a tím je odstraněn. V rámci práce, ale nejsou podporovány operátory pro přístup k datům a algoritmus tedy bude muset být upraven, aby pracoval nad konstantně dlouhou zprávou. Tímto se odstraní i vnější cyklus. Příloha B obsahuje takto převedený algoritmus a nachází se na straně 47. Takto upravený algoritmus je již plně podporován v rámci práce.

Před převodem do DIMACS formátu tohoto algoritmu je možné si pomoci s vyhodnocením vložením podmínky na konec. Přidáním podmínky:

```
int check = remainder == 0xbb3d;
```

je nápomocné pro následné vyhodnocení. Tato podmínka kontroluje jen jednu hodnotu, a tedy tento konkrétní výsledek je možné lehce zkontrolovat přímo z výstupu SAT solveru, podmínka ale může být později mnohem složitější a pak je již vhodné ji využívat.

---

[1] Algoritmus CRC: http://en.wikipedia.org/wiki/Computation_of_cyclic_redundancy_checks, Code fragment 5.
[2] Online CRC kalkulačka http://www.zorc.breitbandkatze.de/crc.html

Upravený algoritmus včetně podmínky na konci se předá programu vytvořenému v rámci této práce, aby ho přeložil do formátu DIMACS. Tento výstup se následně předá SAT solveru a ten vypočítá konkrétní řešení. Výstup programu vyvinutého v rámci této práce není přiložen ani v přílohách, neboť obsahuje před dvě stě osmdesát tisíc řádků. Z výstupu SAT solveru lze vyčíst, že hodnota proměnné `check` je rovna jedné a tedy algoritmus vypočítal očekávanou hodnotu.

S tímto algoritmem je ale možné pracovat dále. Například je možné odmazávat známou zprávu a nechat ho dopočítat chybějící znaky ze zbylých znaků a výsledného CRC kódu. Odmazání známého znaku se provede odebráním definice příslušné proměnné a zanechání její deklarace. Jako další pomocné omezení se může pomoci algoritmu tím, že jako vstupní znaky se očekávají pouze číslice. Upravená podmínka bude mít tedy tvar:

```
int check = (remainder == 0xbb3d) && (messageChar1 >= 48 && messageChar1 <= 57)
&& (messageChar2 >= 48 && messageChar2 <= 57) && (messageChar3 >= 48 &&
messageChar3 <= 57) && (messageChar4 >= 48 && messageChar4 <= 57) &&
(messageChar5 >= 48 && messageChar5 <= 57) && (messageChar6 >= 48 &&
messageChar6 <= 57) && (messageChar7 >= 48 && messageChar7 <= 57) &&
(messageChar8 >= 48 && messageChar8 <= 57) && (messageChar9 >= 48 &&
messageChar9 <= 57)
```

Po umazání posledního znaku se takto upravený algoritmus nechá přeložit do DIMACS. Výstupní DIMACS formát se musí upravit a to tak, že se nastaví nejméně důležitému bitu proměnné `check` na hodnotu jedna a předá se to SAT solveru. Z výstupu SAT solveru lze vyčíst, že odmazanému znaku SAT solver vypočítal očekávanou hodnotu. Postupným dalším zkoušením a odmazáváním dalších známých znaků zprávy se lze dostat až ke zprávě „`123`“, kde již SAT solver vypočítá původní zprávu jako „`123116890`“.

Pro zprávu „1234“ sice vypočítal SAT solver správnou původní zprávu, je ale možné, že to není jediné možné řešení a nelze se tedy v tomto případě spolehnout na doplnění správné zprávy. Pro další verifikaci je možné rozšířit nebo vytvořit novou podmínku.

```
int check2 = (messageChar4 != '4') || (messageChar5 != '5') || (messageChar6 !=
'6') || (messageChar7 != '7') || (messageChar8 != '8') || (messageChar9 != '9');
```

Této podmínce se opět nastaví před předáním do SAT solveru, že musí platit. Pokud by očekávané řešení bylo jediným řešením takto zadaného problému, měl by SAT solver vrátit, že tento SAT problém není možné vyřešit. Po spuštění ale SAT solver vrací „`123496289`“ a nelze se tedy spolehnout na to, že by algoritmus byl schopen doplnil pět číslic z devíti, protože existují i jiná řešení. Přidáním jedné další číslice ke známé zprávě už ale SAT solver není schopen najít jiné než očekávané řešení a s takovýmito omezeními lze od algoritmu očekávat, že doplní čtyři chybějící číslice z devíti.

