FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Minimalizace normálních tvarů
Student:	Mgr. Klára Červenková
Vedoucí:	Mgr. Jan Starý, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Webové a softwarové inženýrství, zaměření Softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2021/2022

Pokyny pro vypracování

1. Definujte minimální normální tvar výrokové formule a popište alespoň jeden algoritmus, který tento tvar hledá.
2. Implementujte parser výrokových formulí. Za tím účelem navrhněte vhodný formální jazyk v rámci ASCII, aby bylo možné zachytit syntax výrokových formulí.
3. Nad tímto parserem implementujte hledání konjunktivního a disjunktivního normálního tvaru.
4. Nad normálními tvary pak implementujte popsaný minimalizační algoritmus.
5. Implementaci proveďte v jazyce C.
6. Dbejte na korektnost a přenostitelnost kódu, přinejmenším mezi POSIX systémy.
7. Vytvořený program zdokumentujte, nejlépe ve formě standardní manuálové stránky.
8. Vytvořený program řádně otestujte; metody testování zdokumentujte.

Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 30. října 2020 v Praze.

Bakalářská práce

MINIMALIZACE NORMÁLNÍCH TVARŮ

Klára Červenková

Fakulta informačních technologií ČVUT v Praze
Katedra softwarového inženýrství
Vedoucí: Mgr. Jan Starý, Ph.D.
2. února 2022

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Na tomto místě chci poděkovat Mgr. Janu Starému, Ph.D., za jeho cenné rady, připomínky a ochotu vyjít vstříc při řešení problémů při psaní této bakalářské práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 2. února 2022 .....

Abstrakt

Cílem této bakalářské práce je vytvořit program, který uživateli vrátí, na základě jím zadané formule výrokové logiky, minimální disjunktivní, resp. konjunktivní normální tvary této výrokové formule. Přípravné analytické práce, které zahrnují definování pojmu minimální normální tvar výrokové formule a popis algoritmu Quine–McCluskey, který tento tvar hledá, jsou následovány návrhem formálního jazyka v rámci ASCII, aby bylo možné zachytit syntaxi výrokových formulí, a návrhem programu jako takového. Implementace syntaktického analyzátoru formule výrokové logiky, hledání disjunktivního, resp. konjuktivního normálního tvaru včetně vlastní minimalizace je provedena v programovacím jazyce C.

Klíčová slova minimalizace výrokových formulí, disjunktivní normální tvar, konjunktivní normální tvar, algoritmus Quine–McCluskey, syntaktický analyzátor výrokových formulí

Abstract

The aim of this bachelor's thesis is to create a program that returns to the user, based on a given formula of propositional logic, the minimal disjunctive, resp. conjunctive normal forms of this propositional formula. Preparatory analytical work, which included defining the notion of the minimal normal form of a propositional formula and describing the Quine–McCluskey's algorithm searching for that form, is followed by the design of a formal ASCII language to capture the syntax of propositional formulas and the design of the program as such. The implementation of a parser of propositional formulas, a search for the disjunctive, resp. conjunctive normal form, including the minimization itself, is done using the programming language C.

Keywords minimization of propositional formulas, disjunctive normal form, conjunctive normal form, Quine–McCluskey's algorithm, parser of propositional formulas

Shrnutí

Motivace

V různých vědních disciplínách, technických oborech či obecně lidských činnostech zjišťujeme, že předložený problém lze převést do řeči matematické výrokové logiky a řešit jej jako optimalizační úlohu, která nás dovede k hledání minimálních normálních tvarů výrokové formule. V současné době není nijak obtížné nalézt různé webové aplikace, které nám takovou službu poskytnou, nicméně naše řešení bude cílit na využití prostřednictvím příkazové řádky, zajistí oproti jiným jednodušší zadávání výrokových formulí a nabídne využitím standardního vstupu a výstupu další možnosti, jak s programem pracovat.

Cíl práce

Naším cílem bylo vytvořit program, který uživateli vrátí, na základě jím zadané formule výrokové logiky, a to v námi navrženém formálním jazyce, minimální disjunktivní, resp. konjunktivní normální tvary této výrokové formule, přičemž implementace syntaktického analyzátoru formule výrokové logiky, hledání disjunktivního, resp. konjunktivního normálního tvaru včetně vlastní minimalizace byla provedena v programovacím jazyce C.

Postup

Po provedení přípravných analytických prací, které zahrnovaly definování pojmu minimální normální tvar výrokové formule a popis algoritmu, který tento tvar hledá, jsme vytvořili návrh formálního jazyka v rámci ASCII, aby bylo možné zachytit syntaxi výrokových formulí, a návrh programu jako takového. Po provedení implementace popsaného minimalizačního algoritmu do takto navrženého programu jsme program otestovali a nakonec jsme pro uživatele vytvořili příručku ve formě standardní manuálové stránky.

Výsledky práce

Výsledkem naší práce je program, který na základě vstupní výrokové formule vrací výstup požadovaný uživatelem. V případě korektního vstupu, což vyhodnocuje implementovaný syntaktický analyzátor, může program vracet zpět zadanou formuli, její pravdivostní tabulku, všechny minimální disjunktivní, resp. konjunktivní normální tvary či jeden z minimálních disjunktivních, resp. konjunktivních normálních tvarů. Pokyny pro práci s programem jsme shrnuli ve standardní manuálové stránce a program jako celek jsme úspěšně otestovali na vytvořené sadě testů.

Závěr

Vytvořený program je možné využít v praxi, např. pro optimalizaci logických obvodů, nebo pro výukové účely, a to jak pro vyučujícího ke kontrole správnosti výsledků testů, tak pro studenty pro nácvik požadovaných dovedností hledání minimálních disjunktivních, resp. konjunktivních normálních tvarů výrokových formulí.

Seznam zkratek

ASCII	American Standard Code for Information Interchange
DNT	Disjunktivní normální tvar
KNT	Konjunktivní normální tvar
POSIX	Portable Operating System Interface
UML	Unified Modeling Language

Kapitola 1
Úvod

Tento text dokumentuje bakalářskou práci, která vznikla v rámci studia na Fakultě informačních technologií Českého vysokého učení technického v Praze pod katedrou softwarového inženýrství. Zvolené téma *Minimalizace normálních tvarů* propojuje teorii s praxí a umožňuje převést tyto teoretické poznatky do praktického využití. Téma vychází z výrokové logiky a zároveň pro uvedení do praxe vyžaduje provedení implementace minimalizačního algoritmu.

Minimalizace normálních tvarů výrokových formulí bývá zařazována do vyučovacích osnov na vysokých školách. Vytvořený spustitelný program jako takový může tedy najít uplatnění v rámci výuky předmětů, které se touto problematikou zabývají, a může se jako doplněk učebních textů (např. [1], [2]) stát užitečnou pomůckou. V rámci práce proto zajistíme, aby se výstup z programu neomezil jen na výčet požadovaných minimálních tvarů, ale aby byly uživatelům, ať studentům, či vyučujícím, k dispozici i další mezivýstupy programu, kterými jsou úplné normální tvary výrokových formulí, vůči kterým si tak budou uživatelé moci zkontrolovat mezivýsledky své.

Minimalizaci formulí výrokové logiky pak prakticky využíváme zejména v optimalizačních úlohách, jako je např. optimalizace logických obvodů v elektrotechnice ([3]). Optimalizace zde umožňuje vytvořit ekvivalentně pracující elektronické součástky z méně komponent, tedy zabírající menší prostor, což přináší nejen materiálové, potažmo finanční úspory, ale také snižuje nežádoucí zpožďování elektrického signálu ([4]).

Rovněž budeme počítat s tím, aby bylo možné využít jednotlivé části zdrojového kódu pro potřebu tvorby jiných programů – i z tohoto důvodu budeme zdrojový kód prokládat komentáři, aby se v tomto kódu případný čtenář–uživatel lépe orientoval.

Neklademe si za cíl nalézt nový algoritmus, či zefektivňovat ty stávající. Jedná se nám čistě o implementaci jednoho konkrétního algoritmu, striktně podle jeho popisu. Poněkud předběhneme, když uvedeme, že se jedná o algoritmus Quine–McCluskey.

Vycházet budeme, jak dále uvidíme, z jeho popisu v několika různých zdrojích. Informace z těchto zdrojů sestavíme tak, abychom dosáhli našeho cíle, tedy najít (v určitém smyslu, což bude upřesněno v dalších kapitolách) všechny minimální disjunktivní normální tvary a všechny minimální konjunktivní normální tvary uživatelem zadané formule výrokové logiky.

V následujících kapitolách nejprve shrneme potřebné teoretické podklady od definic nejnutnějších pojmů matematické logiky přes základní vlastnosti takto definovaných entit, u kterých předpokládáme jejich využití, a algoritmu pro převod formálně zapsané formule výrokové logiky

do struktury lépe strojově zpracovatelné (syntaktický analyzátor), až po představení zmíněného minimalizačního algoritmu Quine–McCluskey. Následně navrhneme způsob, jakým bude program pracovat a komunikovat s uživatelem, a to včetně návrhu formálního jazyka, ve kterém bude programu předloženo zadání a ve kterém bude program vracet odpověď. Dále popíšeme vlastní realizaci–implementaci návrhu řešení a nakonec zhodnotíme správnou funkčnost vytvořeného programu na základě výsledků provedených testů.

Kapitola 2
Cíl práce

Cílem této bakalářské práce je vytvořit program, který vypíše, na základě uživatelem zadané formule výrokové logiky, všechny její minimální tvary.

Nejprve proto před zahájením vlastní implementační práce nadefinujeme pojmy minimální disjunktivní normální tvar výrokové formule a minimální konjunktivní normální tvar výrokové formule. Vzhledem k tomu, že definice těchto dvou stěžejních termínů spočívají na dalších termínech, zejména z oblasti výrokové logiky, a v různých zdrojích se tyto mohou odlišovat, uvedeme proto i definice několika dalších termínů a současně i základní matematické věty z této oblasti, ze kterých bude naše řešení vycházet. Následně popíšeme minimalizační algoritmus, který jsme pro naši další práci zvolili.

Protože formule výrokové logiky, určená uživatelem programu k minimalizaci, bude jako vstup do programu zadávána uživatelem v podobě textového řetězce, bude součástí naší práce také návrh vhodného formálního jazyka, abychom mohli zachytit syntaxi formulí výrokové logiky. Formální jazyk navrhneme tak, aby využíval pouze symboly v rámci ASCII. Samotné převedení formule výrokové logiky z textové podoby v navrženém jazyce do strojově využitelné struktury pak v rámci programování zajistíme implementací některého z typů syntaktických analyzátorů (= parser (z angličtiny)) formule výrokové logiky, který bude odpovídat námi navrženému formálnímu jazyku.

Jako samostatně využitelný mezivýstup našeho programu implementujeme výpočet pravdivostní tabulky dané výrokové formule, tedy vyjádření úplného disjunktivního, resp. konjunktivního normálního tvaru. Nakonec pak nad těmito úplnými normálními tvary implementujeme zvolený minimalizační algoritmus, jehož výstupem budou minimální disjunktivní, resp. konjunktivní normální tvary dané výrokové formule.

Implementaci celého programu provedeme v programovacím jazyce C s důrazem na korektnost a přenositelnost kódu, přinejmenším mezi POSIX systémy. Námi vytvořený program zdokumentujeme ve formě standardní manuálové stránky a rovněž finální program na závěr otestujeme, přičemž testování opět zdokumentujeme.

Kapitola 3
Analýza

V této kapitole shrneme teoretický základ jako výchozí bod pro implementaci jednotlivých částí programu, ze kterých bude následně sestaven. Bude se jednat zejména o definice základních pojmů z výrokové logiky a některé vybrané věty o vlastnostech takto definovaných entit, které doloží smysluplnost našeho postupu.

Protože pro načtení formule výrokové logiky z textového záznamu do strojově přijatelnější podoby budeme potřebovat některý ze syntaktických analyzátorů, popíšeme v této kapitole jeden z algoritmů pro syntaktickou analýzu, který jsme pro naši implementaci vybrali, jehož autorem je E. W. Dijkstra (1961, [5]) a je znám pod označením „seřaďovací nádraží“ (v angl. originálu „shunting–yard“).

Součástí této kapitoly bude také popis algoritmů pro hledání normálních tvarů formulí výrokové logiky a jejich minimalizace. Při hledání normálních tvarů budeme vycházet z tzv. pravdivostních tabulek, jak bude dále popsáno, a pro minimalizaci využijeme algoritmu známého pod označením Quine–McCluskey, což je vlastně algoritmus W. V. Quina (1952, [6]) vylepšený o několik málo let později E. J. McCluskeym (1956, [7]).

Mezi jiné algoritmy, které řeší ekvivalentní úlohu minimalizace, patří např. algoritmy publikované H. Aikenem (1951, [8]) nebo M. Karnaughem (1953, [9]), nicméně, jak uvádí E. J. McCluskey v předmluvě ke svému článku [7] v návaznosti na výčet algoritmů publikovaných právě Aikenem, Karnaughem či Quinem – jím popsaný algoritmus „*může být použit na složitější funkce, než předchozí metody, je systematický, a může být snadno naprogramovaný na digitáním počítači*“ ([7], překlad autor).

3.1 Základní pojmy

Definujme nejprve několik málo nejdůležitějších pojmů z výrokové logiky, o které se budou opírat jednotlivé algoritmy využité následně jako podklady pro implementaci.

▸ **Definice 3.1.** Prvotní výrok *je jednoduchá oznamovací věta, u níž má smysl se ptát, zda je či není pravdivá. Prvotní výroky označujeme symbolicky velkými tiskacími písmeny, A, B, ..., kterým říkáme* prvotní formule. ([2])

Tabulka 3.1 Pravdivostní tabulka

A	B	$\neg A$	$A \wedge B$	$A \vee B$	$A \Rightarrow B$	$A \Leftrightarrow B$
1	1	0	1	1	1	1
1	0	0	0	1	0	0
0	1	1	0	1	1	0
0	0	1	0	0	1	1

▸ **Definice 3.2.** Pravdivostní ohodnocení *v prvotních výroků je funkce z množiny prvotních formulí do množiny* $\{0, 1\}$. *Pokud pro prvotní výrok* A *platí* $v(A) = 1$, *řekneme, že výrok* A *je* pravdivý při ohodnocení v. *Pokud platí* $v(A) = 0$, *řekneme, že je* nepravdivý při ohodnocení v.

▸ **Definice 3.3.** Jazyk výrokové logiky *obsahuje*

- *symboly pro prvotní formule:* $A, B, \ldots,$
- *symboly pro logické spojky* $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$
- *závorky* (,), ... ([2])

Pro úplnost dodejme, že pod symboly pro logické spojky $\neg$, $\wedge$, $\vee$, $\Rightarrow$ a $\Leftrightarrow$, uvedenými v definici 3.3, rozumíme logické spojky pro (po řadě) unární logickou operaci negace a binární logické operace konjunkce, disjunkce, implikace a ekvivalence.

▸ **Definice 3.4.** Formule výrokové logiky *je posloupnost symbolů z jazyka výrokové logiky, která vznikne aplikací následujících pravidel:*

1. *Prvotní formule je formule.*
2. *Jsou-li* A *a* B *formule, pak i* $\neg A$, $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, $(A \Leftrightarrow B)$ *jsou formule.*
3. *Každá formule vznikne pomocí pravidel 1. a 2. v konečně mnoha krocích.* ([2])

Notace, při které se formule zapisují ve shodě s definicí 3.4 je tzv. *infixní notace* – název pochází z označení pozice binárních operátorů, které se nachází mezi svými dvěma operandy. Navíc bychom měli dle definice 3.4 dodržovat striktní uzávorkování každé jednotlivé (binární) operace. Infixní notace není ale jedinou možností, jak formule zapisovat, mezi další patří *prefixní* či *postfixní* notace. Prefixní notace je charakteristická tím, že každý operátor, ať unární či binární, předchází své operandy, pro případ postfixní notace pak opačně ([1]).

Mezi další možnosti zápisu či znázornění formule patří grafické znázornění pomocí tzv. *stromů*, v [2] označovaných jako *formační*, které ve svých vnitřních uzlech mají operátory a ve vnějších uzlech mají prvotní formule. Výstavba takovéhoto formačního stromu odpovídá sestavování formule dle definice 3.4 ([2]).

▸ **Definice 3.5.** *Nechť* A *je formule výrokové logiky a* v *je pravdivostní ohodnocení prvotních formulí, které se v* A *vyskytují.* Pravdivostní hodnotu $v(A)$ *formule* A *stanovíme induktivně postupným určováním pravdivosti podformulí od prvotních až po finální formuli. Využíváme pravdivostní tabulku pro logické spojky 3.1. Říkáme, že formule* A *je* pravdivá při ohodnocení v, *právě když* $v(A) = 1$, *a formule* A *je* nepravdivá při ohodnocení v, *právě když* $v(A) = 0$. ([2])

▸ Poznámka 3.6. Poznamenejme, že počet všech možných různých ohodnocení v formule A o n prvotních formulích je 2^n ([2]).

▸ **Definice 3.7.** *Formule A a B jsou* logicky ekvivalentní*, právě když pro každé ohodnocení v je $v(A) = v(B)$. Píšeme $A \models\!\models B$.* ([2])

Při úpravách formulí výrokové logiky často využíváme specifických vlastností logických spojek, z nichž některé připomeneme, protože je budeme v rámci minimalizačního algoritmu využívat. Větu 3.8 o těchto vlastnostech dokazovat nebudeme, s důkazem odkazujeme na [2].

▸ **Věta 3.8.** (Algebraické vlastnosti spojek). *Následující dvojice formulí jsou logicky ekvivalentní:*

1. $A \wedge B \models\!\models B \wedge A$ – *Komutativní zákon pro konjunkci.*
2. $A \vee B \models\!\models B \vee A$ – *Komutativní zákon pro disjunkci.*
3. $(A \wedge B) \wedge C \models\!\models A \wedge (B \wedge C)$ – *Asociativní zákon pro konjunkci.*
4. $(A \vee B) \vee C \models\!\models A \vee (B \vee C)$ – *Asociativní zákon pro disjunkci.*
5. $(A \wedge B) \vee C \models\!\models (A \vee C) \wedge (B \vee C)$ – *Distributivní zákony.*
6. $(A \vee B) \wedge C \models\!\models (A \wedge C) \vee (B \wedge C)$
7. $A \wedge (A \vee B) \models\!\models A$ – *Zákony absorpce.*
8. $A \vee (A \wedge B) \models\!\models A$ ([2])

Výrokové formule mohou mít při různých pravdivostních ohodnoceních obecně různou pravdivostní hodnotu a z tohoto pohledu mezi různými výrokovými formulemi existují dva speciální typy, pro které má výrokové formule pravdivostní hodnotu stále stejnou bez ohledu na její pravdivostní ohodnocení, a to následovně:

▸ **Definice 3.9.**

- *Formule je* tautologie*, právě když je pravdivá pro každé ohodnocení;*
- *formule je* kontradikce*, právě když je nepravdivá pro každé ohodnocení.* ([2])

▸ **Definice 3.10.** *Do jazyka výrokové logiky přidáme dva nové znaky:*

- *Symbolem $\top$ budeme označovat tautologii;*
- *symbolem $\bot$ budeme označovat kontradikci.* ([2])

Nyní definujeme ještě některé z dalších pojmů, o které se opírají definice minimálních disjunktivních normálních tvarů a minimálních konjunktivních normálních tvarů. Na ně navážeme definicí právě zmíněných minimálních tvarů, přičemž pro minimální disjunktivní normální tvar vyjdeme z [1] a analogicky (aniž by byla definice v [1] uvedena) definujeme i pojem minimální konjunktivní normální tvar.

▸ **Definice 3.11.** *Výroková formule je*

- literál*, pokud je prvotní formulí nebo negací prvotní formule;*
- implikant*, pokud je konjunkcí literálů;*
- klauzule*, pokud je disjunkcí literálů;*
- *v* disjunktivním normálním tvaru *(DNT), pokud je disjunkcí implikantů;*
- *v* konjuktivním normálním tvaru *(KNT), pokud je konjunkcí klauzulí;*
- *v* úplném disjunktivním normálním tvaru*, pokud se ve všech jeho implikantech vyskytují všechny použité výrokové proměnné. Takové implikanty jsou potom jeho* mintermy*;*
- *v* úplném konjunktivním normálním tvaru*, pokud se ve všech jeho klauzulích vyskytují všechny použité výrokové proměnné. Takové klauzule jsou potom jeho* maxtermy*;*
- *v* minimálním disjunktivním normálním tvaru*, pokud žádný ekvivalentní disjunktivní normální tvar nemá méně implikantů nebo méně literálů;*
- *v* minimálním kojunktivním normálním tvaru*, pokud žádný ekvivalentní konjunktivní normální tvar nemá méně klauzulí nebo méně literálů.* ([1])

▸ Poznámka 3.12. Pokud budeme v následujícím textu hovořit o normálních tvarech formulí, budeme mít na mysli formule v DNT či formule v KNT bez dalšího rozlišení. Analogicky pro úplný DNT, resp. úplný KNT, a minimální DNT, resp. minimální KNT.

Protože máme za jeden z dílčích cílů najít blíže nespecifikované normální tvary, můžeme si stanovit, že budeme hledat jejich speciální tvar, a to úplný normální tvar. Dále pak máme najít minimální normální tvar formulí, a je proto třeba vědět, zda má toto hledání smysl pro jakoukoli formuli, případně jaká jsou omezení. Uvedeme proto navíc následující věty, které nám zaručí, že tyto tvary (úplný a minimální) nalezneme pro libovolnou formuli.