Dále je možné například zjemnit původní podmínku, která omezuje vstupní znaky zprávy pouze na číslice, aby povolovala i písmena a další znaky. Podmínka se změní, aby znaky zprávy byly v rozmezí od mezery, ASCII hodnota 32, až po vlnovku, ASCII hodnota 126. Při takto změněné podmínce algoritmus uvažuje o obsahu zprávy jako o libovolné zprávě a s tímto nastavením a podobném zkoušení, jako bylo popsáno dříve v této kapitole, se dojde k výsledku, že je možno spolehlivě doplnit poslední dvě písmena zprávy.

4.5 Vyhodnocení

Záměr práce byl splněn. Práci obsahuje podrobný popis DIMACS formátu a operátorů programovacího jazyka C, které mají být podporovány. Dále byl navržen převod operátorů do CNF a také byl navržen program pro realizaci tohoto převodu. Navrhnutý program byl realizován a je testován automatickými testy obsahujícími testovací sadu. Následně jsou demonstrovány možné užití programu na příkladu.

Tabulka 4.1 zobrazuje časovou náročnost programu vytvořeného v rámci této práce pro překlad výrazu do DIMACS formátu a následnou velikost a složitost výsledku. Testování proběhlo oproti binárnímu sčítání, bitovému operátoru posunu vlevo, upravenému algoritmu CRC-16, viz podkapitola 4.4 a Příloha B, a oproti souboru s náhodně poskládanými mnoha operátory a výrazy.

Tabulka 4.1 – Tabulka času potřebného pro převod do DIMACS a velikost zadaného a výsledného souboru a výrazu.

	Překlad [s]	Zadaný soubor	Výsledný soubor	Proměnných	Klauzulí
Plus	0,003	34 B	6,8 kB	127	435
Bitový posun vlevo	0,005	35 B	38,6 kB	96	1 552
CRC-16	0,375	6,2 kB	10,6 MB	37 064	289 696
Náhodné, složité	0,601	2,6 kB	20,7 MB	40 916	504 260

Další tabulka níže, viz Tabulka 4.2 na straně 42, zobrazuje následnou časovou náročnost vyřešení výsledného DIMACS formátu. Testování proběhlo oproti stejné sadě testů. SAT solver 2clseq nebyl schopen najít řešení posledního problému v rozumném čase (5min), a tak mu výsledek v tabulce chybí. SAT solver minisat neprojevoval nikde větší problém co se paměťové nebo časové náročnosti týče. 2clseq si při řešení posledních dvou SAT problémů alokoval 1,9GB, respektive 1,7GB paměti.

Tabulka 4.2 – Tabulka časové náročnosti SAT solveru pro vyřešení přeloženého problému v DIMACS.

	2clseq [s]	minisat [s]
Plus	0,009	0,003
Bitový posun vlevo	0,011	0,007
CRC-16	1,640	0,460
Náhodné, složité		1,096

Všechny časy byly měřeny unixovým příkazem *time*. Byla provedena desítka spuštění všech testů a ze středních šesti výsledků byl vypočítán aritmetický průměr. Vzhledem k velikosti zadaných výrazů pracuje realizovaná aplikace pro převod do DIMACS v rozumném čase. Výsledná formule je velmi velká, ale i tu je použitý minisat SAT solver v rozumném čase schopen vyřešit. SAT solver 2clseq se ukázal jako nevhodný pro použití na největších testovaných problémech.

5 Závěr

Cílem této práce je navrhnutí převodu výrazů zapsaných v programovacím jazyk C do formátu DIMACS CNF a vytvoření programu v programovacím jazyce C++ pro realizaci tohoto převodu. V rámci této práce byl navržen převod jednotlivých operátorů programovacího jazyka C do konjunktivního normálního formátu. Byl navržen a realizován program, který umí provést tento převod a výrazy programovacího jazyka C převede do CNF a následně do formátu DIMACS CNF. Cíl této práce byl splněn.