▸ Poznámka 3.13. Před následující větou ještě zmíníme, že s ohledem na definice úplných normálních tvarů nelze pro kontradikci sestrojit úplný DNT, resp. pro tautologii nelze sestrojit úplný KNT a tyto případy vyřešíme shodně s [2], a to dodefinováním:

- Za úplný DNT kontradikce budeme považovat symbol $\bot$ a množinu implikantů budeme považovat za prázdnou.
- Za úplný KNT tautologie budeme považovat symbol $\top$ a množinu klausulí budeme považovat za prázdnou.

▸ **Věta 3.14.** *Ke každé formuli existuje logicky ekvivalentní formule, která je v úplném DNT, a logicky ekvivalentní formule, která je v úplném KNT.* ([2])

▸ **Věta 3.15.** *Ke každé formuli existuje logicky ekvivalentní formule, která je v minimálním DNT, a logicky ekvivalentní formule, která je v minimálním KNT.*

Důkaz věty 3.14 uvádět nebudeme, s důkazem odkazujeme na [2]. Důkaz věty 3.15 rovněž uvádět nebudeme, jednalo by se o důkaz triviální.

3.2 Syntaktický analyzátor

Pro syntaktickou analýzu vstupního uživatelského řetězce pro převod textově zapsané výrokové formule do stromové struktury, se kterou bude program dále pracovat, využijeme syntaktického analyzátoru známého jako Dijkstrův algoritmus seřaďovacího nádraží. Tento algoritmus zcela vyhovuje našim potřebám: Jedná se o algoritmus, který převádí aritmetický výraz (obsahující operandy, operátory s definovanou předností a závorky) zapsaný v infixovém tvaru do postfixového tvaru, resp. do podoby formačního stromu ([10]).

Pojmenování algoritmu odráží způsob práce s operátory, které jsou (analogicky ke způsobu sestavování vlakových souprav na seřaďovacích vlakových nádražích) přemísťované do pomocného zásobníku (na vedlejší kolej), kde jsou umístěny do doby, než jsou zpracované operandy/operátory (připojeny vagóny), které mají být zpracovány (připojeny do soupravy) před nimi ([10]).

Algoritmus popsaný v [5], vyžaduje kromě jasně identifikovatelných operandů také předem vyjmenované operátory s definovanou předností, čímž budeme mít zaručeno dodržování přednosti jedné operace před druhou, aniž by musela být naznačena uzávorkováním. Před vlastní syntaktickou analýzou tak bude nutné v rámci návrhu syntaxe rozhodnout, jak budou pravidla pro upřednostňování operací nastavena.

Syntaktický analyzátor podle [5] pracuje tak, že načítá po řadě zleva doprava jednotlivé operandy, resp. operátory. Jedná-li se o operand, tedy v našem případě primární formuli, zašle jej rovnou do výstupního zásobníku. Načte-li operátor, pak jej zašle do pomocného zásobníku, předtím ale z tohoto pomocného zásobníku vyjme všechny operátory, které mají prioritu větší nebo rovnu prioritě tohoto nově načteného operátoru. Vyjmuté operátory jsou jeden po druhém, v pořadí, jak je z vrcholu zásobníku postupně odebíráme, zaslány do výstupního zásobníku, kde zpracují příslušný počet operandů z vrcholu zásobníku (jeden pro unární operátor, dva pro binární operátor).

Závorkám Dijstrův algoritmus přiřazuje stejně jako ostatním operátorům určitou prioritu, a to menší než jsou priority ostatních operátorů (z těch, které budeme pro náš program využívat). Nicméně bez ohledu na prioritu je po načtení levé závorky tato vložena do pomocného zásobníku. Tímto je ale docíleno toho, že tuto levou závorku z vrcholu pomocného zásobníku je možné z tohoto zásobníku odstranit, až pokud z toku přicházejících symbolů je načtena závorka pravá, tedy přednostně je ve shodě s očekáváním provedeno vyhodnocení uzávorkovaného výrazu. Po načtení pravé závorky se tato na pomocný zásobník neukládá, pouze se z vrcholu pomocného zásobníku postupně vyjmou všechny operátory a přesunou se do výstupního zásobníku až po levou závorku, která se z pomocného zásobníku odstraní.

Algoritmus podle [5] končí po zpracování posledního načteného vstupního symbolu tak, že z vrcholu pomocného zásobníku postupně vybere všechny zbývající operátory a přesune je do výstupního zásobníku ke zpracování.

3.3 Algoritmus Quine–McCluskey

Jako vstup pro algoritmus Quine–McCluskey budeme používat úplný DNT formule, kterou máme za úkol minimalizovat (resp. její negace v případě hledání minimálních KNT). Algoritmus ale může na svém vstupu akceptovat obecně jakýkoli DNT formule, nejen úplný. Například pro případy, kdy hledáme minimální DNT výrokové formule, která je sama o sobě již v DNT, by tento postup byl efektivnější. Nicméně řešení přes dohledání úplného normálního tvaru je univerzálnější a bez nutnosti vytváření dalšího syntaktického analyzátoru, který by rozpoznával

zápis formulí v normálním tvaru. Stručně proto shrňme algoritmus k nalezení úplného DNT a přidejme i algoritmus, jak získat úplný KNT.

Úplný DNT i úplný KNT konkrétní výrokové formule můžeme určit z její pravdivostní tabulky ([2]). Mějme výrokovou formuli A o n prvotních formulích. Vyjdeme z definice 3.5 a stanovíme pravdivostní hodnotu $v(A)$ formule A pro každé z jejích 2^n (viz poznámka 3.6) možných pravdivostních ohodnocení v. Již odtud je zřejmé, že naše implementace hledání minimálních normálních tvarů nemůže mít menší složitost než exponenciální.

Podle [2] získáme úplný DNT z pravdivostní tabulky tak, že každý jednotlivý minterm, z jejichž disjunkce hledaný úplný DNT nakonec vznikne, sestavíme z jednoho ohodnocení v, pro které je formule *pravdivá*. Samotný minterm pak vytvoříme jako konjunkci prvotních formulí – v případě, že v daném ohodnocení měly hodnotu 1, a negací prvotních formulí – v případě, že v daném ohodnocení měly hodnotu 0. Obecně tak získáme 1 až 2^n mintermů, které budou obsahovat právě n literálů, avšak vyjma případu, kdy ani jedno z ohodnocení není pravdivé – tedy se jedná o kontradikci. Tento případ jsme ale vyřešili rovněž v souladu s [2] již v poznámce 3.13 a za úplný DNT tedy pro takový případ budeme považovat symbol $\bot$.

Podle [2] získáme úplný KNT z pravdivostní tabulky analogicky tak, že každý jednotlivý maxterm, z jejichž konjunkce hledaný úplný KNT nakonec vznikne, sestavíme z jednoho ohodnocení v, pro které je formule *nepravdivá*. Samotný maxterm pak vytvoříme jako disjunkci prvotních formulí – v případě, že v daném ohodnocení měly hodnotu 0, a negací prvotních formulí – v případě, že v daném ohodnocení měly hodnotu 1 (tedy opačně než v případě hledání úplného DNT). Obecně tak analogicky získáme 1 až 2^n maxtermů, které budou obsahovat právě n literálů, avšak vyjma případu, kdy jsou všechna ohodnocení pravdivá – tedy se jedná o tautologii. Tento případ jsme však rovněž vyřešili dle [2] již v poznámce 3.13 a za úplný KNT tedy pro takový případ budeme považovat symbol $\top$.

▸ Poznámka 3.16. Pro úplné normální tvary formulí, které jsou sestaveny podle výše uvedeného postupu, narozdíl od minimálních normálních tvarů, platí kromě obecného existenčního tvrzení (viz věta 3.14) dokonce ještě silnější existenční tvrzení, které udává, že, v jistém smyslu, k dané formuli existuje právě jeden úplný DNT a právě jeden úplný KNT. Platí totiž, že (uvedeným postupem sestavený) úplný DNT (úplný KNT) dané formule je určen jednoznačně až na pořadí mintermů (maxtermů) a na pořadí literálů v těchto mintermech (maxtermech). Proto narozdíl od hledání všech minimálních DNT a všech minimálních KNT dané výrokové formule má smysl hledat jen jeden úplný DNT a jeden úplný KNT k příslušné výrokové formuli.

Algoritmus Quine–McCluskey, tak jak byl uveden v [7], hledá minimální DNT. Nicméně s odkazem na [2] můžeme hledání minimálního KNT převést na hledání minimálního DNT tak, že zkoumanou formuli znegujeme, nalezneme její minimální DNT a ten opět znegujeme – ve smyslu, jak jej uvádí [1], tedy vzájemně zaměníme symboly $\wedge$ a $\vee$ a jednotlivé literály nahradíme jejich negacemi.

Samotný algoritmus Quine–McCluskey můžeme rozdělit stejně, jako je uvedeno v článku [7], na dvě fáze. V první fázi, jejímž vstupem je DNT minimalizované logické formule (bez újmy na obecnosti např. úplný DNT nalezený dle výše popsaného algoritmu) najdeme nejkratší možné implikanty. V druhé fázi pak z těchto nalezených nejkratších implikantů vybereme jen některé tak, aby dohromady tvořili minimální DNT.

Ještě dříve, než začneme popisovat hledání minimálních normálních tvarů pomocí algoritmu Quine–McCluskey, vyřešme minimalizaci pro speciální případy formulí – tautologie a kontradikce.

- Pro případ tautologie má úplný KNT dle poznámky 3.13 prázdnou množinu klausulí, tj. počet klausulí je roven 0. Menší počet již být nemůže, a tedy symbol $\top$ zastupuje jak úplný KNT, tak minimální KNT. Pokud budeme chtít hledat minimální DNT tautologie, skončí algoritmus Quine–McCluskey s nulovým počtem implikantů, a proto i pro tento případ dodefinujeme výsledek jako symbol $\top$.
- Pro případ kontradikce má úplný DNT dle poznámky 3.13 prázdnou množinu mintermů, tj. počet mintermů je roven 0. Menší počet již být nemůže, a tedy symbol $\bot$ zastupuje jak úplný DNT, tak minimální DNT. Pokud budeme chtít hledat minimální KNT kontradikce, převedli bychom tento případ na hledání minimálního DNT její negace, tedy tautologie, jejíž výsledek jsme definovali jako symbol $\top$. Na tento výsledek bychom měli aplikovat negaci (zaměnit symboly $\wedge$ a $\vee$ a jednotlivé literály nahradit jejich negacemi), a proto i pro tento případ analogicky dodefinujeme výsledek jako symbol $\bot$.

Následující popis těchto dvou fází je zobecněním příkladu v [1]. Naším úkolem je nalézt všechny minimální DNT logické formule, která obsahuje, bez újmy na obecnosti, n různých prvotních formulí ($n \geqslant 1$), není tautologií ani kontradikcí a máme již nalezen její úplný DNT, který nutně obsahuje m mintermů, kde $1 \leqslant m \leqslant 2^n - 1$.

3.3.1 Hledání nejkratších mintermů

Na začátku první fáze provedeme podle ukázkového příkladu v [1] seskupení mintermů úplného DNT formule A podle počtu neznegovaných prvotních formulí. Snadno nahlédneme, že skupin bude $n+1$, přičemž některé z nich mohou zůstat prázdné, a mintermy v nich obsažené se skládají z n literálů, protože vstupem byl právě úplný DNT.

Dále připravme tabulku modifikující vizualizaci uvedenou v [1]. Do prvního sloupce budeme zapisovat mintermy, resp. implikanty, které postupem času vzniknou, do dalších m sloupců budeme zapisovat, který z původních mintermů úplného DNT pokrývá minterm/implikant daného řádku, a navíc přidáme ještě jeden sloupec, který využijeme jako pomocný, abychom měli poznamenáno, který minterm/implikant byl použit ke slučování. V této tabulce v její konečné podobě budou obsaženy hledané nejkratší implikanty spolu s dalšími informacemi, které budou využity v následující fázi při sestavování minimálního DNT.

Počáteční zápis do tabulky provedeme podle vzoru v [1] následovně – do sloupce s mintermy/implikanty vepíšeme ty mintermy, které byly obsaženy v dohledaném úplném DNT. Prvotní tabulka tedy bude mít m řádků. Do každého řádku s konkrétním mintermem vepíšeme dále do sloupce odpovídajícího danému mintermu značku, která bude implikovat, že původní minterm z tohoto sloupce je pokryt mintermem daného řádku.

Nyní přistoupíme ke slučování mintermů, které se liší pouze v jednom literálu, a to tak, že jeden minterm obsahuje tento literál jako prvotní formuli a druhý minterm obsahuje tento literál ve formě negace této prvotní formule. Vzhledem k tomu, že máme mintermy rozděleny stejně jako v [1] do skupin podle počtu neznegovaných prvotních formulí, postačí, abychom tyto dvojice mintermů hledali ve skupinách, které se od sebe liší počtem neznegovaných prvotních formulí o 1.

Ke sloučení mintermů nemusí vůbec dojít, pokud neexistuje žádná dvojice vhodná ke sloučení. Naopak, pokud v rámci zpracování jedné sady skupin mintermů ke sloučení dojde, je třeba vytvořit novou sadu skupin implikantů (nikoli již mintermů), která bude sloužit jako podklad pro další cyklus pokusů o slučování. Tato nová sada skupin bude mít, jak je zřejmé, o jednu

skupinu implikantů méně a bude obsahovat implikanty s počtem literálů o jedna menším, než ve skupinách z předchozí sady.

Při slučování mintermů a posléze implikantů (dále již jen obecné označení implikanty) postupujeme podle instrukcí v [1]: Najdeme-li dvojici implikantů, které mají požadovanou odlišnost, vytvoříme nový implikant tak, že v původních implikantech vynecháme literál, který je odlišoval. V pomocné tabulce zapíšeme do pomocného sloupce do řádků se slučovanými implikanty příznak, že implikanty již byly použity ke sloučení – pokud už tam taková značka je, ponecháme ji tam – neoznačené řádky budou znamenat, že implikant nebylo možno sloučit, což podle [1] implikuje to, že se již jedná o minimální implikanty, ze kterých výhradně bude minimální tvar dané formule sestávat. Příznak, že byl implikant použit ke sloučení, však nebrání tomu, aby byl jeden a ten samý implikant použit při více sloučeních, pokud jsou splněny ostatní podmínky pro sloučení.

Následně musíme zjistit, jestli nově vzniklý implikant již je v tabulce obsažen, protože podle [1] můžeme duplicity ignorovat. Pokud obsažen dosud není, vytvoříme pro něj další řádek a okopírujeme k němu všechny značky, které u implikantů, z nichž vznikl, určovaly, které z mintermů původního úplného DNT pokrývají. Rovněž ho vložíme do příslušné skupiny v nové sadě.

Výjimku při slučování tvoří případy, kdy se mají sloučit implikanty o jednom literálu. Vynecháním takového literálu bychom nezískali žádný nový implikant do hledaného minimálního normálního tvaru. Proto v takovém případě nevytváříme žádný nový implikant, pouze do tabulky ke každému ze slučovaných implikantů uvedeme příznak, že byly použity ke sloučení.

Pokud v rámci zpracování jedné sady skupin implikantů nedojde k žádnému sloučení nebo byl dokončen cyklus slučování skupin obsahujících implikanty tvořené jedním literálem, tato fáze končí a výsledná tabulka sestávající z řádků, u kterých není poznámka, že byly použity ke sloučení, je předána do další fáze zpracování, neboť podle [1] jsou implikanty, které obsahuje, sice nejkratší možné, ale jejich disjunkce ještě nemusí být minimální.

3.3.2 Sestavení minimálního DNT

Do druhé fáze, ve které budeme podle návodu naznačeného v [7] sestavovat z nalezených minimálních implikantů hledaný minimální DNT, vstupuje tabulka s řádky pro každý implikant, který se může v hledaném minimálním DNT vyskytnout, a s určením, které původní mintermy tento minimální implikant implikuje. Pokud je některý z původních mintermů implikován pouze jedním z nalezených minimálních implikantů, je zřejmé, že bude muset být v minimálním DNT obsažen. O takových minimálních implikantech hovoříme jako o tzv. *esenciálních* implikantech. Pokud by všechny minimální implikanty byly implikanty esenciálními, získali bychom jejich disjunkcí požadovaný minimální DNT, v opačném případě je třeba vybrat vhodné implikanty např. právě dle algoritmu v [7].

Najdeme-li v souboru minimálních implikantů nějaké esenciální implikanty, můžeme o ně pomocnou tabulku zjednodušit vynecháním dotčeného řádku a rovněž můžeme snížit počet sloupců s původními mintermy, které jsou těmito esenciálními implikanty již implikovány. Není to však nezbytně nutné, protože obecný postup dle [7] pokryje i tento případ.

Nyní je třeba ve shodě s definicí minimálního DNT najít v prvém kroku nejmenší počet k minimálních implikantů, které budou implikovat všechny zbývající původní mintermy, a samozřejmě i konkrétní k-tice minimálních implikantů, které toto splňují. Těchto k-tic tvořených minimálními implikanty může být více a ve druhém kroku z nich vybereme ty, které mají nejméně literálů. Následující kroky sledují postup uvedený v [7].

V rámci prvého kroku si zavedeme symbolické označení minimálních implikantů, které nám po redukci pomocné tabulky zbyly, např. $X1$, $X2$, ..., a budeme na ně nahlížet jako na prvotní formule výrokové logiky v tomto novém jazyce.

Pro každý původní minterm provedeme disjunkci jednotlivých prvotních formulí zastupujících implikanty, které daný původní minterm implikují, a z takto vzniklých klausulí sestavíme konjunkcí novou pomocnou výrokovou formuli v KNT. Takto zapsanou pomocnou formuli budeme dále upravovat tak, abychom získali tuto pomocnou výrokovou formuli v DNT. Využijeme k tomu algebraické vlastnosti spojek 3.8, zejména distributivní zákony. Již během distribuce závorek můžeme vznikající formuli průběžně zjednodušovat s využitím dalších zákonitostí, např. slučováním ekvivalentních podformulí.

Cílem výše popsaného kroku převzatého z [7] je vytvořit DNT pomocné formule, který sestává z navzájem různých implikantů, které obsahují každou z prvotních formulí nejvýše jednou. Každý z takových implikantů pomocné formule vytvoří po zpětném dosazení původních minimálních implikantů za jednotlivé literály DNT pomocné výrokové formule výrokovou formuli logicky ekvivalentní s původní výrokovou formulí, jejíž minimální DNT hledáme. Ty minimální z nich ve smyslu definice 3.11 vybereme následovně: Z konečného DNT pomocné formule zjistíme, jaký je nejmenší počet literálů v jednom implikantu, to je námi hledané číslo k, a ponecháme pouze implikanty s tímto nejmenším počtem – obecně jich může být více než jeden. Každý takový pomocný implikant určuje k-tici literálů, které zastupují minimální implikanty.

V předchozím kroku jsme našli jednu či více k-tic, které implikují všechny zbývající původní mintermy, které nebyly pokryty esenciálními implikanty. Pokud je počet k-tic roven 1, našli jsme ve skupině implikantů, které odpovídají prvotním formulím v k-tici, spolu s případnými esenciálními implikanty po jejich disjunkci hledaný minimální DNT. Pokud je takových k-tic více, je třeba ve shodě s definicí 3.11 zjistit, které z nich mají nejmenší počet literálů, tedy provést druhý krok.

Pro každou k-tici, která vyšla z předchozího kroku, zjistíme počet literálů, a to tak, že sečteme počet literálů, které obsahují implikanty zastoupené jednotlivými symboly v k-tici. Z k-tic pak vybereme ty, které mají tento součet shodný s nejmenším z těchto součtů. Obecně jich opět může být více a jejich počet určuje, kolik minimálních DNT původní výrokové formule jsme našli. Každý z nich pak můžeme sestavit z dané k-tice jako disjunkci implikantů, které jsou určeny symboly v této k-tici, a případných esenciálních implikantů.

Kapitola 4
Návrh

V kapitole 3 jsme předestřeli jednotlivé procesy, které bude třeba do vnitřku našeho programu implementovat. V této kapitole navrhneme, jak se bude program chovat navenek, tedy ve vztahu k jeho uživateli.

Za tím účelem nejprve navrhneme formální jazyk, ve kterém bude uživatelem zadávána formule určená k minimalizaci, zachycující syntaxi formulí výrokové logiky, jak jsme ji zavedli v definici 3.3, a tento formální jazyk navíc rozšíříme ve shodě s definicí 3.10, aby mohl program na tyto dva typy vstupů adekvátně reagovat.

Následně navrhneme vlastní chování programu - jeho komunikaci s uživatelem. Program bude ve vztahu k uživateli poměrně jednoduchý - požadujeme po něm pouze to, aby uměl vzít od uživatele výrokové formule a předal uživateli požadovaný výstup. Využijeme proto možnosti vytvořit program tak, aby byl spustitelný z konzole, kde si ze standardního vstupu načte zadání a výsledek vypíše na standardní výstup.