Nastudoval jsem formát DIMACS, který slouží pro zápis formulí v konjunktivní normální formě a nastudoval jsem podrobně operace programovacího jazyku C. Popis těchto témat se nachází v kapitole 2. Navrhl jsem převod výrazů v programovacím jazyku C do CNF a navrhl jsem program, který by prováděl tento převod, a výraz v CNF by dále převedl do formátu DIMACS CNF. Tento návrh se nachází v kapitole 3. Tuto část práce jsem udělal v rámci semestrální práce. Podle návrhu jsem naprogramoval aplikaci v C++, která provádí právě tento převod a popis její implementace se nachází v kapitole 4. Vytvořil jsem testovací sadu aplikace a tato testovací sada je součástí automatických testů, které jsou popsány v podkapitole 4.3. Vytvořil jsem demonstraci použití nástroje nacházející se v podkapitole 4.4.

Výkon aplikace, co se převodu výrazu z C do DIMACS i následné časové složitosti pro SAT solver, je popsán v podkapitole 4.5 začínající na straně 41. Časové složitosti jsou uspokojivé vzhledem k zadaným výrazům. Celý program je pokryt testy pro kontrolu správné funkčnosti. Některé implementované unit testy kontrolující validitu programu využívají SAT solvery a kontrolují tak i samotný návrh převodů operátorů do CNF.

Program je možné rozšířit o podporu dalších zatím nepodporovaných operátorů. Jedná se o operátory sloučené s přiřazením (například „+=“) a sufixové i prefixové operátory inkrementace i dekrementace. Za zvážení by stálo, jestli je možné rozšířit program o operátory násobení, dělení a modulo. Program by také bylo možné rozšířit o podmíněnou konstrukci a cyklus s fixním počtem cyklů. Na zvážení, jestli je to možné, je cyklus s neznámým počtem cyklů. Jiným směrem rozšíření je optimalizace výstupu analyzátoru jazyka C i výstup převodu do CNF, aby výstupní formule byly co nejmenší a následný SAT solver měl co nejméně práce. Dalším možným rozšířením může být uživatelské rozhraní, které mimo jiné také může umožňovat modifikaci výstupní formule v CNF před převodem do DIMACS.

6 Literatura

[1] DENNIS M. RITCHIE. The Development of the C Language [on-line]. 2003 [cit. 2014-11-09]. Dostupné na: < http://cm.bell-labs.com/cm/cs/who/dmr/chist.html >.

[2] CUBBI, aj. History of C [on-line]. 2012-05-08, aktualizováno 2014-12-11 [cit. 2014-12-26]. Dostupné na: < http://en.cppreference.com/w/c/language/history >.

[3] GPIETSCH, aj. C programming language [on-line]. 2002-09-07, aktualizováno 2014-11-20 [cit. 2014-11-22]. Dostupné na: < http://en.wikipedia.org/wiki/C_(programming_language) >.

[4] BRIAN W. KERNIGHAN, DENNIS M. RITCHIE. The C Programming Language, Second Edition. Prentice Hall, Inc., 1988. ISBN 0-13-110370-9.

[5] KOLEKTIV AUTORŮ. Programming languages – C. *ISO/IEC 9899:TC3* [on-line]. 2007-09-07 [cit. 2014-11-09]. Dostupné na: < http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf >.

[6] CUBBI, aj. Expressions [on-line]. 2015-01-02, aktualizováno 2015-01-02 [cit. 2015-01-04]. Dostupné na: < http://en.cppreference.com/w/c/language/expressions >.

[7] FORDERUD, aj. Operators in C and C++ [on-line]. 2005-11-06, aktualizováno 2014-11-04 [cit. 2014-11-22]. Dostupné na: < http://en.wikipedia.org/wiki/Operators_in_C_and_C++ >.

[8] P12, aj. C Operator Precedence [on-line]. 2012-04-16, aktualizováno 2014-12-26 [cit. 2014-12-26]. Dostupné na: < http://en.cppreference.com/w/c/language/operator_precedence >.