4.1 Zápis formulí výrokové logiky

Z definice 3.3 vyplývá, že pro zachycení syntaxe výrokových formulí je nutné navrhnout formu zápisu symbolů pro prvotní formule, symbolů pro logické spojky a případně další pomocné symboly. Symboly máme vybrat ze standardizovaných symbolů – z tzv. Amerického standardního kódu pro výměnu informací ([11], v angl. originálu American Standard Code for Information Interchange – ASCII, překlad autor), přičemž se omezíme pouze na její základní (tj. sedmibitovou) podobu.

Pro symboly prvotních formulí můžeme vyjít z definice 3.1 a ponechat jejich zápis dle zde uvedených pravidel, tj. každé prvotní formuli přiřadíme velké tiskací písmeno.

Symboly pro logické spojky z definice výrokové formule 3.3 $\neg$, $\wedge$, $\vee$, $\Rightarrow$ ani $\Leftrightarrow$ nepatří mezi ASCII znaky základní tabulky, a je tedy třeba zvolit jiný způsob zápisu. Navržené řešení by mělo být pro uživatele snadno zapamatovatelné a jednoduché pro zápis. Vyloučíme nejprve víceznaková seskupení, tím docílíme určité jednoduchosti. Nelogické, a tedy hůře zapamatovatelné, by bylo i využití číslic.

Vzhledem k tomu, že velká písmena již máme rezervována pro prvotní formule, nabízí se nám řešení v podobě malých písmen. Pro pět logických spojek tedy vybereme pět různých malých

písmen. Ve shodě s požadavkem snadné zapamatovatelnosti můžeme vzít vždy první písmeno z názvu logické spojky, tedy pro symboly logických spojek navrhujeme používat znaky n, c, d, i a e, přiřazené takto po řadě logickým spojkám negace, konjunkce (počáteční písmeno z anglického conjunction), disjunkce, implikace a ekvivalence.

Výrokovou formuli budeme chtít zapisovat v infixovém tvaru, a tedy s odkazem na definici formule výrokové logiky 3.4, ze které převezmeme postupnou (induktivní) výstavbu složitější formule, budeme potřebovat ještě pomocné znaky pro závorky. Potřebujeme, aby se jednalo o dvojici různých znaků – otevírací a uzavírací symbol. V tomto případě zůstaneme u klasických závorek (a), přestože ASCII tabulka obsahuje i další možnosti jako jsou závorky hranaté, složené či znaky pro nerovnost < a >, které bychom případně také mohli použít.

V rámci zápisu formule budeme ještě povolovat jako bílý znak mezeru, aniž by měla v rámci formule syntaktický význam, ale může uživateli pomoci rozčlenit zapsaný řetězec znaků a zlepšit tím přehlednost.

Přehlednost formule by naopak mohlo zhoršovat velké množství závorek přibývajících po dvou s každou binární logickou spojkou. Ve shodě s [2] nebudeme vyžadovat vnější závorky, ani závorky v případě více po sobě následujících konjunkcí, resp. disjunkcí. Navíc nebudeme požadovat ani jiné další závorky s tím, že pokud nebude pomocí závorek určena přednost operací jinak, bude mít největší přednost spojka negace, dále konjunkce následovaná disjunkcí, pak implikace a nakonec ekvivalence, přičemž stejné spojky budou vyhodnocovány postupně zleva. Takto navržená syntaxe bude připravena k syntaktické analýze podle námi popsaného Dijkstrova algoritmu seřaďovacího nádraží bez větších úprav.

Tím je vyřešen vstup směrem od uživatele, nicméně musíme počítat s tím, že výstupem z programu může být symbol $\top$, resp. $\bot$, jako zástupné symboly pro tautologii, resp. kontradikci (viz podkapitola Algoritmus Quine–McCluskey 3.3). Tyto symboly nepatří mezi ASCII znaky, a měli bychom tedy zvolit jiné označení. Protože ale jde o formální vyjádření typu formule a nikoli o formuli jako takovou, se kterou by se dále pracovalo, necháme program v místech, kde by měl takovéto symboly uvést, vypsat tautologii, resp. kontradikci slovně.

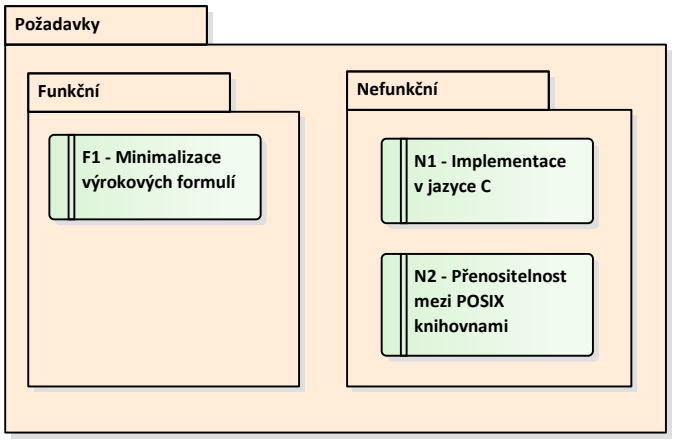
4.2 Uživatelské rozhraní programu

Pro popis návrhu programu využijeme vyjadřovací schopnosti grafického jazyka – tzv. Unifikovaného modelovacího jazyka ([12], v angl. originálu Unified Modeling Language – UML, překlad autor).

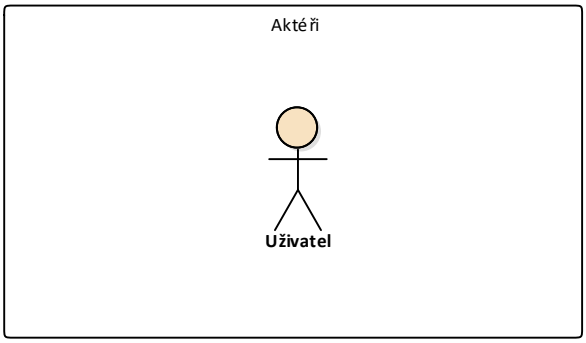
S odkazem na cíle této bakalářské práce je hlavním funkčním požadavkem na vytvářený program požadavek *F1 – Minimalizace výrokových formulí* (obr. 4.1 Požadavky). Hlavními nefunkčními požadavky jsou pak požadavek na implementační programovací jazyk, a to požadavek *N1 – Implementace v jazyce C*, a požadavek na kompatibilitu s jinými systémy, tedy požadavek *N2 – Přenositelnost mezi POSIX knihovnami* (taktéž obr. 4.1 Požadavky).

Aktéry v rámci funkčního požadavku budou dále nerozlišení uživatelé programu (obr. 4.2 Aktéři), kteří od programu požadují, aby si od nich program načetl zadání a zobrazil na standardní výstup požadovaný výsledek.

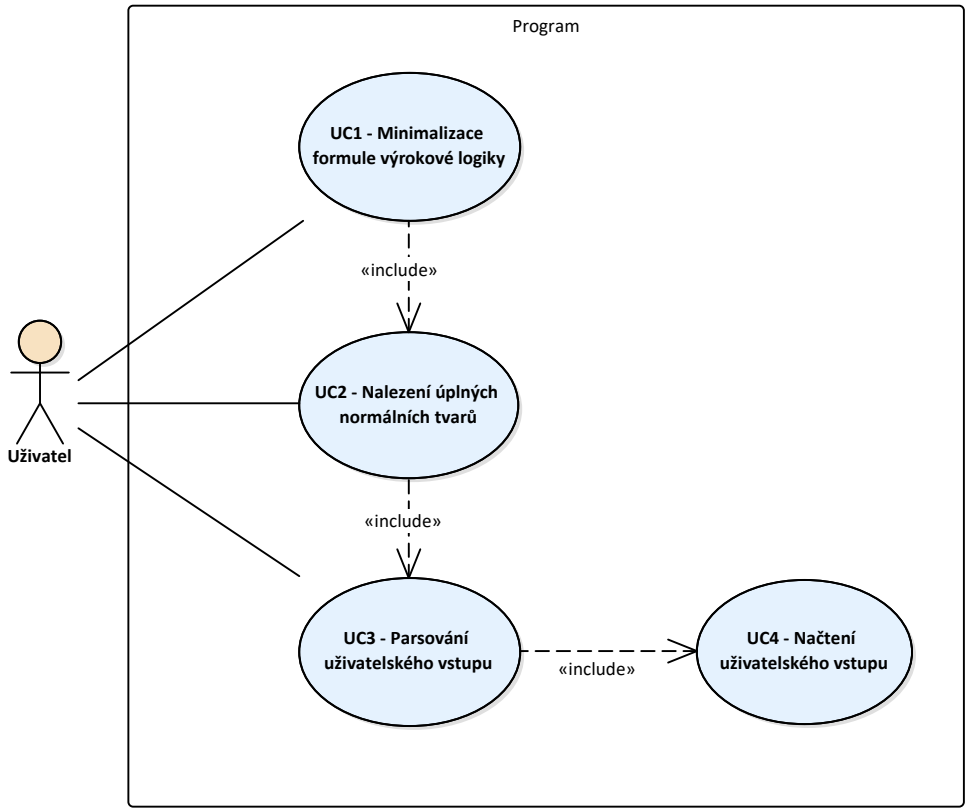
Chování programu korespondující s výše uvedenými požadavky uživatelů můžeme navrhnout do větších detailů takto (obr. 4.4 Diagram aktivit): Program bude vytvořen tak, aby byl z konzole spustitelný buď bez přepínačů, či s určitými přepínači, kterými bude uživatel volit typy výstupu.



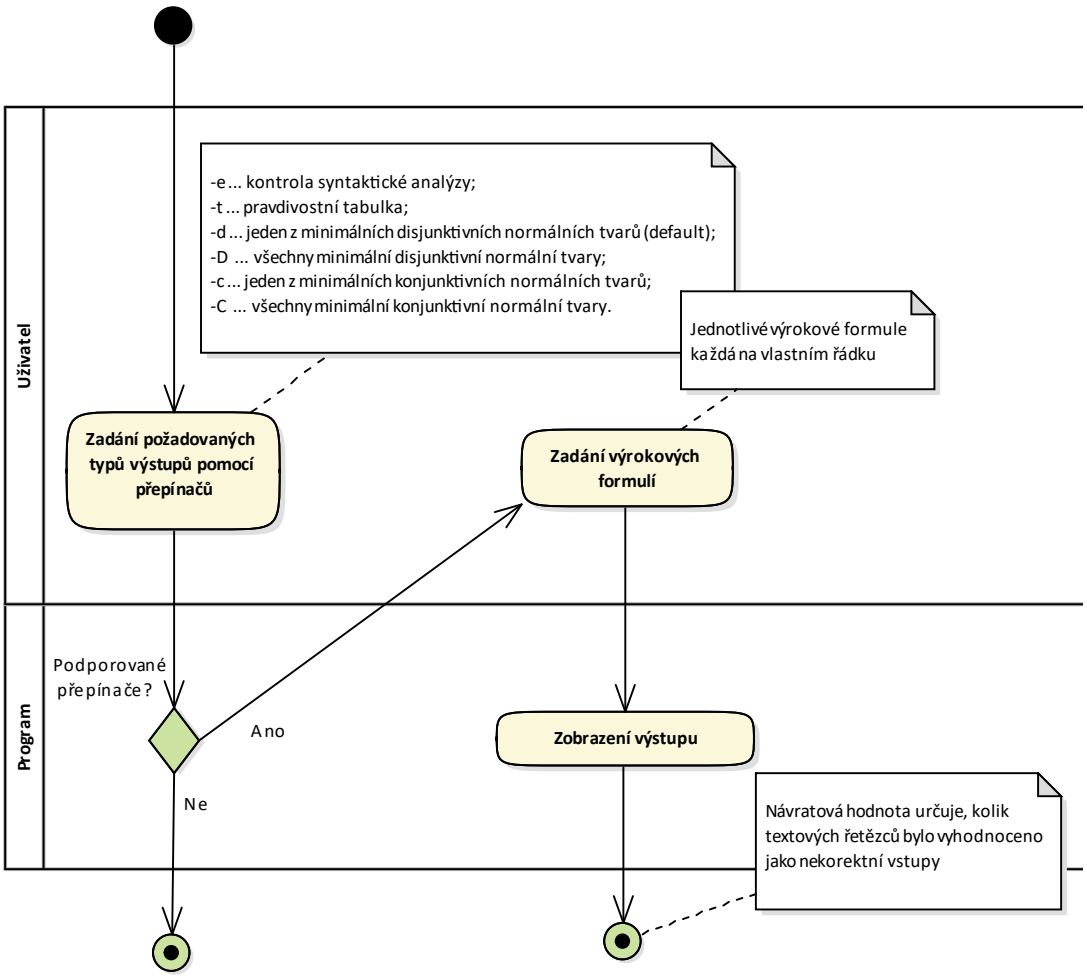
Obrázek 4.1 Požadavky



Obrázek 4.2 Aktéři



Obrázek 4.3 Případy užití



Obrázek 4.4 Diagram aktivit

Tabulka 4.1 Mapování případů užití na požadavky

	Případy užití			
Požadavky	UC1 – Minimalizace formule výrokové logiky	UC2 – Nalezení úplných normálních tvarů	UC3 – Parsování uživatelského vstupu	UC4 – Načtení uživatelského vstupu
F1 – Minimalizace výrokových formulí	MAP	MAP	MAP	MAP

Program zkontroluje správnost přepínačů a bude-li tato kontrola úspěšná, vrátí požadovaný výstup, tedy pro případ bez přepínače, který ztotožníme s případem použití přepínače `-d`, vrátí některý z minimálních DNT, pro přepínač `-D` vrátí všechny minimální DNT, analogicky pro přepínač `-c` vrátí jeden z minimálních KNT, pro přepínač `-C` všechny minimální KNT.

Pravdivostní tabulku, jako zápis úplného DNT, resp. KNT, které má program rovněž umět vypsat, vrátí program při použití přepínače `-t` a jako ukazatel správnosti zadané formule bude sloužit přepínač `-e`, který v případě korektně zadané formule vrátí tuto výrokovou formuli ve zpětně zkonstruované textové podobě.

Přepínače je možno různě kombinovat, přičemž výstup při použití přepínače `-d` považujeme za součást výstupu při použití přepínače `-D`, tedy v případě použití kombinace přepínačů `-d` a `-D` bude uživateli vrácen pouze výstup z přepínače `-D` a analogicky pro přepínače `-c` a `-C`. Při použití více přepínačů bude mít výstup pro každou z korektních výrokových formulí pořadí přepínačů `-e`, `-t`, `-c`, resp. `-C` a `-d`, resp. `-D`.

V případě zadání nepodporovaných přepínačů bude program bez dalšího ukončen se zápornou návratovou hodnotou a upozorněním na chybovém výstupu, které přepínače jsou podporovány. V opačném případě program načte ze standardního vstupu výrokové formule. Výrokových formulí (textových řetězců) může uživatel zadat více, přičemž za jeden textových řetězec k syntaktické analýze se považuje jedna řádka a načítání se zastaví s koncem vstupu, případně u první prázdné řádky. Následuje ze strany programu vlastní zpracování jednotlivých načtených řádků. V případě vyhodnocení textového řetězce jako korektně zadané formule, je uživateli vrácen požadovaný výstup, na nekorektní vstup není vrácen výstup žádný. Po svém ukončení bude návratová hodnota programu udávat, kolik textových řetězců bylo vyhodnoceno jako nekorektní vstupy.

Rozborem požadovaného chování programu se pak dostáváme k jednotlivým případům užití (obr. 4.3 Případy užití). Aktér uživatel je přímo svázán s případem užití *UC1 – Minimalizace*

formule výrokové logiky, který odpovídá očekávání uživatele od programu, který mu jako odpověď na zadanou formuli vrátí její minimalizovanou podobu.

Stejně tak je uživatel svázán i s případem užití *UC2 – Nalezení úplných normálních tvarů*, který je však potřeba včlenit do programu i pro předchozí případ užití *UC1 – Minimalizace formule výrokové logiky*, neboť úplný DNT využíváme jako vstup pro algoritmus Quine–McCluskey, který je využit k vlastní minimalizaci výrokové formule.

Naposledy je uživatel přímo svázán s případem užití *UC3 – Parsování uživatelského vstupu*, kterým program převede výrokovou formuli vloženou uživatelem jako textový řetězec do potřebné struktury, v rámci čehož provede kontrolu validity vstupu, a tedy případně celý proces minimalizace výrokové formule zastaví. Logicky je tento případ užití *UC3 – Parsování uživatelského vstupu* potřebný pro případ užití *UC2 – Nalezení úplných normálních tvarů*, potažmo pro případ užití *UC1 – Minimalizace formule výrokové logiky*.

Případ užití *UC3 – Parsování uživatelského vstupu*, a tím nepřímo i případy užití *UC1 – Minimalizace formule výrokové logiky* a *UC2 – Nalezení úplných normálních tvarů*, v sobě zahrnuje ještě jeden další případ užití, a to *UC4 – Načtení uživatelského zadání*, který zajišťuje získání zadání od uživatele.

Realizace požadavku *F1 – Minimalizace výrokových formulí* se pak rozpadá na implementaci čtyř případů užití (tab. 4.1 Mapování případů užití na požadavky), které dohromady jako celek s přihlédnutím k nefunkčním požadavkům *N1 – Implementace v jazyce C* a *N2 – Přenositelnost mezi POSIX knihovnami* vytvoří požadovaný program. Tyto nefunkční požadavky pak splníme provedením implementace v jazyce C v souladu se standardem IEEE Std 1003.1, což je standard pro rozhraní přenositelné mezi operačními systémy označované jako POSIX (z angl. originálu Portable Operating System Interface), tedy s využitím knihoven tohoto jazyka tímto standardem definovaných (přehled hlavičkových souborů viz [13]).

Uživatelská příručka bude uživatelům k dispozici ve formátu standardní manuálové stránky (více viz [14]).

Kapitola 5
Realizace

V této kapitole projdeme postupně implementaci jednotlivých případů užití, které jsme identifikovali v kapitole Návrh v podkapitole Uživatelské rozhraní programu (4.2), a to v pořadí, v jakém je bude program postupně spouštět.

Zdrojový kód byl rozdělen podle jednotlivých případů užití do samostatných modulů – souborů `minform.c` pro *UC1 – Minimalizace formule výrokové logiky*, `table.c` pro *UC2 – Nalezení úplných normálních tvarů* a `formula.c` pro *UC3 – Parsování uživatelského vstupu*. Implementace případu užití *UC4 – Načtení uživatelského zadání* je triviální a byla ponechána přímo v hlavním souboru `minim.c`. Hlavní soubor obsahuje především funkci `main()`, která zodpovídá za načtení uživatelského zadání, sama o sobě obsahuje zmíněný případ užití *UC4 – Načtení uživatelského zadání* a zajišťuje spouštění potřebných funkcionalit z ostatních modulů. K uvedeným zdrojovým souborům patří příslušné hlavičkové soubory `minform.h`, `table.h`, `formula.h` a `minim.h`.

Vzájemnou návaznost modulů pro sestavování výsledného programu pak určíme v hlavičkových, případně zdrojových souborech pomocí funkcionality `#include`. Touto funkcionalitou rovněž připojíme do programu několik hlavičkových souborů standardní knihovny jazyka C, které budeme potřebovat. Pro jednotlivé moduly a hlavní zdrojový soubor byly podle potřeby použity v prvé řadě hlavičkový soubor `stdlib.h`, dále hlavičkový soubor pro standardní vstup a výstup `stdio.h`, hlavičkový soubor pro práci s textovými řetězci `string.h` a hlavičkový soubor pro práci s jednotlivými znaky `ctype.h`. Pro práci s chybami, ke kterým může v programu dojít (typicky nedostatek paměti), byl inkludován i hlavičkový soubor `err.h`. Dynamicky alokovanou paměť budeme vytvářet pomocí knihovní funkce `calloc`, která paměť alokuje a zároveň zařídí její zformátování (vynulování). Právě pro případy, kdy tato funkce vrátí hodnutu `NULL`, tedy nedojde k řádné alokaci požadované paměti, využijeme z hlavičkového souboru `err.h` v něm obsaženou funkci `err()`, pomocí které program okamžitě ukončujeme a zabraňujeme jeho neřízenému pádu. Mimo C standardní knihovnu (libc) definovanou standardem ANSI C (součást standardu POSIX) využijeme ještě hlavičkový soubor `unistd.h` poskytovaný v rámci širší C knihovny POSIX, který nám mimo jiné usnadní zpracování zadání od uživatele.

Protože budeme v rámci implementace často využívat rozhodnutí, zda něco je, či není pravda, zavedeme globální konstanty `TRUE` a `FALSE`, přičemž se bude jednat o konstanty typu integer a budou zastupovat hodnoty 1 pro `TRUE` a 0 pro `FALSE`. Tyto konstanty deklarujeme (a současně definujeme) v odděleném hlavičkovém souboru `minim.h`, aby mohly být používány napříč všemi zdrojovými soubory.

Program jako celek má od svého spuštění pracovat jednorázově, jak je dobře vidět z diagramu aktivit 4.4, a po provedení přepínači specifikovaného požadavku vrátit výstup a ukončit se. Takto

popsané chování je implementováno v hlavní funkci programu `main()`, která je umístěna ve zdrojovém souboru `minim.c`. Funkce převezme parametry, se kterými byl program spuštěn – první parametr funkce `argc` určuje počet předávaných parametrů a samotné parametry jsou pak předány jako pole textových řetězců `argv[]`.

Pomocí funkce `getopt()` (z hlavičkového souboru C knihovny POSIX) zjistíme, které z přepínačů uživatel zadal, a uložíme je k pozdějšímu využití do předem připravených proměnných, přičemž ošetříme případ, že nebyl zadán přepínač žádný, a to tak, že nastavíme jako „požadovaný“ výstup ten, který odpovídá přepínači `-d`. Pokud uživatel zadal přepínač, který není podporován (funkci `getopt()` předáváme mimo jiné parametr, který obsahuje výčet podporovaných přepínačů), program zastavujeme se zápornou návratovou hodnotou a na chybový výstup zasíláme zprávu se stručným návodem na použití programu (`usage: minim [-CDcdet]`).

Při správně zadaném pokynu uživatele ke spuštění programu pokračuje hlavní funkce `main()` vlastním obslužným cyklem – cyklickým načítáním samotného zadání výrokových formulí. Cyklus končí s první prázdnou řádkou. Pro každou takto načtenou řádku se dále spustí jednotlivé procesy k vytvoření požadovaného výstupu specifikovaného použitými přepínači. V rámci obslužného cyklu jsou tedy spouštěny postupně tyto funkce, z nichž každá odpovídá jednomu případu užití:

- funkce `getline()` (z hlavičkového souboru C standardní knihovny `stdio.h`) odpovídá případu užití *UC4 – Načtení uživatelského zadání*;
- funkce `parseInput()` zodpovídá za implementaci případu užití *UC3 – Parsování uživatelského vstupu* a je deklarována v hlavičkovém souboru modulu `formula`;
- funkce `buildTruthTable()` je implementací případu užití *UC2 – Nalezení úplných normálních tvarů* a je deklarována v hlavičkovém souboru modulu `table`;
- funkce `minimize()` zajistí implementaci případu užití *UC1 – Minimalizace formule výrokové logiky* a je deklarována v hlavičkovém souboru modulu `minform`

Než se budeme věnovat implementacím případů užití každé zvlášť, zmiňme, že ne všechny funkce odrážející jednotlivé případy užití musí být vždy spuštěny. Jediná taková funkce, která bude v rámci každého jednotlivého běhu obslužného cyklu spuštěna vždy, je funkce `getline()`, protože pokud načte prázdnou řádku, program je ukončen. Další funkce jsou volány podle potřeby, a to v souladu s provázaností jednotlivých případů užití (viz diagram 4.3). Funkce `minimize()` může být volána až dvakrát – pokud je uživatelský požadavek na minimální DNT i minimální KNT, ale nebudeme ji volat vůbec, pokud v požadavcích nefiguruje. Stejně tak nebudeme volat funkci `buildTruthTable()`, pokud uživatel požadoval pouze výstup odpovídající přepínači `-e`. Samozřejmě také nebudeme volat příslušné navazující funkce v případě nekorektního zadání výrokové formule, což s konečnou platností rozhoduje funkce `parseInput()`, i když uživatel výstupy z nich požadoval.

5.1 Načtení uživatelského vstupu

Jak jsme již uvedli, případ užití *UC4 – Načtení uživatelského zadání* je implementován přímo v hlavním zdrojovém souboru `minim.c` ve funkci `main()` pomocí funkce `getline()`, která do připraveného odkazu vloží celou řádku (posloupnost znaků vyskytujících se před znakem pro ukončení řádku včetně tohoto ukončovacího znaku). Protože poslední načtený znak – znak `'\n'` pro ukončení řádku nemá být součástí uživatelského zadání, zajistíme před dalším zpracováním jeho nahrazení znakem `'\0'` pro ukončení textového řetězce.

Výpis kódu 5.1 Rozhraní modulu formula

```
1  typedef struct SPrim {
2      char label;
3      char value;
4  } SPRIM;
5
6  typedef struct SNode {
7      char type;
8      SPRIM* primForm;
9      struct SNode* lNode;
10     struct SNode* rNode;
11 } SNODE;
12
13 int findPrims(SNODE* node, SPRIM** prims, int first);
14 int getValue(SNODE* tree);
15 void freeTree(SNODE* node, int type);
16 SNODE* parseInput(char* input);
17 void printFormula(SNODE* fla);
```

Výpis kódu 5.2 Funkce buildNode()

```
1  SNODE* buildNode(char type, SPRIM* primForm,
2                               SNODE* lNode, SNODE* rNode){
3      SNODE* node;
4      if ((node = (SNODE*) calloc(1, sizeof (SNODE))) == NULL) {
5          err(1, NULL);
6      }
7      node->type = type;
8      node->primForm = primForm;
9      node->lNode = lNode;
10     node->rNode = rNode;
11     return node;
12 }
```

Funkce `getline()` sama zajistí alokaci pole potřebné délky. Z naší strany pouze zkontrolujeme, že k načtení řádku došlo v pořádku a že načtený vstup je nenulové délky, resp. nejedná se o prázdný řetězec. Podle výstupní hodnoty funkce `getline()` se tedy rozhodneme, zda pokračovat v dalších krocích směřujících k parsování vstupu a případně i k nalezení normálních tvarů a k minimalizaci obdrženého vstupu dle požadavku uživatele. V opačném případě (při načtení prázdné řádky) zastavíme načítání dalších řádků, zajistíme uvolnění dosud neuvolněné dynamicky alokované paměti a program ukončíme.

5.2 Parsování uživatelského vstupu

Modul `formula` obsahuje zejména funkci `parseInput()`, kterou pro každý relevantní řádek volá obslužný cyklus funkce `main()` a zajišťuje jejím prostřednictvím syntaktickou analýzu uživatelem zadaného textového řetězce. Vstupem do funkce `parseInput()` (viz výpis kódu 5.1, ř. 16) je uživatelský vstup ve formě textového řetězce, který byl získán v rámci obslužného cyklu. Výstupem funkce je pak kořen stromové struktury, do které budeme textový řetězec převádět, abychom v této podobě s výrokovou formulí následně pracovali. Volající procedura pak od této funkce očekává jako výstupní hodnotu tuto stromovou strukturu, případně ukazatel `NULL`, který

značí nesprávný uživatelský vstup. V případě nesprávného uživatelského vstupu je po návratu do obslužného cyklu inkrementována hodnota proměnné, která uchovává počet nekorektních vstupů. Hodnota této proměnné je pak při ukončení programu vrácena jako návratová hodnota. V rámci obslužného cyklu pak tento nekorektní vstup navíc zamezí spuštění dalších modulů a cyklus pokračuje v načítání další řádky.

Modul `formula` ale obsahuje ve veřejném rozhraní kromě funkce `parseInput()` ještě další funkce a procedury, které jsou pro jiné moduly potřebné (viz výpis kódu 5.1), a to proto, že modul `formula` zavádí novou strukturu reprezentující stromovou strukturu, obsahuje její definici a stanovuje, jak s ní mají pracovat ostatní moduly. Proceduru `freeTree()` (viz výpis kódu 5.1, ř. 15), která zajišťuje bezpečnou dealokaci paměti při uvolňování stromové struktury, bude potřebovat obslužný cyklus, který stromovou strukturu obdrží, ale její dealokaci už musí zajistit sám. Obslužný cyklus pak využije ještě jednu proceduru z modulu `formula`, a to `printFormula()` (viz výpis kódu 5.1, ř. 17), ve které je implementován výpis výrokové formule uložený v podobě stromové struktury. Výpis výrokové formule nám bude mimo jiné sloužit k ověření korektního načtení této výrokové formule z hlediska námi navrženého formálního jazyka, a proto jsme výpis implementovali jako striktně uzávorkovaný (dle definice 3.4), aby byla zřejmá přednost jednotlivých operací. Obslužný cyklus pak tuto proceduru volá v případě korektního vstupu po zadání přepínače `-e`.

Funkce `findPrims()` a `getValue()` (viz výpis kódu 5.1, ř. 13 a 14) bude naproti tomu potřebovat modul `table`, který si pomocí funkce `findPrims()` uloží do pomocného pole ukazatele na všechny primární formule implementované jako struktury `SPRIM`, což modulu `table` umožní, jak ještě dále uvidíme, nastavovat jejich pravdivostní hodnoty a po nastavení těchto pravdivostních hodnot všech primárních formulí pak můžeme v modulu `table` zjišťovat pomocí funkce `getValue()` pravdivostní hodnotu celé výrokové formule implementované jako stromovou strukturu složenou z uzlů, které implementujeme jako struktury `SNODE`.

Výše jsme zmínili zavedení nových struktur, a to `SPRIM` pro primární formuli a `SNODE` pro uzel stromové struktury, do které chceme textový řetězec převést. Primární formule, jak ji máme definovanou, je jednoznačně určena některým z velkých písmen. Navíc ale budeme potřebovat, aby nesla informaci o svém ohodnocení, a proto primární formuli implementujeme jako námi definovanou strukturu `SPRIM` (viz výpis kódu 5.1, ř. 1 – 4), která v sobě bude zahrnovat jak svoje označení, tak aktuální ohodnocení.

Protože do stromové struktury budeme potřebovat ukládat uzly jak pro jednotlivé logické operace, tak pro primární formule, musíme být schopni tyto jednotlivé typy uzlů od sebe odlišit, ale na druhou stranu s nimi musíme umět pracovat jako s jedním konstruktem. Definovali jsme proto novou strukturu `SNODE` tak, aby implementovala podobu zamýšleného uzlu (viz výpis kódu 5.1, ř. 6 – 11). Musí tedy obsahovat zejména typ uzlu a další položky, které těmto dvěma potřebným typům odpovídají. Podle znaku, který bude u každého uzlu uveden v jeho položce typ, vždy poznáme, o jaký typ uzlu se jedná a jak s ním můžeme dále pracovat.

Budeme-li vytvářet uzel pro logickou spojku, uvedeme do typu uzlu přímo symbol dané logické operace. V programu je definováno makro `isOperator()`, které vyhodnotí, zda symbol patří mezi logické spojky. Samo toto makro `isOperator()` využívá jiných dvou maker `isUnOperator()` a `isBinOperator()`, která rozpoznávají unární logické spojky (pro nás se týká pouze logické spojky pro negaci) a binární logické spojky (konjunkce, disjunkce, implikace a ekvivalence). Jedná-li se o unární spojku, přiřadíme do uzlu jako jeho levého potomka podformuli, kterou tato spojka neguje. Pokud by se jednalo o binární spojku, přiřadíme do uzlu jako jeho levého potomka podformuli, která bude vyhodnocována dříve, než podformule, kterou přiřadíme jako pravého potomka.

Budeme-li vytvářet uzel pro primární formuli, kterou rozeznáváme makrem `isOperand()`, uvedeme jako jeho typ symbol `'p'` a do příslušné položky přiřadíme ukazatel na dotčenou primární formuli. Do uzlu s primární formulí budeme dále vkládat do odpovídající položky ukazatel na danou primární formuli, kterou předem vytvoříme. Pokud už taková existuje, pak samozřejmě nevytváříme další, ale přiřadíme ukazatel na již existující, což nám do budoucna dovolí nastavovat hodnoty primárních formulí jen jednou pro každou jednotlivou z nich.

Alokaci dynamické paměti pro nově vznikající uzel včetně jeho inicializace jsme zabudovali do funkce `buildNode()` (viz výpis kódu 5.2). Funkce nemá kontrolní mechanismy, jestli předávané parametry splňují požadavky kladené na korektní uzel podle námi zadaných pravidel, ale jedná se o interní funkci uvnitř jednoho modulu, nepředpokládá se její využití mimo modul, není ani součástí veřejného rozhraní modulu a dovolíme si přenechat kontrolu korektnosti na volající funkci.

Naproti tomu bude třeba pečlivě kontrolovat vstupní řetězec, neboť uživatelský vstup nemusí být korektní a teprve až od funkce `parseInput()` požadujeme, aby s konečnou platností rozhodla, zda je uživatelský vstup řádnou výrokovou formulí podle námi nastavených pravidel, či nikoli. Některé typy nesprávných vstupů odhalí samotný Dijkstrům algoritmus seřaďovacího nádraží, který pro syntaktickou analýzu využijeme, ale pro některé typy je třeba využít dalších kontrolních mechanismů.

Vzhledem k tomu, že se v algoritmu seřaďovacího nádraží vlivem umisťování operátorů do čekací fronty může zčásti ztratit informace o následnosti jednotlivých znaků, vyhodnotí tento algoritmus jako korektní vstupy jak řetězec `dAB`, tak řetězec `AdB`, a to se shodným výstupem v podobě stromové struktury, přičemž pro nás je korektním řetězcem pouze druhý z uvedených. Provedeme proto v rámci funkce `parseInput()` kontrolu na dovolené sousední znaky, kterou implementujeme jako funkci `checkPrevious()` přebírající od funkce `parseInput()` znak právě načtený se znakem předchozím (který je třeba, aby si funkce `parseInput()` uchovávala v pomocné proměnné). Funkce `checkPrevious()` porovná předané dva znaky oproti vyjmenovaným zakázaným dvojicím z řad znaků, které jsou ve výrokové formuli tak, jak jsme ji navrhli, povolené. První porovnání nechává funkce `parseInput()` provést i po načtení prvního znaku textového řetězce, přičemž do předchůdce je umístěn pomocný znak `'\0'` a ve funkci `checkPrevious()` tak můžeme zkontrolovat i povolený první znak řetězce. Vrátí-li funkce `checkPrevious()` hodnotu `FALSE`, ukončí se i funkce `parseInput()` a vrátí obslužnému cyklu `NULL` indikující nesprávný uživatelský vstup.

Naopak Dijkstrův algoritmus vyhodnotí jako nekorektní vstup i určitý typ textových řetězců, který je v souladu s námi definovanou syntaxí. Jedná se o textové řetězce, které obsahují několik po sobě následujících symbolů pro operaci negace, tedy např. řetězec `nnA`. Při implementaci algoritmu tedy bude třeba upravit pravidla pro zpracování tohoto symbolu tak, aby syntaktický analyzátor odpovídal našim požadavkům.

Protože jsme uživateli povolili vkládat infixní tvar formule, který není striktně uzávorkovaný, a tedy jako takový neurčuje přesné pořadí vyhodnocování, ale na druhou stranu jsme stanovili přesně pravidla, jak bude vyhodnocení probíhat, musíme zajistit, aby tato pravidla byla dodržena. Definujeme proto každé z pěti logických spojek ve shodě s vybraným algoritmem její tzv. přednost. Nejvyšší přednost bude mít dle našich požadavků logická spojka negace, té přiřadíme hodnotu 5 a dále postupně snižujeme tuto hodnotu o jedna pro každou z logických spojek po řadě konjunkce, disjunkce, implikace a ekvivalence ve shodě s naším požadavkem a definovanou přednost konkrétní spojky si necháme vracet funkcí `getPrec()`.

Základem funkce `parseInput()` je cyklus `while`, který postupně načítá jednotlivé znaky uživatelského zadání a o každém takto načteném znaku se na základě pravidel určených Dijkstro-

vým algoritmem rozhoduje, jak má se znakem naložit. Jiné kroky následují, pokud se jedná o operand, jiné pro operátor či závorky, ošetřený je i výskyt mezery, jako znaku, který jsme v rámci textového zápisu povolili, a samozřejmě pokud je načten znak jiný než povolený, ukončíme funkci celou a do obslužného cyklu vrátíme hodnotu `NULL`. Současně namísto pouhého přesunu operandů a operátorů do konečného místa a jejich až následnému zpracování, provádíme jejich zpracování ihned, což nám umožní dříve rozpoznat nekorektní vstup.

Přesun do konečného místa je tedy implementován tak, že konečným místem rozumíme pole ukazatelů na strukturu `SNODE`. Pro znak odpovídající primární formuli pak přesun do tohoto pole znamemá, že pro něj vytváříme uzel odpovídající primární formuli s ukazatelem na danou primární formuli. Přehled o již vytvořených primárních formulích máme pomocí pomocného pole a před tím, než vytvoříme novou primární formuli jako proměnnou typu `SPRIM`, kontrolujeme, zda tato již neexistuje, případně do vytvářeného uzlu vložíme odkaz na tuto již existující namísto vytváření nové. Pro znak odpovídající unární logické spojce pak vytvoříme uzel, který jí odpovídá, do jeho levého potomka vložíme ukazatel na poslední uzel v konečném místě a vložíme jej právě na místo tohoto posledního. Analogicky postupujeme pro binární logickou spojku, s tím rozdílem, že posledního z konečného místa připojujeme do pravého potomka vytvářeného uzlu, předposledního z konečného místa připojujeme jako levého potomka, z konečného umístění odstraníme posledního i předposledního a nově vzniklý uzel vložíme na místo předposledního, kterého jsme před tím z tohoto místa odebrali.

Podle Dijkstrova algoritmu se před vložením operátoru do pomocného pole mají z vrcholu tohoto zásobníku odebrat (zpracovat) všechny operátory, které mají prioritu vyšší nebo rovnu prioritě vkládaného operátoru. Toto dodržíme až na výjimku, kterou je operátor pro logickou spojku negace. Pro tento operátor nastavíme pravidlo, že před jeho vložením do pomocného pole nebudeme odebírat z pomocného pole žádný jiný operátor. Pokud bychom ponechali pravidlo původní, odebíraly by se operátory s vyšší nebo stejnou prioritou. Žádný operátor nemá prioritu vyšší, tedy by docházelo k odebírání pouze shodných operátorů negace. Nicméně takto se vyhneme problému se syntaktickou analýzou textových řetězců, které obsahují několik negací za sebou.

Pro názornost uvažujme textový řetězec `nnA`. První z načtených symbolů je `n`, tedy operátor, který je vložen do pomocného pole. Druhým je rovněž symbol `n`. Pokud ponecháme původní předpis, máme podle algoritmu z pomocného zásobníku vyjmout (a zpracovat) operátor negace z předchozího kroku. Protože ale konečný zásobník je dosud prázdný (symbol `A` pro primární formuli nebyl dosud ze vstupu přečten), nemá tato spojka co zpracovat a algoritmus končí s chybným vyhodnocením textového řetězce jako nekorektního vstupu. Pravidlo o neodebírání operátorů z pomocného pole při načtení operátoru negace tento problém odstraní, aniž by pozměnilo chování algoritmu v jiných bodech, a tedy syntaktická analýza proběhne podle našich požadavků.

Po úspěšném průchodu celého textového řetězce jsou implementovány i následné kroky podle Dijkstrova algoritmu, tedy přesun zbývajících čekajících operátorů ke zpracování. Je-li kdekoli v průběhu algoritmu zjištěna nekorektnost vstupního řetězce, funkce `parseInput()` zavolá pomocnou proceduru `clearMemory()`, která slouží k zajištění uvolňování dynamicky alokované paměti a kterou jsme implementovali pro zpřehlednění kódu, neboť míst, ve kterých může algoritmus stanovit nekorektnost vstupu, je více a docházelo by k neúměrnému opakování téhož. V opačném případě je do obslužného cyklu předán ukazatel na první uzel v konečném umístění, protože jiný uzel už toto umístění ani nesmí obsahovat, neboť by to znamenalo, že máme po ukončení algoritmu více podstromů, které se nespojí dalšími spojkami, a jedná se tedy o nekorektní vstup. Předávaný uzel je kořenem hledané stromové struktury, který obsahuje jako své další vnitřní uzly jednotlivé logické spojky a jako listy obsahuje uzly odpovídající primárním formulím.

Výpis kódu 5.3 Rozhraní modulu table

```
typedef struct STruthTable {
    int cntPrims;
    int cntRows;
    char* header;
    int** table;
} STRUTHTABLE;

STRUTHTABLE* buildTruthTable(SNODE* tree);
void printTruthTable(STRUTHTABLE* truthTable);
void freeTruthTable(STRUTHTABLE* truthTable);
```

5.3 Nalezení úplných normálních tvarů

Případ užití *UC2 – Nalezení úplných normálních tvarů* jsme implementovali do příslušného modulu s hlavičkovým souborem `table.h` a zdrojovým souborem `table.c`. Modul je využit obslužným cyklem k vytvoření pravdivostní tabulky, která odpovídá výrokové formuli, kterou mu obslužný cyklus po jejím obdržení z modulu `formula` předá, tedy ve formě stromové struktury. Obslužný cyklus pak dále na základě uživatelského požadavku rozhoduje, zda bude pravdivostní tabulka vypsána na standardní výstup (pro případ použití přepínače `-t`) či ji pouze předá k dalšímu zpracování. Aby se obslužný cyklus nemusel sám starat o konkrétní implementaci pravdivostní tabulky, přidáme do rozhraní tohoto modulu `table` (viz výpis kódu 5.3) k funkci `buildTruthTable()` zajišťující vytvoření pravdivostní tabulky ještě proceduru `printTruthTable()` sloužící k vypsání pravdivostní tabulky na standardní výstup a proceduru `freeTruthTable()`, pomocí které dochází k uvolňování dynamicky alokované paměti pro konkrétní pravdivostní tabulku.