[9] ZDENĚK RYJÁČEK. Teorie NP-úplnosti [on-line]. [cit. 2015-04-19]. Dostupné na: < http://www.cam.zcu.cz/~ryjacek/students/TGD1-folie/TGD1-10-folie.pdf >

[10] KOLEKTIV AUTORŮ. Satisfiability Suggested Format [on-line]. Aktualizováno 1993-05-08 [cit. 2014-11-07]. Dostupné na: < http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf >

[11] JOHN BURKARDT. CNF Files [on-line]. Aktualizováno 2008-05-22 [cit. 2014-11-07]. Dostupné na: < http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html >.

[12] GREYMATTER. CNF – Conjunctive Normal Form (Dimacs format) Explained [on-line]. 2011-07-29 [cit. 2014-11-07]. Dostupné na: < https://fairmut3x.wordpress.com/2011/07/29/cnf-conjunctive-normal-form-dimacs-format-explained/ >.

[13] ERIK GARRISON, aj. Conjunctive normal form [on-line]. 2005-12-16, aktualizováno 2014-09-17 [cit. 2014-11-07]. Dostupné na: < http://en.wikipedia.org/wiki/Conjunctive_normal_form >.

[14] MIROSLAV N. VELEV, RANDAL E. BRYANT. TLSim and EVC: A Term-Level Symbolic Simulator and an Efficient Decision Procedure for the Logic of Equality with Uninterpreted Functions and Memories. *International Journal of Embedded Systems* [on-line]. 2005, Vol. 1, No. 1/2, S. 134-149 [cit. 2014-11-07]. Dostupné na: < https://www.cs.cmu.edu/~bryant/pubdir/ijes05.pdf >.

[15] EDSGER W. DIJKSTRA. Algol-60 translation [on-line]. 1961, s. 27-32 [cit. 2015-14-02]. Dostupné na: < http://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF >.

[16] KEENAN PEPPER, aj. Shunting-yard algorithm [on-line]. 2005-09-17, aktualizováno 2015-12-02 [cit. 2015-14-02]. Dostupné na: < http://en.wikipedia.org/wiki/Shunting-yard_algorithm >.

Příloha A
Obsah CD

Na přiloženém CD lze nalézt následující adresáře a soubory:

- README – stručná dokumentace aplikace
- LICENSE – licence jednotlivých vzniklých a užitých částí
- Makefile – pro překlad pomocí GNU make
- ctodimacs.bundle – zabalený git repozitář
- bin/ – zkompilovaný program a SAT solvery pro Windows (včetně redistribučních balíčků Microsoft Visual Studia 2013)
- doc/ – všechny soubory obsažené ve zprávě a zpráva v původním formátu
- lib/ – externí užité části včetně jejich licence a adresy, odkud je lze získat
- msvc-proj/ – projektové soubory pro Microsoft Visual Studio
- src/*.[cpp|h] – zdrojové soubory aplikace
    - tests/*.[cpp|h] – zdrojové soubory aplikace spouštějící unit testy
        - 2clseq/ – zdrojové soubory 2clseq SAT sovleru
        - gtest/ – zdrojové soubory Google test frameworku
        - minisat/ – zdrojové soubory minisat SAT solveru
- tests/examples/ – testovací příklady včetně jejich překladu do DIMACS

Příloha B
Algoritmus CRC-16

Algoritmus CRC-16 pro polynom `0x8005`, respektive jeho reverzi `0xA001` a vstupní zprávu „`123456789`“. Algoritmus je upraven, aby byl použitelný pro převod v rámci této práce. Původní varianta algoritmu se nachází v kapitole 4.4.

```
char messageChar1 = '1';
char messageChar2 = '2';
char messageChar3 = '3';
char messageChar4 = '4';
char messageChar5 = '5';
char messageChar6 = '6';
char messageChar7 = '7';
char messageChar8 = '8';
char messageChar9 = '9';
unsigned short remainder = 0;

remainder = remainder ^ messageChar1;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);

remainder = remainder ^ messageChar2;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);

remainder = remainder ^ messageChar3;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
```

```
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);

remainder = remainder ^ messageChar4;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);

remainder = remainder ^ messageChar5;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);

remainder = remainder ^ messageChar6;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);

remainder = remainder ^ messageChar7;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
```

```
remainder = remainder ^ messageChar8;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);

remainder = remainder ^ messageChar9;
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
remainder = remainder & 0x0001 ? (remainder >> 1) ^ 0xA001 : (remainder >> 1);
```