Pravdivostní tabulku jsme implementovali jako novou strukturu `STRUTHTABLE` (viz výpis kódu 5.3), která uchovává informace o počtu prvotních formulí a počtu řádků vlastní pravdivostní tabulky. Dále struktura `STRUTHTABLE` nese záhlaví pravdivostní tabulky, tedy seznam prvotních formulí, který jsme implementovali jako znakové pole o velikosti počtu prvotních formulí, kam vypisujeme názvy prvotních formulí. Nakonec struktura obsahuje i pravdivostní tabulku jako takovou, tedy dvourozměrné pole celočíselných hodnot, které má 2^n řádků a počet sloupců je o jedna větší, než je počet prvotních formulí. První sloupce, jejichž počet odpovídá počtu prvotních formulí, využijeme k zápisu všech možných kombinací ohodnocení prvotních formulí a poslední sloupec pak bude sloužit k zápisu pravdivostní hodnoty analyzované výrokové formule pro kombinaci pravdivostního ohodnocení prvotních formulí daného řádku.

Jak jsme již uvedli, stěžejní část implementace případu užití *UC2 – Nalezení úplných normálních tvarů* je implementována ve funkci `buildTruthTable()`, kterou využíváme v programu tak, že ji volá obslužný cyklus, předává jí ukazatel na kořen stromové struktury zachycující výrokovou formuli a očekává od ní jako výstup pravdivostní tabulku v námi definované struktuře. Funkce `buildTruthTable()` proto začne tak, že vytvoří ukazatel na proměnnou struktury `STRUTHTABLE` a dynamicky pro ni alokuje paměť.

Pravdivostní tabulku vytvoří funkce `buildTruthTable()` tak, že si nejprve nechá pomocí funkce `findPrims()` definované ještě v modulu `formula` shromáždit všechny primární formule použité ve stromové struktuře. Navíc funkce `findPrims()` vrací jako výstupní hodnotu počet těchto takto nalezených primárních formulí, což je údaj, který necháme vepsat do proměnné odpovídající pravdivostní tabulce do příslušné položky struktury `STRUTHTABLE`. To je údaj, který nám umožní alokovat potřebnou paměť jednak pro znakové pole v další položce struktury, kam

budeme ukládat názvy primárních formulí, a jednak pro vlastní pravdivostní tabulku jako další položku této struktury, jejíž počet sloupců a řádků se od této hodnoty rovněž odvíjí. Počet řádků určíme na základě počtu sloupců, a to jen jednou, a zjištěnou hodnotu rovněž uložíme do příslušné proměnné v rámci struktury `STRUTHTABLE`.

Přiřazení prvotních formulí ze záhlaví ke sloupcům vlastní pravdivostní tabulky implementujeme jako pole o velikosti počtu prvotních formulí, kde každému prvku přiřadíme právě jednu prvotní formuli. Index tohoto pole pak bude implikovat index sloupce v pravdivostní tabulce. Naplnění tohoto pole – záhlaví následně provedeme pomocí pole ukazatelů na primární formule, které funkce `findPrims()` po svém zavolání naplnila. Tato funkce je implementována tak, že naplňuje pole o velikosti počtu všech možných primárních formulí, tedy 26, a to tak, že do prvního indexu zařadí ukazatel na prvotní formuli s označením A – pokud existuje, do druhého pole zařadí ukazatel na primární formuli s názvem B – opět pouze v případě, že tato primární formule existuje, atd. V případě, že se formule určitého označení ve stromové struktuře nikde neobjeví, zůstává v poli na místě příslušného indexu ukazatel `NULL`. Takto implementované řešení, kdy záhlaví tabulky plníme postupným průchodem takto naplněného pole ukazateli na primární formule, nám jako záhlaví pravdivostní tabulky dává záhlaví abecedně seřazené dle označení ve výrokové formuli použitých prvotních formulí.

Po alokaci paměti pro vlastní pravdivostní tabulku necháme funkci `buildTruthTable()` provést její předvyplnění, tedy vyplnění jejích sloupců pro jednotlivé prvotní formule tak, aby tabulka obsahovala všechny možné kombinace ohodnocení prvotních formulí, které následně budeme po řádcích chápat jako ohodnocení celé formule. Po předvyplnění tabulky přistoupíme k postupnému průchodu všech řádků – kombinací ohodnocení prvotních formulí a necháme podle této kombinace zjistit, jaké ohodnocení pro ně nabývá výroková formule jako celek, a výsledek zapíšeme na týž řádek do posledního sloupce.

Pro vlastní vyhodnocení výrokové formule využijeme funkci `getValue()`, kterou jsme rovněž připravili ještě v modulu `formula`. Předáváme jí výrokovou formuli – stromovou strukturu, kterou předem přizpůsobujeme, aby ohodnocení jejích prvotních formulí – listů odpovídalo ohodnocení primárních formulí v daném řádku, který právě zpracováváme. K nastavování pravdivostního ohodnocení jednotlivých formulí ještě jednou využijeme pole ukazatelů na primární formule, které jsme obdrželi při volání funkce `findPrims()`, a to tak, že při průchodu řádkem přeneseme každou z těchto hodnot do pravdivostního ohodnocení příslušné primární formule, na kterou máme ukazatel v poli. Po nastavení pravdivostního ohodnocení každé z primárních formulí hodnotami z daného řádku už jen necháme funkci `getValue()`, aby vrátila pravdivostní hodnotu celé formule. Přitom funkce `getValue()` pracuje jednoduše na principu rekurzivního průchodu stromem a postupně vyhodnocuje jednotlivé podstromy podle pravidel 3.1 až ke kořeni, přičemž výsledná hodnota je hledanou pravdivostní hodnotou celé výrokové formule.

Zcela vyplněnou pravdivostní tabulku, kterou obslužný cyklus obdrží, necháme pro případ přepínače `-t` vypsat na standardní výstup, a to včetně záhlaví, aby bylo zřejmé, jaké pravdivostní ohodnocení prvotních formulí vedlo k danému výsledku. V tabulce jsou uvedeny jak disjunktivní, tak konjunktivní úplné normální tvary, které jsme chtěli v případu užití *UC2 – Nalezení úplných normálních tvarů* získat. Disjunktivní úplný normální tvar je z tabulky odvoditelný jako disjunkce konjunkcí primárních formulí, resp. jejich negací pro řádky, které mají v posledním sloupci hodnotu 1, a analogicky pak konjunktivní úplný normální tvar je z tabulky odvoditelný jako konjunkce disjunkcí primárních formulí, resp. jejich negací pro řádky, které mají v posledním sloupci hodnotu 0 (podrobněji popsáno v kapitole Analýza v úvodu podkapitoly Algoritmus Quine–McCluskey (3.3)). Případně tento výstup z funkce `parseInput()` bude sloužit jako vstup do dalšího modulu, který provádí minimalizaci dané výrokové formule.

Výpis kódu 5.4 Rozhraní modulu minform

```
1  typedef struct SStringChain {
2      char* string;
3      struct SStringChain* next;
4  } SSTRINGCHAIN;
5
6  typedef struct SMinNF{
7      char type;
8      char variant;
9      int cntNFs;
10     SSTRINGCHAIN** normalForms;
11 } SMINNF;
12
13 SMINNF* minimize(STRUTHTABLE* truthTable, char variant);
14 void renderOneMinNF(SMINNF* minNFs);
15 void renderAllMinNFs(SMINNF* minNFs);
16 void freeMinNF(SMINNF* minNF);
```

5.4 Vlastní minimalizace

V rámci vlastní minimalizace budeme implementovat v příslušném modulu `minform` dříve popsaný algoritmus Quine–McCluskey. Vstupem do této části je pravdivostní tabulka, kterou prostřednictvím obslužného cyklu funkce `main()` přebíráme z modulu `table` a která tak, jak jsme vytvořili strukturu `STRUTHTABLE`, která ji představuje, s sebou nese veškeré potřebné informace. Obslužný cyklus bude pro vlastní minimalizaci spouštět z modulu `minform` funkci `minimize()`, od které očekává jako návratovou hodnotu strukturu, která odpovídá všem nalezeným minimálním normálním tvarům hledané varianty (disjunktivní či konjunktivní) minimalizované výrokové formule, jejíž pravdivostní tabulku předává v parametrech. Tuto strukturu, kterou jsme nazvali `SMINNF` (viz výpis kódu 5.4, ř. 6 – 11), odpovídající popsané entitě zavádíme opět z důvodu zjednodušení práce s takovýmto výsledkem.

Struktura `SMINNF` nese v prvé řadě informace o tom, zda se případně jedná o speciální formy výrokové formule, jakými jsou tautologie a kontradikce. V takovém případě ponese struktura v příslušné položce `type` informaci `'t'`, resp.`'c'`, v opačném případě se do položky uvede znak `'r'`, který signalizuje regulární výrokovou formuli, u které byla minimalizace, tedy hledání minimálních normálních tvarů, provedena. Další potřebný odlišující znak zavádíme pro odlišení toho, zda se jednalo o hledání minimálních normálních tvarů disjunktivních či konjuktivních, což bude mít pro naši implementaci vliv na výpis zmíněného minimálního normálního tvaru. Dále pak struktura obsahuje informaci o tom, kolik minimálních normálních tvarů bylo nalezeno a samozřejmě i tyto minimální normální tvary jako takové.

Pro práci s nalezenými minimálními normálními tvary jsme zavedli opět strukturu, která odpovídá našim potřebám, a to strukturu `SSTRINGCHAIN` (viz výpis kódu 5.4, ř. 1 – 4). Minimální normální tvar bez ohledu na to, zda se jedná o disjunktivní či konjunktivní variantu, budeme zapisovat jako zřetězení textových řetězců, které vzniknou vynecháním konjunktivních, resp. disjunktivních spojek uvnitř každého implikantu, resp. klausule. Navíc budeme vynechávat i případné spojky negující jednotlivé primární formule a tuto informaci o znegování příslušné primární formule uchováme tak, že namísto velkého písmena pro název primární formule uvedeme do řetězce písmeno malé.

Veřejné rozhraní modulu `minform` (viz výpis kódu 5.4) obsahuje kromě shora popsaných struktur `SSTRINGCHAIN` a `SMINNF` a funkce `minimize()` ještě proceduru nazvanou `renderOneMinNF()`,

Výpis kódu 5.5 Funkce minimize()

```
SMINNF* minimize(STRUTHTABLE* truthTable, char variant) {
    int cntOfMints;
    SROWCHAIN* auxTable = NULL;
    SSTRINGCHAIN** primarySet = NULL;
    SMINNF* minNF = NULL;
    if ((minNF = (SMINNF*) calloc(1, sizeof (SMINNF))) == NULL) {
        err(1, NULL);
    }
    minNF->variant = variant;
    if (isTautology(truthTable) || isContradiction(truthTable)) {
        minNF->cntNFs = 0;
        minNF->normalForms = NULL;
        if (isTautology(truthTable)) minNF->type = 't';
        else minNF->type = 'c';
        return minNF;
    }
    minNF->type = 'r';
    auxTable = buildAuxTable(truthTable, variant);
    cntOfMints = countMints(auxTable);
    primarySet = buildPrimarySet(auxTable, truthTable->cntPrims);
    doFirstPhase(auxTable, primarySet, truthTable->cntPrims,
                        cntOfMints);
    freeSetOfStringChains(primarySet, truthTable->cntPrims + 1);
    doSecondPhase(auxTable, cntOfMints, minNF);
    return minNF;
}
```

Výpis kódu 5.6 Struktura SRowChain

```
typedef struct SRowChain {
    char* mint;
    int* values;
    int used;
    struct SRowChain* next;
} SROWCHAIN;
```

Výpis kódu 5.7 Funkce buildPrimarySet()

```
SSTRINGCHAIN** buildPrimarySet(SROWCHAIN* auxTable, int cntOfPrims) {
    SSTRINGCHAIN** setOfMints = NULL;
    SROWCHAIN* rowPointer = auxTable;
    setOfMints = (SSTRINGCHAIN**) malloc((cntOfPrims + 1)
                                * sizeof (SSTRINGCHAIN*));
    for (int i = 0; i < cntOfPrims + 1; i++) setOfMints[i] = NULL;
    while (rowPointer != NULL) {
        addMint(rowPointer->mint, setOfMints);
        rowPointer = rowPointer->next;
    }
    return setOfMints;
};
```

která zajišťuje výpis jednoho z nalezených minimálních normálních tvarů, současně také proceduru `renderAllMinNFs()`, která zajišťuje výpis všech nalezených minimálních normálních tvarů, a proceduru `freeMinNF()`, která zajišťuje uvolnění paměti pro strukturu `SMINNF`. Všechny tyto tři procedury jsou ve veřejném rozhraní z toho důvodu, že je používá obslužný cyklus, aby splnil uživatelské zadání, a to aniž by se staral o způsob implementace, který modul `minform` obsahuje.

Pokud obslužný cyklus volá z modulu `minform` funkci `minimize()`, předává jí jako parametry příslušnou pravdivostní tabulku a informaci o tom, zda má funkce hledat disjunktivní či konjunktivní normální tvar. Právě druhý parametr, který do funkce `minimize()` předáváme, určuje, zda se má provést hledání minimálního DNT nebo minimálního KNT. Obslužný cyklus totiž může tuto funkci `minimize()` volat právě pro tyto dva rozdílné případy, a to jednou s parametrem `'d'` a jednou s parametrem `'c'`. Jak bylo dříve popsáno, hledání minimálního DNT a hledání minimálního KNT se od sebe odlišuje pouze ve dvou místech, a to na začátku, kdy máme v případě hledání minimálního KNT dané formule vzít negaci této formule, a na konci, kdy provedeme na rozdíl od minimálního DNT záměnu literálů za jejich negaci a záměnu operátorů pro disjunkci a konjunkci.

Rozdíl na začátku, tedy vzít namísto původní formule její negaci, implementujeme tak, že z pravdivostní tabulky nebudeme vybírat řádky, pro které je původní formule pravdivá, ale právě ty, pro které je původní formule nepravdivá (existenci alespoň jednoho ohodnocení formule, pro které je formule pravdivá, a alespoň jednoho ohodnocení formule, pro které je formule nepravdivá, nám zaručuje to, že do vlastní minimalizace nevstupují tautologie ani kontradikce, kteréžto případy funkce `minimize()` ošetří ještě před dalším zpracováním - viz výpis kódu 5.5, ř. 10 – 16). Nemusíme tedy provádět negaci formule jako takovou, ale pro oba případy využijeme tu samou pravdivostní tabulku, kterou tak máme předvypočítanou právě jednou pro obě varianty.

Rozdíl na konci vyřešíme až při výpisu výstupu, tedy pro minimální KNT nebudeme vypisovat literál, který je ve výstupu, ale jeho negaci, a oproti výpisu minimálního DNT zaměníme označení operace disjunkce a konjunkce. I pro tento případ si v rámci struktury odpovídající entitě nalezených minimálních normálních tvarů ponecháváme informaci o tom, zda bylo prováděno hledání minimálních normálních tvarů disjunktivních či konjunktivních, a to i přesto, že obslužný cyklus tuto informaci má k dispozici, ale chceme, aby byl modul použitelný i pro případné jiné programy.

Ve shodě s postupem naznačeným v algoritmu Quine–McCluskey si dále musíme připravit pomocnou tabulku, tedy tabulku, která bude mít ve svých sloupcích jednotlivé mintermy, které jsme převzali z pravdivostní tabulky. Prvních řádků bude takový počet, kolik mintermů jsme převzali. K této tabulce pak budeme přidávat další a další řádky podle toho, jak budeme postupně jednotlivé mintermy slučovat a jak budeme slučovat implikanty vzniklé předchozím slučováním. Navíc si budeme potřebovat pamatovat, jaký z těchto řádků jsme ke sloučení použili, a samozřejmě u každého řádku potřebujeme vědět, jaký z původních mintermů implikuje. Největší nevýhoda této tabulky je taková, že předem nedokážeme určit, kolik budeme potřebovat řádků, a alokovat takovouto tabulku pro všechny možné kombinace by vedlo na tabulku s velkým podílem prázdných řádků.

Implementujeme proto tuto tabulku jako spojový seznam, který v každém svém článku obsahuje údaje potřebné k jednomu řádku a odkaz na článek další. Tato implementace nás vede k vytvoření další struktury `SROWCHAIN` (viz výpis kodu 5.6). Dotyčný minterm, resp. implikant, ke kterému se řádek váže, budeme zapisovat jako řetězec znaků (viz výpis kódu 5.6, ř. 2). Vyplnění jednotlivých sloupců, tedy zda řádek implikuje některý ze sloupců, budeme zapisovat do pole čísel (viz výpis kódu 5.6, ř. 3), přičemž 0 bude znamenat, že sloupec implikován není, jiná hodnota bude znamenat, že implikace je přítomna.

Počet sloupců budeme znát (je roven počtu původních mintermů) a při tvorbě každého článku řetězce si podle tohoto počtu zaalokujeme příslušnou délku pole, do kterého pak budou údaje o implikaci zapisovány. Součástí struktury bude i údaj o tom, zda daný článek řetězce, tedy řádek pomocné tabulky, byl použit ke sloučení či nikoli (viz výpis kódu 5.6, ř. 4), přičemž hodnota 0 bude znamenat, že využit nebyl, jakákoli jiná hodnota pak, že použit byl.

Pro účely pomocné tabulky si opět interně zjednodušíme zápis mintermů/implikantů, a to obdobně, jak je zapisujeme ve výsledné struktuře `SMINNF`. Binárními operátory v těchto mintermech/implikantech jsou totiž opět pouze konjunkce, tedy tyto spojky nebudeme zaznamenávat vůbec – budeme je vynechávat bez jakékoli náhrady (nebudeme připouštět ani mezery). Vzhledem k tomu, že primární formule mohou být označeny pouze velkými písmeny anglické abecedy, budeme v rámci pomocné tabulky označovat jejich negaci příslušnými malými písmeny anglické abecedy – bez uvedení operátoru negace `n`. Pro tento proces využijeme z knihovny `ctype.h` funkci `tolower()` (opačný proces, tedy převedení malého písmene na příslušné velké písmeno, zajistíme při případném zpětném převodu inverzní funkcí `toupper()`).

Pomocnou tabulku v jejím základním nastavení, tedy po zadání dat z pravdivostní tabulky, připravíme v rámci funkce `minimize()` prostřednictvím funkce `buildAuxTable()` (viz výpis kódu 5.5, ř. 18). Vybereme z pravdivostní tabulky příslušné řádky (podle varianty, kterou hledáme – buď minimální DNT, nebo minimální KNT). Pro každý z nich vytvoříme článek řetězce, do kterého vložíme zjednodušeně zapsaný minterm (velké písmeno pro primární formuli, která má ohodnocení 1, a malé písmeno pro primární formuli, která má ohodnocení 0 – vysvětlení viz výše) a nastavíme příznak použití na nulu. Během přidávání řádků si napočítáváme jejich počet.

Po přidání posledního řádku tak známe rovnou počet sloupců, který máme v každém řádku alokovat jako pole celočíselných hodnot. Projdeme tedy celou pomocnou tabulkou ještě jednou a v každém řádku zaalokujeme pole o této velikosti a rovnou ji doplníme o hodnoty – všude budou nulové hodnoty, kromě hlavní diagonály, na kterou doplníme číslo jedna, tedy obecně v n-tém řádku bude hodnota 1 v poli pouze na místě n. Takto vytvořená pomocná tabulka pak bude sloužit jako vstup do první fáze vlastní minimalizace (viz výpis kódu 5.5, ř. 21 – 22), kde se doplní o další řádky podle seskupování, a s tímto rozšířením pak bude vstupovat do druhé závěrečné fáze vlastní minimalizace (viz výpis kódu 5.5, ř. 24).

Současně ještě jako další vstup do první fáze vytvoříme v rámci funkce `minimize()` prvotní rozčlenění mintermů do skupin podle počtu v nich obsažených negací primárních formulí. Za tím účelem implementujeme funkci `buildPrimarySet()`. Této funkci předáme naši pomocnou tabulku, ze které může přečíst označení mintermů, a také počet literálů, resp. znaků v těchto mintermech, což je v tomto bodě postupu také rovno počtu primárních formulí. Požadujeme pak, aby tato funkce vrátila sadu skupin (ve shodě s tím, co jsme uvedli během popisu algoritmu Quine–McCluskey, jich má být o jedna více, než je počet literálů), v nichž budou seskupeny mintermy se shodným počtem negací primárních formulí (pro naši implementaci zjednodušeného označování literálů je to shodné s počtem malých písmen).

V tomto případě známe počet skupin, nikoli však počty mintermů v jednotlivých skupinách (některé z nich mohou být i prázdné). Můžeme tedy opětovně využít spojového seznamu z určité struktury, kterou jsme již zavedli pro účely struktury `SMINNF`, a to strukturu `SSTRINGCHAIN` (viz výpis kódu 5.4). Ve struktuře nebudeme potřebovat nic jiného, než je řetězec znaků, do kterého poznamenáme daný minterm, resp. při další práci obecně implikant, a odkaz na další článek spojového seznamu. Takovýchto spojových seznamů vytvoříme v rámci sady prostřednictvím funkce `buildPrimarySet()` (viz výpis kódu 5.7) o jedna víc, než je počet primárních formulí (viz výpis kódu 5.7, ř. 4), a do každého z nich pak postupně přidáme jeden po druhém mintermy z pomocné tabulky (viz výpis kódu 5.7, ř. 8) pomocí funkce `addMint()` jako články samostatných spojových seznamů, přičemž funkce `addMint()` pracuje jednoduše tak, že spočítá počet negací

Výpis kódu 5.8 Procedura doFirstPhase()

```
1  void doFirstPhase(SROWCHAIN* auxTable, SSTRINGCHAIN** oldSet,
2          int cntOfChars, int cntOfMints) {
3      int flag = FALSE;
4      SSTRINGCHAIN** newSet = NULL;
5      if ((newSet = (SSTRINGCHAIN**) calloc(cntOfChars + 1,
6              sizeof (SSTRINGCHAIN*))) == NULL) {
7          err(1, NULL);
8      }
9      for (int i = 0; i < cntOfChars; i++) {
10         newSet[i] = mergeBags(oldSet[i], oldSet[i + 1], auxTable,
11                 cntOfChars, cntOfMints);
12         if (newSet[i] != NULL) {
13             flag = TRUE;
14         }
15     }
16     if (flag && (cntOfChars > 1)) {
17         doFirstPhase(auxTable, newSet, cntOfChars - 1, cntOfMints);
18     }
19     freeSetOfStringChains(newSet, cntOfChars);
20     return;
21 }
```

primárních formulí v předaném mintermu, vytvoří pro něj článek řetězce a zařadí jej do spojového seznamu toho řetězce, který v rámci sady odpovídá právě zjištěnému počtu negací primárních formulí.

5.4.1 První fáze vlastní minimalizace

První fázi vlastní minimalizace v naší implementaci představuje procedura `doFirstPhase()` (viz výpis kódu 5.8), do které vstupují sada mintermů rozčleněných dle počtu negovaných primárních formulí do různých skupin a pomocná tabulka s jednotlivými řádky ke každému mintermu, a to včetně doplňujících informací k sadě týkající se délky označení mintermu a k pomocné tabulce týkající se počtu sloupců.

Hlavní úkol procedury `doFirstPhase()` je zajistit slučování implikantů (kterými jsou po zavolání z funkce `minimize()` původní mintermy a v následujích voláních implikanty, které z těchto původních mintermů vznikly v rámci sloučení), tedy upravovat pomocnou tabulku, která je jí předávána jako parametr, tak, že ji doplňuje o nové řádky a aktualizuje ty stávající podle postupu slučování uvedeného v popisu algoritmu Quine–McCluskey. Tuto funkci jsme implementovali tak, že po každém provedeném slučování rozhodne, zda má být provedeno slučování další, a má-li tomu tak být, pak zavolá samu sebe s upravenými parametry.

V proceduře `doFirstPhase()` si nejprve připravíme novou sadu skupin (viz výpis kódu 5.8, ř. 5). Skupin bude v každé další iteraci slučování o jedna méně, než v iteraci předchozí. Potom v rámci procedury `doFirstPhase()` postupně procházíme všechny sousední dvojice skupin a za každou dvojici naplníme v nové sadě jednu skupinu (viz výpis kódu 5.8, ř. 10). Porovnávání pouze sousedních skupin nám umožňuje algoritmus Quine–McCluskey a také to, jakým způsobem jsme implementovali sestavení prvotní sady – v každém poli jsou mintermy s takovým počtem negovaných primárních formulí, který odpovídá indexu pole. Abychom i nadále udrželi tento

stav, ukládáme implikant ze sloučené dvojice vždy do pole s indexem, který odpovídá nižšímu indexu ze slučovaných implikantů v rámci původní sady.

Po vytvoření nové sady musíme zkontrolovat, zda má být zavoláno další slučování. Podle popisu algoritmu Quine–McCluskey má k dalšímu kolu slučování dojít, pokud v tom aktuálním vůbec k nějakému slučování došlo. Z toho důvodu jsme si zavedli v proceduře `doFirstPhase()` proměnnou `flag` (viz výpis kódu 5.8, ř. 3), která nám bude sloužit jako zarážka – před provedením prvního pokusu o sloučení v rámci nově zavolané procedury `doFirstPhase()` nastavíme její hodnotu na nějakou konkrétní hodnotu (vybrali jsme si hodnotu FALSE) a pokud v rámci pokusů o sloučení k nějakému skutečně došlo, změníme její hodnotu (v našem případě měníme na TRUE). Pokud dojde k druhému a dalšímu slučování, ponecháváme hodnotu stále na TRUE, čímž po ukončení procesu snah o slučování víme, zda došlo alespoň k jednomu sloučení či nikoli, a podle této hodnoty tedy voláme či nevoláme další slučování.

Současně také při rozhodování o znovuzavolání procedury `doFirstPhase()` kontrolujeme, zda by slučování mělo smysl. V případě, že už bychom volali slučování implikantů o nulovém počtu literálů, pak takové volání realizovat nebudeme.

O provedení slučování dvou konkrétních skupin dané sady se nám stará funkce `mergeBags()`. Tato funkce v prvé řadě zkontroluje, zda obě skupiny obsahují každá alespoň jeden implikant. V opačném případě se ukončí, protože slučování skupin, z nichž alespoň jedna je prázdná, by nemělo smysl. Následně prochází všechny možné dvojice implikantů z těchto dvou skupin a zjišťuje, zda jdou sloučit, či nikoliv.

Toto slučování implikantů jsme implementovali do funkce `mergeStrings()`, která vrací nový řetězec – pro případ, že implikanty sloučit lze a vzniká nový imlikant, který ještě v pomocné tabulce není uveden, či prázdný řetězec – pro případ, kdy implikanty podle podmínek stanovených v algoritmu Quine–McCluskey sloučit nelze nebo již v pomocné tabulce existuje.

Pro připomenutí – sloučit lze implikanty, které se liší v jednom literálu, a to tak, že jeden implikant tento literál obsahuje jako primární formuli a druhý implikant ho obsahuje jako negaci této primární formule. Řetězec navrácený funkcí `mergeStrings()` je právě ten výsledný řetězec, který vznikne sloučením, tj. „vynecháním“ literálu, ve kterém se původní dva implikanty odlišují. Funkce `mergeBags()` pak tento řetězec vezme a vytvoří z něj článek do řetězce implikantů, který po ukončení předává jako celý řetěz do volající procedury `doFirstPhase()`.

Funkce `mergeStrings()` se v rámci procesu slučování stará rovněž o aktualizaci pomocné tabulky. Poté, co zjistí, že má ke sloučení dojít, a má k dispozici i nový implikant, projde pomocnou tabulkou a u řádků, které patří původním implikantům, zajistí změnu příznaku na použitý implikant. Současně projde pomocnou tabulkou a pokud tam výsledek sloučení ještě není uveden, přidá tam nový řádek, do označení uvede zjištěný výsledek, do příznaku použitosti uvede symbol pro nepoužitý implikant a pole odpovídající implikaci jednotlivých původních mintermů nechá funkcí `addValues()` pro každý index nastavit tak, že sečte hodnoty v odpovídajících indexech u původních implikantů, ze kterých tento nový implikant vzešel.

Postupným slučováním, tedy cyklickým voláním procedury `doFirstPhase()`, měníme pomocnou tabulku, takže po ukončení slučování máme tabulku, ve které k původním řádkům obecně přibyly řádky další, víme, který řádek byl ke sloučení použit a který řádek implikuje jaké původní mintermy. Máme tedy připravenou pomocnou tabulku tak, jak ji potřebujeme do druhé fáze vlastní minimalizace.

Výpis kódu 5.9 Procedura doSecondPhase()

```
1  void doSecondPhase(SROWCHAIN* auxTable, int cntOfMints, SMINNF* minNF) {
2      SROWCHAIN* cutAuxTable = NULL;
3      SROWCHAIN* delTable = NULL;
4      int cntOfFound = 0;
5      SINTSCHAIN* combinationsOfMinterms = NULL;
6      int* cntNFs = NULL;
7      if ((cntNFs = (int*) calloc(1, sizeof (int))) == NULL) {
8          err(1, NULL);
9      }
10     cutAuxTable = cutOutUsed(auxTable);
11     cntOfFound = countFound(cutAuxTable);
12     combinationsOfMinterms = developCombinations(cutAuxTable,
13             cntOfMints, cntOfFound);
14     combinationsOfMinterms = removeLong(combinationsOfMinterms,
15             cntOfFound, cutAuxTable);
16     minNF->normalForms = buildAllMinNFs(combinationsOfMinterms,
17             cntOfFound, cutAuxTable, cntNFs);
18     minNF->cntNFs = (*cntNFs);
19     freeIntsChain(combinationsOfMinterms);
20     free(cntNFs);
21     delTable = cutAuxTable;
22     while (cutAuxTable != NULL) {
23         cutAuxTable = cutAuxTable->next;
24         free(delTable->mint);
25         free(delTable->values);
26         free(delTable);
27         delTable = cutAuxTable;
28     }
29     return;
30 }
```

5.4.2 Druhá fáze vlastní minimalizace

Druhou fázi vlastní minimalizace v naší implementaci představuje procedura, kterou jsme nazvali `doSecondPhase()` (viz výpis kódu 5.9). Jejím hlavním vstupním parametrem je pomocná tabulka, jejíž konečnou podobu jsme dotvořili v předcházející první fázi. K tomu, abychom mohli s touto tabulkou bezpečně pracovat, předáváme jako další parametr počet mintermů v původní pomocné tabulce, tedy délku pole, které obsahuje informace o implikaci těchto mintermů. Dále ještě předáváme procedurou `doFirstPhase()` již částečně vyplněnou proměnnou typu struktury `SMINNF` odpovídající vznikajícímu přehledu o minimálních normálních tvarech, která bude po dokončení této druhé fáze vlastní minimalizace ve své konečné podobě.

Do druhé fáze minimalizace potřebujeme podle popisu algoritmu Quine–McCluskey z pomocné tabulky vlastně pouze řádky, které nebyly použity ke sloučení, ale do druhé fáze předáváme celou pomocnou tabulku, a výběr řádků relevantních pro druhou fázi vlastní minimalizace tedy provedeme až zde. Výběr těchto relevantních řádků, tedy řádků, které nebyly použity ke sloučení, provedeme na základě jejich příznaků v příslušné položce, a to pomocí funkce `cutOutUsed()` (viz výpis kódu 5.9, ř. 10).

Funkce `cutOutUsed()` neudělá nic jiného, než že vrátí ukazatel na nově vzniklou ořízlou pomocnou tabulku, kterou vyrobí tak, že najde v původní pomocné tabulce první řádek, který nebyl použit, a připojuje za něj všechny další řádky, které nebyly použity, poté, co přeskočí všechny ty, které použity byly. Počet řádků, které obsahuje tato nová oříznutá pomocná tabulka, si spočítáme pomocí funkce `countFound()` (viz výpis kódu 5.9, ř. 11) a uložíme do proměnné, abychom ji mohli dále využívat, aniž bychom ji vypočítávali několikrát znovu.

Ve druhé fázi vlastní minimalizace budeme tedy nejprve hledat vhodné kombinace z nalezených implikantů (viz výpis kódu 5.9, ř. 12) – těch, které zůstaly v oříznuté pomocné tabulce, tak, aby dohromady pokryly všechny původní mintermy, a následně z těchto kombinací vybereme ty nejkratší (viz výpis kódu 5.9, ř. 14). Nakonec pak z nalezených kombinací necháme vytvořit hledané minimální normální tvary (viz výpis kódu 5.9, ř. 16) a přiřadíme je do předaného parametru, který odpovídá přehledu minimálních normálních tvarů, a to včetně informace o jejich počtu.

Nalezení všech možných kombinací implikantů, které dohromady pokryjí všechny původní mintermy, implementujeme do funkce `developCombinations()`. Protože předem nevíme, kolik nalezneme kombinací, využijeme opět konstruktu spojového seznamu a necháme si vrátit do procedury `doSecondPhase()` od funkce `developCombinations()` tyto kombinace zakódované pomocí zřetězení polí celočíselných hodnot. Jedno pole bude odpovídat jedné konkrétní kombinaci a budeme je chápat jako jeden řádek v pomocné tabulce, přičemž každé pole bude mít vždy tolik hodnot, kolik bylo nalezených minimálních implikantů, a pokud do kombinace bude příslušný implikant patřit, tak bude mít v příslušném indexu pole hodnotu nenulovou, v opačném případě hodnotu nulovou.

Kombinace vytvoříme podle algoritmu Quine–McCluskey tak, že máme pro každý původní minterm vytvořit obdobu klausule, což odpovídá tomu, že původní minterm je implikován jedním z nesloučených implikantů (tedy implikantů, které zůstaly v oříznuté pomocné tabulce) nebo druhým nesloučeným implikantem a tak dále. Vybíráme samozřejmě pouze ty implikanty, které daný původní minterm implikují, tedy v oříznuté tabulce bude v příslušném indexu nesloučeného implikantu nenulová hodnota.

Z takto vytvořených klausulí pro každý původní minterm sestavíme konjunkcí něco jako konjunktivní normální tvar výrokové formule, kde ale prvotní formule zastupují celé implikanty, které nebyly sloučeny. Z tohoto konjunktivního normálního tvaru máme vytvořit disjunktivní

normální tvar, a to zejména pomocí distributivních zákonů (věta 3.8). Výstupem pak bude určitý počet implikantů, což v našem případě odpovídá kombinacím, které hledáme.

Tabulku kombinací impementujeme tak, že každý minterm bude umístěn v jednom řádku této tabulky (tedy jeden článek řetězce) a minterm do řádku implementujeme jako pole, jehož délka je rovna počtu všech nesloučených implikantů (tj. počet řádků ořezané pomocné tabulky), přičemž index v tomto poli budeme vztahovat ke konkrétnímu implikantu tak, že index pole budeme chápat jako pořadí řádku v ořezané pomocné tabulce. Pokud v dané kombinaci bude implikant přítomen, bude v poli odpovídajícího indexu nenulová hodnota, jinak bude pole obsahovat hodnoty nulové.

Takto navržená implementace tabulky kombinací obchází problém se zaváděním zástupného označení nesloučených implikantů a navíc automaticky zajišťuje vlastnost formulí výrokové logiky, že v implikantu postačuje jedna přítomnost jednoho konkrétního literálu. Navíc máme vždy automaticky seřazeny literály ve shodném pořadí, což umožní jejich efektivnější vzájemné porovnávání, zda se jedná o implikanty ekvivalentní či nikoli.

Funkce `developCombinations()` nejprve zajistí jako nultou iteraci naplnění tabulky kombinací prvotní kombinací odpovídající těm implikantům, které implikují první z původních mintermů, což jsou jednotlivé literály odpovídající nesloučeným implikantům, neboť to odpovídá tomu, že v rámci každého výsledného implikantu má být některý z těchto literálů. Dále pak funkce `developCombinations()` volá cyklicky funkci `expand()`, které poskytne současnou podobu tabulky kombinací a následující klausuli, a přijme od ní aktualizovanou podobu této tabulky kombinací, do které je předaná klausule zapracovaná. Po každé iteraci se provede odstranění duplicit pomocí procedury `removeDupl()` a výsledná tabulka kombinací je předána zpět volající funkci `doSecondPhase()`.

Funkci `expand()` jsme implementovali tak, že každý implikant uvedený v původní tabulce kombinací nakombinujeme s každým z literálů přidávané klausule. Tak, jak máme implementovanou tabulku kombinací a klausule, to znamená, že pro každý literál v klausuli (tj. pro každý nenulový záznam v oříznuté pomocné tabulce v poli implikací s daným indexem) vytvoříme v nové tabulce kombinací další řádek, do kterého překopírujeme implikant z původní tabulky a doplníme ho o záznam literálu, který přidáváme.

Odstranění duplicit v tabulce kombinací zajišťujeme v rámci funkce `developCombinations()` pomocí procedury `removeDupl()`, která projde tabulkou kombinací a porovná každý implikant s každým a v případě jejich ekvivalence (tj. v naší implementaci mají shodně vyplněná pole) jeden z nich odstraní, tedy jej ze spojového seznamu vynechá.

Zjistit možné kombinace je jedním z úkolů, které jsme podle popisu Quine–McCluskeyho algoritmu měli v druhé fázi vyřešit, tím dalším je vybrat z nich ty nejkratší. Nejkratší ve smyslu nejméně implikantů, a z těchto ještě ty nejkratší, tedy s nejmenším počtem literálů. Obojí máme implementováno ve funkci `removeLong()`, kterou funkce `doSecondPhase()` spustí po obdržení konečné tabulky kombinací.

Funkce `removeLong()` vyřeší nejprve první úlohu, tedy najít nejkratší kombinaci. Necháme ji proto projít každý řádek tabulky kombinací a zjistit minimální počet nenulových polí v každém svém řádku. Začneme tak, že nastavíme tento počet na počet všech nesloučených implikantů – více jich být nikdy nemůže, a pokud napočítá v nějakém řádku méně, než je v této proměnné, přepíše její hodnotu na tuto hodnotu nižší. Po projití celým polem nám zůstane právě hledaný minimální počet kombinací. Poté projde funkce `removeLong()` tabulkou znovu a každý řádek tabulky, který obsahuje více kombinací, než je tento limit, opět ze zřetězeného seznamu vynechá, tedy odstraní kombinaci, která je delší, než je ta minimální.

V rámci funkce `removeLong()` si dále předpočítáme do předem připraveného pole o délce rovné počtu nesloučených implikantů, kolik literálů obsahuje který nesloučený implikant. Nato opět projdeme tabulkou kombinací a obdobně, jako jsme hledali nejmenší počet kombinací, nyní hledáme nejmenší počet literálů dané kombinace, což je vždy součet počtu literálů za každý implikant, který jsme si právě uložili do pomocného pole, kde každý index odpovídá indexu v ořezané pomocné tabulce, a tedy indexu v poli kombinací. Vypočítaný nejmenší počet literálů použijeme k tomu, že ještě jednou projdeme tabulku kombinací, a vynecháme – odstraníme kombinace, které mají počet literálů vyšší. Do volající funkce pak předáváme tabulku kombinací, která obsahuje pouze ty kombinace, které jsou minimální podle zadání.

Tímto jsme dosáhli našeho cíle nalézt minimální DNT (resp. KNT), protože správná kombinace jeho implikantů (resp. klausulí) je zakódována v každém zbývajícím řádku tabulky kombinací, kde každá nenulová hodnota daného indexu odpovídá nesloučenému implikantu v odpovídajícím řádku ořezané pomocné tabulky. Zbývá už jen předat tento výsledek do připravené proměnné typu struktury `SMINNF`, což zajistí funkce `buildAllMinNFs()`.

Tímto končí i druhá fáze vlastní minimalizace a do obslužného cyklu je předán výsledný seznam minimálních normálních tvarů. Podle uživatelského zadání pak obslužný cyklus volá buď výpis jednoho minimálního normálního tvaru, nebo všech minimálních normálních tvarů. Jeden minimální normální tvar volá pomocí procedury `renderOneMinNF()`, která zajistí výpis prvního z nalezených minimálních normálních tvarů pomocí zavolání funkce `renderMinNF()`, které předá ukazatel právě na tento první minimální normální tvar, který je v příslušném poli proměnné typu struktury `SMINNF` obsažen. Pro výpis všech minimálních normálních tvarů volá proceduru `renderAllMinNFs`, která pro každý z nalezených minimálních normálních tvarů zavolá rovněž funkci `renderMinNF()`. Každý z nalezených minimálních normálních tvarů necháme pro přehlednost vypsat na samostatný řádek.

V rámci implementované funkce `renderMinNF()` zajistíme, aby zápis těchto výrokových formulí odpovídal syntaxi, kterou jsme navrhli, tedy implikanty v minimálním DNT budou složeny z literálů oddělených symbolem `'c'` pro konjunkci a navzájem spojeny symbolem `'d'` pro disjunkci, pro minimální KNT budou literály odděleny symbolem `'d'` a jednotlivé klausule pospojovány symboly `'c'`.

Samozřejmě převedeme do této syntaxe i námi přeznačené literály v nesloučených implikantech. Tedy pro minimální DNT ponecháme označení literálu zastupujícího primární formuli, ale namísto literálu zastupujícího negovanou primární formuli, který jsme zjednodušeně zapisovali malým písmenem, se navrátíme k původnímu označení velkým písmenem s uvozujícím symbolem `'n'` pro operaci negace. Obdobně pro minimální KNT, nicméně je třeba zajistit dokončení negace výsledku, tedy namísto literálu zastupujícího primární formuli budeme tento vypisovat s uvozujícím symbolem `'n'` a namísto malého písmena zastupujícího negovanou primární formuli vypíšeme velké písmeno.

Pro zjednodušení čtení výstupu zejména v případě požadavku výpisu všech minimálních normálních tvarů, ať již disjunktivních či konjunktivních, použijeme pro obě varianty minimálních normálních tvarů závorky, a to i když nejsou nezbytně nutné, jako například kolem implikantu v minimálním DNT, který je tvořen jediným literálem.

Po výpisu uživatelem požadovaných výstupů na standardní výstup se obslužný cyklus vrací na svůj začátek k načtení případného dalšího uživatelského vstupu.

5.5 Uživatelská příručka

Uživatelskou příručku připravíme v podobě standardní manuálové stránky spouštěné v shellu zpravidla pomocí příkazu `man`.

Naši manuálovou stránku, tedy stránku k programu `minim`, zařadíme do skupiny 1, tedy mezi ostatní běžné programy, které může uživatel spustit z prostředí shellu. Proto soubor, který vytvoříme, bude mít v názvu jméno programu a příponu `'1'` – `minim.1`. Uživatelská příručka bude podle obvyklého standardu vytvořena v anglickém jazyce a počítáme s tím, že bude uložena v hlavním adresáři uživatelských příruček `man`.

Do příručky zahrneme oddíly:

- Název (v anglickém znění NAME)
- Rozhraní (v anglickém znění SYNOPSIS)
- Popis (v anglickém znění DESCRIPTION)
- Návratová hodnota (v anglickém znění RETURN VALUE)
- Autoři (v anglickém znění AUTHORS)

Do oddílu „Název“ uvedeme název příkazu a jeho stručný popis, tedy informaci o tom, že se jedná o program, který minimalizuje formule výrokové logiky. Ostatní možné výstupy programu, jakým je například pravdivostní tabulka, uvedeme až v dalším oddílu, aby tento oddíl „Název“ byl co možná nejstručnější. Z důvodu stručnosti se proto omezíme právě pouze na uvedení hlavního výstupu programu „minimalizuje výrokové formule“, a to bez dodatečných upřesňujících informací, které uživatel nalezne dále.

Do oddílu „Rozhraní“ vložíme informaci o rozhraní programu – jakým způsobem je možné s programem pracovat a jaké jsou podporované přepínače. Program je připraven tak, aby byl spustitelný z shellu příkazem `minim` a volba typu výstupu byla uživatelem zadána pomocí přepínačů. Jednoduše tak v tomto oddílu postačí uvést právě název programu a seznam přepínačů, a to jako čistý seznam bez dalšího popisu jejich funkcí, což má být obsahem následujícího oddílu.

Chování programu blíže charakterizujeme v oddílu „Popis“, ve kterém rovněž zmíníme odlišné chování programu ve specifických situacích, jako jsou tautologie či kontradikce. Předně do tohoto oddílu rozvedeme informaci z prvního oddílu „Název“, že program `minim` hledá minimální disjunktivní a konjunktivní normální tvary výrokových formulí, které musí být od sebe odděleny odřádkováním, a načítání vstupu je ukončeno s prázdným řádkem.

Další důležitou informací, kterou je třeba do oddílu „Popis“ uvést, je podoba vstupních výrokových formulí, tedy tvar textového řetězce zadávaný uživatelem, který odpovídá programem akceptované syntaxi výrokových formulí tak, jak jsme ji navrhli a do programu následně implementovali. Zmínit tedy musíme, že se jedná o infixní zápis, který pro označení primárních výrokových formulí využívá velkých písmen `'A'` – `'Z'` a pro logické spojky používá `'n'` pro negaci, `'d'` pro disjunkci, `'c'` pro konjunkci, `'i'` pro implikaci a `'e'` pro ekvivalenci. Nakonec je třeba k syntaxi uvést, že přednost spojek je určena v pořadí `'n'`, `'c'`, `'d'`, `'i'` a `'e'`, může být změněna použitím závorek a v případě zřetězení spojek stejné přednosti je formule vyhodnocována zleva.

Dosud máme v oddílu „Popis“ uvedeny informace o tom, jakým způsobem má uživatel vložit vstup a jaké jsou možné výstupy. Nyní ještě potřebujeme uživatele informovat o tom, jakým

způsobem dosáhne požadovaného typu výstupu. V oddílu „Rozhraní“ jsme přepínače pouze vyjmenovali, zde k nim přiřadíme jednotlivé typy výstupů. Pro přepínač `-C` program vrací všechny minimální KNT vstupní formule, pro přepínač `-D` všechny minimální DNT a pro přepínače `-c` jeden minimální KNT a `-d` jeden minimální DNT. Pro přepínač `-e` program zopakuje vloženou vstupní formuli a pro `-t` vypíše její pravdivostní tabulku. Jako součást vyjmenování chování jednotlivých přepínačů uvedeme k přepínači `-d`, že tento přepínač je defaultní, tzn. v případě neuvedení žádného přepínače bude program postupovat, jako by byl zadán právě tento přepínač `-d`.

Do oddílu „Návratová hodnota“ pak vepíšeme informaci o tom, že v případě úspěchu programu je jeho návratová hodnota rovna počtu nekorektních vstupů a v opačném případě (použití nepodporovaných přepínačů) je vrácena hodnota -1.

Poslední informací, kterou do uživatelské příručky uvedeme, je autor programu včetně mailového spojení.

Kapitola 6
Testování

Vytvořený zdrojový kód, který jsme uložili do zdrojových souborů `minim.c`, `minform.c`, `table.c` a `formula.c` a hlavičkových souborů `minim.h`, `minform.h`, `table.h` a `formula.h` převedeme do spustitelného programu pomocí souboru `Makefile` (viz výpis kódu 6.1), ve kterém nadefinujeme, jakým způsobem má být program přeložen a sestaven.

Překlad provedeme pomocí překladače pro jazyk C daného proměnnou prostředí `$CC` (viz výpis kódu 6.1, ř. 19, resp. 22) s nastavením parametrů (viz výpis kódu 6.1, ř. 1) tohoto překladače na:

- `--std=99` pro nastavení konkrétní verze jazyka C dle standardu ISO C
- `-pedantic` pro zobrazení všech varování striktně podle výše uvedeného standardu
- `-Wall` pro zobrazování varování z oblastí, které jsou pod tímto parametrem definované
- `-Wextra` pro zobrazování varování z dalších oblastí, které nejsou pokryty parametrem `-Wall`
- `-g` pro zobrazení informací o ladění programu

S přihlédnutím k přenositelnosti mezi operačními systémy byl pro překladač navíc přidán parametr `-D_GNU_SOURCE`, kterým byla odstraněna nekompatibilila s operačním systémem Debian GNU/Linux 9.

Program a jeho standardní manuálovou stránku necháme uložit do složek `bin`, resp. `man/man1`, které umístíme do složky, která odpovídá proměnné prostředí `$(HOME)` (viz výpis kódu 6.1, ř. 3 – 5). Výsledný spustitelný binární soubor chceme, aby se jmenoval `minim`, manuálovou stránku máme pojmenovanou `minim.1` a soubory s koncovkou `.o`, které vzniknou překladem příslušných zdrojových souborů (s koncovkou `.c`), do proměnné `OBJ` (viz výpis kódu 6.1, ř. 7 – 9). Soubory, které nám budou sloužit jako vstupy do testování, a jejich výstupní protějšky uložíme do pomocné proměnné `TEST` (viz výpis kódu 6.1, ř. 10 – 14), která poslouží při kontrole toho, zda pro testování máme k dispozici všechny potřebné soubory.

Jako první pravidlo máme v souboru `Makefile` definováno pravidlo `all` (viz výpis kódu 6.1, ř. 16), které je tedy defaultní (spustí se nejen při zadání příkazu `make all`, ale i po zadání příkazu `make` bez specifikace pravidla) a zajišťuje vytvoření binárního souboru `minim`.

K finálnímu sestavení binárního souboru `minim` je potřeba existence všech souborů s koncovkami `.o`, tedy všech souborů uvedených v proměnné `$(OBJ)` (viz výpis kódu 6.1, ř. 18). Má-li

Výpis kódu 6.1 Obsah souboru makefile

```
cntCFLAGS       = -std=c99 -Wall -Wextra -pedantic -g -D_GNU_SOURCE

PREFIX  = $(HOME)
BINDIR  = $(PREFIX)/bin
MANDIR  = $(PREFIX)/man/man1

BIN     = minim
MAN     = minim.1
OBJ     = minim.o formula.o table.o minform.o
TEST    = test-syntax.in test-syntax.out \
          test-tables.in test-tables.out \
          test-mforms.in test-mincnf.out test-mindnf.out \
          test-morenf.in test-allcnf.out test-alldnf.out test-cmplx.out\
          test-broken.in

all: $(BIN)

$(BIN): $(OBJ)
        $(CC) $(CFLAGS) -o $(BIN) $(OBJ)

%.o : %.c
        $(CC) -c -o $@ $< $(CFLAGS)

test: $(BIN) $(TEST)
        ./$(BIN) -e < test-syntax.in | diff -uw - test-syntax.out
        ./$(BIN) -t < test-tables.in | diff -uw - test-tables.out
        ./$(BIN) -c < test-mforms.in | diff -uw - test-mincnf.out
        ./$(BIN) -d < test-mforms.in | diff -uw - test-mindnf.out
        ./$(BIN)    < test-mforms.in | diff -uw - test-mindnf.out
        ./$(BIN) -C < test-morenf.in | diff -uw - test-allcnf.out
        ./$(BIN) -D < test-morenf.in | diff -uw - test-alldnf.out
        ./$(BIN) -DDdeCtcte < test-morenf.in | diff -uw - test-cmplx.out
        ./$(BIN) -etcdCD < test-broken.in | diff -uw - /dev/null
        ./$(BIN) -etcdCD < test-broken.in; \
                [ $$? -eq $(shell wc -l < test-broken.in) ]
        ./$(BIN) -x < test-syntax.in 2>&1 /dev/stdout | \
                diff -uw - test-broken.opt

install: $(BIN) $(MAN) test
        install -d -m 755 $(BINDIR) && install -m 755 $(BIN) $(BINDIR)
        install -d -m 755 $(MANDIR) && install -m 644 $(MAN) $(MANDIR)

uninstall:
        rm -rf $(BINDIR)/$(BIN)
        rm -rf $(MANDIR)/$(MAN)

lint: $(MAN)
        mandoc -Tlint -Wstyle $(MAN)

doc:
        doxygen

clean:
        rm -fr $(BIN) *.o *.core *~ doc
```

je utilita `make` již k dispozici, zahájí sestavování (viz výpis kódu 6.1, ř. 19) pomocí určeného překladače s nastavenými parametry, a to tak, že výsledný soubor uloží s názvem shodným s proměnnou `$(BIN)`, tedy s názvem `minim`.

Nemá-li utilia `make` k dispozici při sestavování programu potřebné `.o` soubory, vytvoří je pomocí dalšího pravidla (viz výpis kódu 6.1, ř. 21 – 22). Každý z `.o` souborů vznikne z příslušného zdrojového souboru `.c` s využitím hlavičkových souborů, a to tak, že zavolaný překladač pro dané parametry vygeneruje `.o` soubor (použitím přepínače `-c`) s názvem shodným s proměnnou umístěnou na levo od symbolu „:“ (použitím konstruktu `-o $@`), přičemž použije zdrojový soubor s názvem, který je určen první položkou umístěnou vpravo od symbolu „:“ (použitím konstruktu `$<`), tedy shodného názvu s vytvářeným `.o` souborem, jen s koncovkou `.c`.

Utilitu `make` využijeme ještě pro další práci s programem, a to definováním několika dalších pravidel. První z nich využijeme pro zautomatizování testování. Toto pravidlo nazveme `test` (viz výpis kódu 6.1, 24 – 37) a potřebujeme k němu spustitelný soubor `minim` v jeho aktuální podobě, což zajistí právě utilita `make`, která, pokud dojde ke změně zdrojových souborů, vytvoří automaticky novou verzi spustitelného programu, a samozřejmě testovací soubory, ať se vstupy pro program či s očekávanými výstupy z programu. Naším cílem při zautomatizovaném testování je předkládat programu postupně jednotlivé výrokové formule a kontrolovat jeho výstup pomocí porovnávací funkce `diff` se správnými výstupy, a to pro všechny podporované přepínače s přihlédnutím k tomu, že pro spuštění programu bez přepínače se tento má chovat jako po spuštění s přepínačem `-d`.

K testování s výhodou využijeme vlastnosti programu, že načítá zadání ze standardního vstupu a zasílá výsledek na standardní výstup, takže si proces testování zautomatizujeme a na standardní vstup přesměrujeme předem připravená zadání v jednotlivých vstupních souborech, přičemž je budeme spouštět s různými přepínači (příp. bez přepínače). Výsledky pak rovněž přesměrujeme, a to na standardní vstup porovnávací funkce `diff`, která porovná takto obdržený vstup oproti souborům obsahujícím správná řešení, čímž nahradíme opakovanou vizuální kontrolu výstupů. Jednotlivé připravené testy rozebereme podrobněji v následujících podkapitolách.

Do souboru `Makefile` nakonec ještě nadefinujeme pravidla pro instalaci programu – `install` (viz výpis kódu 6.1, ř. 39 – 41), odinstalaci programu – `uninstall` (viz výpis kódu 6.1, ř. 43 – 45), kontrolu manuálových stránek – `lint` (viz výpis kódu 6.1, ř. 47 – 48), vygenerování html dokumentace programem Doxygen – `doc` (viz výpis kódu 6.1, ř. 50 – 51) a vyčištění adresáře od nepotřebných souborů, které mohou vzniknout při použití jednotlivých pravidel – `clean` (viz výpis kódu 6.1, ř. 53 – 54).

Otestovat program pro nás znamená zjistit, zda program na konkrétní zadání vrací předpokládaný – požadovaný výstup, tedy zda program na korektní vstup vrací správný textový řetězec (při použití přepínače `-e`), správnou pravdivostní tabulku (při použití přepínače `-t`) a rovněž správné odpovědi i na další přepínače, jejich kombinaci či spuštění bez přepínače, a to i se zohledněním speciálních případů tautologie a kontradikce. Současně je však třeba zkontrolovat, zda program správně reaguje i na nekorektní vstupy, tedy že nevrací nic. Každé z těchto variant korektních a nekorektních vstupů věnujeme nyní samostatný oddíl.

6.1 Příprava korektních vstupů

Program má za úkol načítat výrokové formule a po provedení jejich syntaktické analýzy, pro případy korektně zadaných výrokových formulí, vracet požadované výstupy. Jednotkových testů připravíme několik sad, a to jako vstupy pro přepínač `-e` (umístíme je do souboru s názvem

`test-syntax.in` a testovat je budeme pomocí funkce `diff` oproti souboru `test-syntax.out`) a pro přepínač `-t` (umístíme je do souboru `test-tables.in` a testovat je budeme oproti souboru `test-tables.out`). Pro přepínače `-c` a `-d` vytvoříme jednu sadu testů pro oba přepínače shodnou (bude uložena v souboru `test-mforms.in`, přičemž testování proběhne oproti souborům `test-mincnf.out` pro přepínač `-c` a `test-mindnf.out` pro přepínač `-d`) a stejně tak připravíme jednu vstupní sadu testů jako vstup pro oba přepínače `-C` a `-D` (testování vstupu ze souboru `test-morenf.in` pak proběhne oproti souborům `test-allcnf.out` pro přepínač `-C` a `test-alldnf.out` pro přepínač `-D`).

Protože má program načítat řádky ze vstupu do konce vstupu či do prvního prázdného řádku, otestujeme správnost tohoto chování v rámci testovací sady pro přepínač `-e`, a to tak, že po vytvoření sady testů přidáme do vstupního souboru jedno odřádkování navíc a zopakujeme poslední vstup. Výstupní soubor už ale upravovat nebudeme, protože tak, jak byl vytvořen, odpovídá požadovanému chování.

Vyjdeme-li z definice výrokové formule, pak je třeba otestovat v prvé řadě primární formule. Tyto jsou zastoupeny symboly `'A'` – `'Z'`, a tedy prvních 26 testů syntaktického analyzátoru (testujeme přepínačem `-e`) připravíme pro vstup sestávající z tohoto jediného znaku. Očekávanými výstupy jsou pro přepínač `-e` triviálně výstup shodný se vstupem. Pro testy ostatních přepínačů na tomto triviálním vstupu vybereme pouze jeden z nich, např. se symbolem `'A'`. Pro tento vstup pak přepínač `-t` vrací kromě záhlaví dvou řádkovou tabulku s jedním sloupcem pro danou primární formuli a výsledným sloupcem, který hodnotami kopíruje hodnoty prvního sloupce. Chceme-li otestovat na tomto vstupu i minimalizaci (přepínače `-d` a `-c`), pak, protože se nejedná o tautologii ani kontradikci, je třeba najít i jejich očekávaný výstup. Tím je pro oba přepínače právě zadaná primární formule, přičemž s ohledem na implementovaný způsob výpisu těchto minimálních normálních tvarů je třeba tuto vložit do závorek.

Podle definice výrokové formule můžeme postupovat v testování dále tak, že otestujeme, pro dané primární formule A a B, že program správně pracuje i s výrokovými formulemi nA, AdB, AcB, AiB a AeB. Pro přepínač `-e` opět postačí triviální zkopírování vstupního řetězce do souboru s výstupem (pro případy binárních spojek je třeba doplnit o vnější závorky). Pravdivostní tabulky pro přepínač `-t` přebírají hodnoty z tabulky pravdivostních hodnot 3.1, jen je třeba při přípravě výstupu dodržet implementovaný způsob vyplnění ohodnocení primárních formulí, který je postaven tak, že v prvním sloupci horní polovina zastupuje pravdivostní ohodnocení jako nepravdy a spodní polovina jako pravdy a každý následující sloupec má dvojnásobný počet pravdivostních skupin, které se střídají, přičemž začínáme shora od bloků nepravdivých ohodnocení.

Pro vstup nA bude jednoduchá i příprava výstupů pro další přepínače `-c` a `-d`, protože postačí provést z již vytvořených výstupů pro výrokovou formuli A záměnu A za nA. Pro další čtyři testované výrokové formule už je připravení výstupu složitější a vyžadovalo by projít celým procesem výpočtu minimálních normálních tvarů přesně podle implementace, protože minimální normální tvary, ať disjunktivní či konjunktivní, jsou sice pevně určeny, ale až na pořadí, tedy bychom museli vytvořit všechny možnosti a testovat, zda je výstup s některým z nich (textově) shodný. Proto necháme program, aby nám tyto výstupy připravil a před uložením do výstupů z testů je překontrolujeme vizuálně. Tyto výstupy s konkrétním pořadím již dále můžeme používat, protože v rámci implementace algoritmu pro hledání minimálních normálních tvarů není nikde použito náhodného pořadí, vše je skriktně určeno, a tedy takto jednou sestavený tvar výstupu již bude pro všechny výrokové formule ekvivalentní se zadanou výrokovou formulí vracen stále stejný (je potřeba pouze shodná pravdivostní tabulka, která vstupuje do vlastní minimalizace, zadaný tvar výrokové formule nesmí mít vliv).

Pokud bychom dále chtěli pokračovat v testování dle definice výrokové formule, navíc s přihlédnutím k tomu, že povolujeme jako pomocný znak mezeru a uzávorkování v některých případech není třeba a přednost operací řešíme jinak, pak bychom museli připravit vstupy a k nim odpovídající výstupy pro všechny možné výrokové formule, které operační systém umožní do programu načíst. Není jich tedy nekonečně mnoho, nicméně i když bychom se omezili jen na korektní řetězce, což bychom ale stejně museli provést vizuální kontrolou, je toto množství pro manuální testování naprosto nereálné. S ohledem na časovou náročnost je nereálné i automatizované testování – i kdybychom je měli či nově vytvořili. Omezíme se již proto jen na přípravu určitých typů výrokových formulí, na kterých otestujeme, zda syntaktická analýza splňuje i další naše požadavky vycházející ze syntaxe výrokových formulí tak, jak jsme ji navrhli. Pro otestování správnosti následného výpočtu minimálních normálních tvarů pak využijeme komplexnější úlohy – takové, aby měly více minimálních DNT, resp. KNT. Tyto komplexnější testy zařadíme do souboru s názvem `test-morenf.in` a očekávané výstupy, oproti kterým budeme skutečné výstupy porovnávat, pak do souborů `test-allcnf.out` pro přepínač `-C` a `test-alldnf.out` pro přepínač `-D`.

Pokud budeme dále předpokládat, že program vytváří stromovou strukturu správně dle definice 3.4, pak je třeba ještě otestovat naše požadavky na přednost uzávorkování a logických spojek při výstavbě stromové struktury, která má vliv na vyhodnocení pravdivostní hodnoty výrokové formule pro daná ohodnocení primárních formulí.

Největší přednost (pokud není určena závorkami jinak) má mít spojka negace. Uvažujme proto následující čtyři výrokové formule $nAdB$, $nAcB$, $nAiB$ a $nAeB$. Očekáváme podle našeho požadavku, že první z těchto výrokových formulí bude ekvivalentní s výrokovou formulí $(nA)dB$, aby byla splněna přednost logické spojky negace. Pokud by program vyhodnocoval tuto výrokovou formuli nesprávně, pak by ji vyhodnotil jako $n(AdB)$, což by bylo zřejmé jak ze samotného zpětného sestavení výrokové formule (přepínač `-e`), tak také z pravdivostní tabulky (přepínač `-t`). Analogicky přípravíme pro testování oběma přepínači `-e` a `-t` i výrokové formule $nAcB$ a $nAiB$, protože z pravdivostní tabulky bychom dovodili, že přednost logické spojky negace oproti konjunkci, resp. implikaci není splněna. Tato úvaha však nebude platit pro poslední z uvedených výrokových formulí, protože formule $nAeB$ je logicky ekvivalentní jak s formulí $(nA)eB$, tak s formulí $n(AeB)$ a požadovanou přednost negace před ekvivalencí můžeme zkontrolovat jen pomocí přepínače `-e`.

Abychom vyloučili možnost, že spojka negace byla vyhodnocena přednostně před disjunkcí, konjunkcí a implikací z důvodu pořadí v zápisu, připravíme ještě další čtyři výrokové formule, a to $AdnB$, která by po nesprávném vyhodnocení měla pravdivostní tabulku odpovídající výrokové formuli $n(AdB)$, se kterou není logicky ekvivalentní. Dále potom $AcnB$, jejíž nesprávné vyhodnocení by vedlo na pravdivostní tabulku výrokové formule $n(AcB)$, se kterou rovněž není logicky ekvivalentní, a $AinB$, která rovněž není logicky ekvivalentní s výrokovou formulí $n(AiB)$, na kterou by vedlo nesprávné vyhodnocení. Čtvrtou výrokovou formuli $AenB$ opět otestujeme jen pomocí přepínače `-e`, protože pravdivostní tabulka by pro případy korektní syntatické analýzy i nekorektní syntaktické analýzy měla stejné hodnoty.

Dále otestujeme přednost logické spojky konjunkce oproti disjunkci, implikaci a ekvivalenci a pro vyloučení možnosti, že vyhodnocení závisí na pořadí v zápisu, uvedeme v zápětí i tvar se záměnou pozic v zápisu. Výroková formule $AdBcC$ má být vyhodnocena jako $Ad(BcC)$. Pokud by byla vyhodnocena nesprávně, vedla by na výrokovou formuli $(AdB)cC$, se kterou není logicky ekvivalentní, a test je tedy pro oba přepínače `-e` i `-t` na místě. Obráceně otestujeme výrokovou formuli $AcBdC$, která má být vyhodnocena jako $(AcB)dC$, a tedy nikoli jako formule $Ac(BdC)$, se kterou není logicky ekvivalentní, a testy mají tedy rovněž smysl provádět. Dále otestujeme výrokovou formuli $AiBcC$, která má být vyhodnocena jako $Ai(BcC)$, a nikoli jako výroková formule s ní logicky neekvivalentní $(AiB)cC$. Pro obrácené pořadí otestujeme výrokovou

formuli $AcBiC$, která má být logicky ekvivalentní s $(AcB)iC$, a tedy ne s $Ac(BiC)$. Na konjunkci otestujeme ještě přednost před ekvivalencí, tedy pomocí výrokových formulí $AeBcC$ a $AcBeC$, které mají být vyhodnoceny jako $Ae(BcC)$ a $(AcB)eC$ a nejsou logicky ekvivalentní s výrokovými formulemi, na které by vedlo jejich nesprávné vyhodnocení.

Analogicky přípravíme další výrokové formule k otestování přednosti disjunkce před implikací a ekvivalencí a přednosti implikace před ekvivalencí, a to u každé pro obě varianty pořadí v zápisu. U všech těchto případů kromě jednoho vede případné nesprávné pořadí vyhodnocení logických spojek na výrokové formule, které nejsou logicky ekvivalentní s těmi, které jsou z hlediska pořadí operací vyhodnoceny správně, a tedy provedení těchto testů má smysl. Jedinou zmíněnou výjimku tvoří výroková formule $AiBdC$, která je logicky ekvivalentní jak s výrokovou formulí $(AiB)dC$, tak s výrokovou formulí $Ai(BdC)$, a nemá tak smysl zařazovat ji do testování přepínačem `-t`, protože z podstaty věci nemůže rozhodnout, zda námi testovaná vlastnost – přednost disjunkce před implikací je splněna či nikoli.

V souvislosti s pořadím vyhodnocování logických spojek máme v návrhu syntaxe výrokových formulí požadavek, aby v případě stejných logických spojek byl výraz vyhodnocován zleva. Pro logické spojky konjunkce, disjunkce a ekvivalence nemá tento požadavek ve vztahu k přepínači `-t` význam, protože výraz $AdBdC$ je logicky ekvivalentní s $(AdB)dC$ i s $Ad(BdC)$ a analogicky pro konjunkci a ekvivalenci. Pouze pro případ implikace má smysl tento test provést, protože výraz $AiBiC$ má mít v pravdivostní tabulce hodnoty shodné s hodnotami pro výraz $(AiB)iC$, které jsou odlišné od hodnot pro výraz $Ai(BiC)$.

Následně vytvoříme testy pro vyhodnocení komplexnějšího výrazu, který převezmeme z [1], přičemž se jedná o úlohu, na které byl algoritmus Quine–McCluskey demonstrován. Jedná se o výrokovou formuli zapsanou v úplném disjunktivním normálním tvaru, kterou pro náš případ přepíšeme v námi požadované syntaxi. Výstup z programu necháme programem vytvořit, výsledek zkontrolujeme a bude-li správný, zařadíme tento výstup do příslušné testovací sady k ostatním výstupům. Tato výroková formule má jako výstup několik skutečně různých minimálních DNT, nicméně minimální KNT má pouze jeden. Abychom mohli otestovat správnost programu i pro výrokovou formuli, která má více minimálních KNT, využijeme výrokové formule [1] ještě jednou tak, že na vstup programu vložíme její negaci. Tím dosáhneme toho, že z pravdivostní tabulky budou do fáze vlastní minimalizace převzaty právě jen ty řádky, které jsou shodné s řádky, které pro původní nezměněnou výrokovou formuli byly použity k hledání minimálních DNT. Výsledek, který tedy algoritmus Quine–McCluskey při hledání minimálních KNT na základě znegované výrokové formule vrátí, bude shodný jako v předchozím případě, jen při výpisu tohoto výsledku bude zohledněno právě hledání minimálních KNT a výstup bude podle toho upraven (má být provedena negace nalezených minimálních DNT – ve smyslu negace literálů a záměny spojek konjunkce a disjunkce, čímž vzniknou hledané minimální KNT). V rámci testování této sady současně otestujeme ještě možnost, že program při načítání vstupu ignoruje mezery, které se ve vstupním řetězci vyskytnou, a uvedené dvě výrokové formule na několika místech doplníme o několik mezer.

Pro testování dalšího požadované chování programu umístíme vstupy do souboru s názvem `test-cmplx.in` a kontrolní výstupy do souboru `test-cmplx.out`. Testovat budeme na této sadě chování programu při zadání všech podporovaných přepínačů, tedy zda ve shodě s naším požadavkem budou jednotlivé výstupy ve správném pořadí (proto pořadí přepínačů zpřeházíme a můžeme otestovat i chování programu pro násobné použití některých přepínačů). Do sady zařadíme oba vstupy s více minimálními normálními tvary, a to proto, abychom zároveň otestovali, že při souběhu přepínače `-c` a `-C` (resp. `-d` a `-D`), je přepínač s malým písmenem ignorován.

Nakonec vytvoříme testy pro speciální případy tautologie a kontradikce, a to tak, že pro tyto účely použijeme výrokové formule $AdnA$ jako zástupce tautologií, resp. $AcnA$ jako zástupce

kontradikcí, které vložíme do testovacích sad pro testování syntaxe (soubor `test-syntax.in`), pro testování pravdivostní tabulky (soubor `test-table.in` a pro testování hledání minimálních normálních tvarů (soubor `test-mforms.in`). Adekvátně k přidaným testům upravíme odpovídající výstupní soubory. Pro tyto případy budou výstupy pro přepínač `-e` triviální, pravděpodobnostní tabulky pro přepínač `-t` budou odpovídat očekávaným hodnotám (samé logické pravdy, resp. logické nepravdy pro všechna ohodnocení) a pro zbývající přepínače `-d` a `-c` bude v kontrolním výstupním souboru uveden text „tautology“, resp. „contradiction“.

6.2 Příprava nekorektních vstupů

Nekorektní vstupy vzhledem k našemu programu můžeme rozdělit na dva typy – jeden z nich odpovídá nesprávnému zadání ve smyslu zadání nepodporovaného přepínače, a druhým typem je nesprávně zadaná vstupní formule. První typ otestujeme např. spuštěním testu s nepodporovaným přepínačem `-x`. Program se tedy má řádně ukončit, standardní výstup nechat prázdný a na chybový výstup zaslat daný textový řetězec. To zkontrolujeme pomocí funkce `diff` oproti výstupnímu souboru `test-broken.opt`, do kterého necháme chybové hlášení (včetně standardního výstupu) prvně vepsat, zkontrolujeme ho a následně již jen dohlížíme v rámci pravidla `test`, že se s případnými úpravami programu toto chybové hlášení nemění.

V případě vyloučení nekorektního vstupu prvního typu, který by vedl k okamžitému ukončení celého programu, máme ještě rozhodnout o korektnosti vstupu druhého typu, což musí být podle našeho návrhu definitivně rozhodnuto nejpozději před spuštěním modulu, který hledá pravdivostní tabulku, protože vstupem do tohoto modulu je právě (korektní) výroková formule zachycená ve stromové struktuře. Toto testování nekorektních vstupů tedy testuje implementace případů užití *UC4 – Načtení uživatelského zadání* a *UC3 – Parsování uživatelského vstupu*, které zmíněnému modulu `table` (případ užití *UC2 – Nalezení úplných normálních tvarů*) předchází.

Obdobně jako u testování korektních vstupů nelze provést otestování všech možných nekorektních vstupů, a to už jen s ohledem na možnost libovolně dlouhého vstupního řetězce. Vybereme tedy opět jen určité typy těchto nekorektních vstupů a uspořádáme je do jednoho souboru s názvem `test-broken.in`. Program na ně má reagovat tak, že na standardní výstup nic nezasílá, pouze se řádně ukončí a informace o počtu nekorektních vstupů má být obsažena v návratové hodnotě programu (návratová hodnota funkce `main()`). Kontrolní výstupní soubory budou tedy pro všechny testy nekorektních vstupů a pro všechny podporované přepínače obsahovat jen prázdný textový řetězec a návratová hodnota programu musí odpovídat počtu vstupních řádků, přesněji do prvního prázdného řádku, ale protože tuto kontrolu testujeme v rámci testování korektních vstupů, neberme nyní v úvahu jiné než neprázdné řádky.

Většina kontrolních mechanismů korektnosti je umístěna v modulu `formula`, který jako implementace syntaktického analyzátoru má za úkol výrokovou formuli sestavit. Předchozí načtení vstupu má v sobě zabudovanou pouze kontrolu toho, zda je vložený textový řetězec nenulové délky, což by program vyhodnotil jako konec načítání a ukončil by se. Naopak toto jediné je kontrolováno v hlavní funkci `main()`, která obsahuje implementaci případu užití *UC4 – Načtení uživatelského zadání*, který textový řetězec načítá.

Modul `formula` má mimo jiné kontrolovat, že textový řetězec obsahuje pouze povolené znaky, mezi které patří symboly pro primární formule, symboly pro logické spojky, závorky a mezera. Vyberme jednu číslici, např. `'1'`, jeden speciální symbol, např. `'-'` a jedno malé písmeno nepatřící mezi logické spojky, např. `'p'`. Vytvořme např. z textového řetězce `AdBcC`, který odpovídá korektnímu vstupu, úpravou nekorektní vstup tak, že na začátek umístíme jako nepovolený znak vybranou číslici. Další test vytvořme tak, že za vybranou korektní výrokovou formuli umístíme

zvolený speciální znak a další tři testy nekorektních vstupů vytvořme tak, že vybrané písmeno `'p'` umístíme jednou libovolně dovnitř řetězce, jednou namísto některé z primárních formulí a jednou na místo jedné z logických spojek. Všechny takto vzniklé textové řetězce má modul vyhodnotit jako nekorektní vstupy a nechat standardní výstup bez odpovědi.

Nyní se můžeme soustředit na možné chyby ve vstupech, které sice obsahují jen povolené znaky, ale přesto se nejedná o korektní vstupy. Jedním z takových typů může být chybějící primární formule, přičemž pokud vyjdeme z námi využívaného korektního vstupu `AdBcC`, můžeme postupně vytvořit tři testy vynecháním vždy jedné z primárních formulí. Další test můžeme vytvořit vynecháním jedné z logických spojek. Pokud mohou primární formule či logické spojky v řetězci nedopatřením chybět, může se stát, že vstup bude obsahovat i jejich zdvojení, což mimo znak pro operaci negace a mezeru rovněž vede na nekorektní vstup. Proto můžeme vytvořit další test, ve kterém zdvojíme například první primární formuli, a další test, ve kterém zdvojíme například první logickou spojku disjunkci.

V rámci testování bychom neměli opominout uzávorkování výrazů. Pokud před používaný korektní vstup `AdBcC` umístíme levou závorku, ale nedoplníme pravou závorku a opačně, vytvoříme další dva testy nekorektních vstupů. Rovněž můžeme otestovat, že nelze levou a pravou závorku zaměnit, tedy že výraz `)AdBcC(` je nekorektním vstupem, případně, že za nekorektní vstup je považováno použití závorek bez existence uzávorkovaného výrazu, tedy např. `Ad()BcC`.

Nakonec ještě můžeme vyzkoušet, že program neakceptuje výrazy, které by sice odpovídaly jinému typu zápisu – např. postfixovému, ale nikoli infixovému, tedy např. `ABCdd`.

6.3 Vlastní testování

Pro testování máme nyní připraveno celkem 5 sad se vstupy (4 sady korektních vstupů a 1 sada nekorektních vstupů) a k nim odpovídající výstupy, tak, aby testování mohlo proběhnout automatizovaně pomocí utility `make` a její námi připraveného pravidla `test`. Překlad a vytvoření spustitelného souboru včetně otestování i všech zbývajících pravidel utility `make` provedeme na dvou operačních systémech, a to na Debian GNU/Linux 9 a Ubuntu 20.04.1 LTS.

Samotné testování spustíme příkazem `make test`, přičemž utilita `make` sama zkontroluje, zda nedošlo ve zdrojových souborech programu k žádné změně, případně automaticky sama zajistí opětovné přeložení a následné sestavení programu.

Protože žádný z provedených testů vyhodnocovaných pomocí pravidla `test` neohlásil rozdíl mezi očekávaným výstupem a výstupem skutečným, můžeme vyslovit závěr, že provedeným testováním nebyla nalezena v našem programu žádná chyba.

Kapitola 7
Závěr

Hlavním cílem této bakalářské práce bylo vytvořit počítačový program, který bude minimalizovat výrokové formule, tedy hledat minimální disjunktivní, resp. konjunktivní normální tvar výrokových formulí.

První dílčí úkol definovat minimální normální tvar výrokových formulí a popsat některý z algoritmů, který tento tvar hledá, jsme splnili v rámci kapitoly Analýza v podkapitole Základní pojmy (3.1). Při definování minimálních normálních tvarů jsme vyšli z normálních tvarů jak disjunktivních, tak konjunktivních a ke každému z těchto typů jsme zadefinovali pojmy minimální disjunktivní normální tvar a analogicky minimální konjunktivní normální tvar. Pro vlastní minimalizaci jsme vybrali algoritmus Quine–McCluskey, který jsme popsali v kapitole Analýza v podkapitole Algoritmus Quine–McCluskey (3.3).

Jako další dílčí úkol jsme měli navrhnout vhodný formální jazyk v rámci ASCII, aby bylo možno zachytit syntaxi výrokových formulí. Návrh tohoto požadovaného jazyka je uveden v kapitole Návrh v podkapitole Zápis formulí výrokové logiky (4.1). Navržený zápis obsahuje konstrukty potřebné pro zápis výrokových formulí, jak vyplývá z definice pojmu formule výrokové logiky, ale tak, že využívá pouze symboly v rámci základní tabulky ASCII.

Na základě takto navrženého jazyka jsme do programu implementovali parser výrokových formulí, který převádí textově zapsanou výrokovou formuli do stromové struktury, se kterou program dále pracuje. V rámci tohoto parseru je implementovaná kontrola vstupní formule a řádný zápis vstupní formule lze v programu zkontrolovat pomocí přepínače `-e`, který v případě úspěšného načtení řádné formule vrátí na standardní výstup zpětně zkonstruovanou textovou podobu této zadané výrokové formule.

Nad tímto parserem jsme dále implementovali hledání disjunktivního a konjunktivního normálního tvaru, a to jejich speciální tvar, tzv. úplný disjunktivní, resp. konjunktivní normální tvar, které využíváme jako vstup do algoritmu Quine–McCluskey. Program oba tyto tvary vypíše v rámci jednoho výstupu po použití přepínače `-t`, a to v podobě pravdivostní tabulky.

Nad těmito normálními tvary jsme nakonec implementovali popsaný minimalizační algoritmus. Konkrétní typ výstupu z programu pak určují přepínače. Pro výpis některého z minimálních disjunktivních, resp. konjunktivních normálních tvarů použije uživatel přepínač `-d`, resp. `-c`. V případě požadavku na výpis všech minimálních disjunktivních, resp. konjunktivních normálních tvarů pak uživatel spustí program s přepínačem `-D`, resp. `-C`. Případ spuštění programu bez přepínače je shodný s případem použití přepínače `-d`.

Informace o použití programu, podporovaných přepínačích a popis implementované syntaxe jsme shrnuli do standardní manuálové stránky, která je uživatelskou dokumentací programu. Dokumentace kódu vygenerovaná nástrojem Doxygen je k dispozici na přiloženém datovém nosiči.

Implementaci jsme provedli v jazyce C s využitím POSIX knihoven a vytvořený program úspěšně otestovali sadami testů k prověření jednotlivých součástí programu, jakými jsou načtení uživatelského zadání, parser, výpočet pravdivostní tabulky a samozřejmě minimalizace zadané výrokové formule. Dosáhli jsme tak splnění všech cílů, které jsme si v úvodu práce vytyčili.

Vytvořený program tak, jak byl navržen a implementován, je možné využít k dříve zmíněným účelům, tedy jako pomůcka pro řešení optimalizačních úloh, nácvik řešení těchto úloh či kontrola správnosti výsledků při řešení tohoto typu úloh. Kód programu pak lze dále upravovat, vybírat z něj určité části pro jiné programy či je nahrazovat jinými. Škála využitelnosti programu je široká a lze předpokládat, že své uplatnění nalezne.

Literatura

[1] Starý J. Úvod do matematické logiky, verze 24. listopadu 2021. https://courses.fit.cvut.cz/BI-ALO/matematicka-logika.pdf. Accessed: 2021/12/22.

[2] Trlifajová K. a Vašata D. Matematická logika. České vysoké učení technické v Praze, 2013. ISBN 978-80-01-05342-3.

[3] Shannon C. E. A symbolic analysis of relay and switching circuits. Trans. AIEE 57:12, 1938: p. 713 – 723.

[4] Šteffan P. Návrh systémů s digitálními integrovanými obvody a mikroprocesory pro integrovanou výuku vut a vŠb-tuo. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2014. ISBN 978-80-214-5072-1.

[5] Dijkstra E. W. Algol-60 translation. ALGOL Bulletin Supplement nr. 10, 1961. https://www.cs.utexas.edu/~EWD/MCReps/MR35.PDF. Accessed: 2021/12/22.

[6] Quine W. V. The problem of simplifying truth functions. The American Mathematical Monthly, vol. 59, nr. 8, 1952: p. 521 – 531.

[7] McCluskey E. J. Minimization of boolean functions. Bell System Technical Journal, vol. 35, 1956: p. 1417 – 1444.

[8] Staff of the Harvard Computation Laboratory. Synthesis of electronic computing and control circuits. B.S.T.J., Harvard University Press, Cambridge, Mass., vol. 26, 1951: p. 593 – 598.

[9] Karnaugh M. The map method for synthesis of combinational logic circuits. Trans. A.I.E.E., vol. 72, part I, 1953: p. 593 – 598.

[10] Danvy O. a Millikin K. Refunctionalization at work. Science of Computer Programming, vol. 74, 2009: p. 534 – 549. https://www.sciencedirect.com/science/article/pii/S0167642309000227. Accessed: 2021/12/23.

[11] Injosoft AB. Ascii code - the extended ascii table. https://www.ascii-code.com. Acessed: 2021/12/23.

[12] The Object Management Group®. Omg® unified modeling language®. https://www.omg.org/spec/UML/2.5.1/PDF. Accessed: 2021/03/20.

[13] The Open Group. Headers.
https://pubs.opengroup.org/onlinepubs/9699919799/idx/head.html.
Accessed: 2021/03/23.

[14] Dzonsons K. Openbsd manual page server.
http://man.openbsd.org/mdoc.
Accessed: 2021/12/23.

Kapitola A

Manuálové stránky

```
MINIM(1)                 BSD General Commands Manual                 MINIM(1)

NAME
     minim – minimize propositional formulas

SYNOPSIS
     minim [-CDcdet]

DESCRIPTION
     minim finds a minimal disjunctive and conjunctive normal form of proposi-
     tional formulas given as input. Formulas must be separated by a new line.
     The end of the input is marked by an empty line.  As a special case, it
     recognizes tautologies and contradictions.

     The input formula is specified in infix using the following language.
     Letters ‘A’ to ‘Z’ are the primary propositions. The propositional
     connectives are expressed as follows: ‘n’ for negation, ‘c’ for conjunc-
     tion, ‘d’ for disjunction, ‘i’ for implication, ‘e’ for equivalence.

     The binding precedence of the connectives is ‘n, c, d, i, e’; this can be
     overriden with parentheses.  When in doubt, connectives associate from the
     left.

     The options are as follows:

     -C      Print all minimal CNFs of every input formula.
     -D      Print all minimal DNFs of every input formula.
     -c      Print a minimal CNF of every input formula.
     -d      Print a minimal DNF of every input formula.  This is the default.
     -e      Echo every parsed input formula.
     -t      Print the truth table of every input formula.

RETURN VALUE
     On success, the number of incorrect formulas is returned.
     In case of incorrect options, -1 is returned.

AUTHORS
     Klara Cervenkova <cervekl1@fit.cvut.cz>

BSD                              January 31, 2022                              BSD
```

Kapitola B

Obsah přiloženého média

- `readme.txt` — stručný popis obsahu média
- `exe` — adresář se spustitelnou formou implementace
- `src`
  - `impl` — zdrojové kódy implementace
  - `thesis` — zdrojová forma práce ve formátu LaTeX
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF