



Západočeská Univerzita v Plzni

Fakulta Aplikovaných Věd

Informatika

Semestrální práce

Celočíselná kalkulačka s neomezenou přesností

Obsah

1	Zadání	2
1.1	Specifikace vyhodnocovaných výrazů	2
1.2	Specifikace výstupu programu	3
2	Analýza úlohy	3
2.1	Reprezentace čísel	3
2.2	Matematické operace	3
2.3	Interpretace infixové formy	4
2.4	Načítání výrazů ze vstupu	4
3	Popis implementace	5
3.1	Základní datové struktury	5
3.2	Struktura pro reprezentaci čísel	5
3.3	Matematické operace s MPT	5
3.4	Převod řetězce na MPT	7
3.5	Výpisy MPT	7
3.6	Operátory	8
3.7	Shunting yard	8
3.8	Vyhodnocení RPN	8
4	Uživatelská příručka	9
4.1	Přeložení a sestavení programu	9
4.2	Spuštění a ovládání programu	10
5	Závěr	11
	Odkazy	11

1 Zadání

Naprogramujte v jazyce ANSI C konzolovou aplikaci, která bude fungovat jako jednoduchý interpret aritmetických výrazů zapsaných v infixové formě. Vstupem programu bude kromě řídicích příkazů i seznam aritmetických výrazů obsahující celočíselné operandy (\mathbb{Z}) s neomezenou velikostí, zapsané ve dvojkové, desítkové či šestnáctkové číselné soustavě. Pro kódování operandů zapsaných ve dvojkové či šestnáctkové soustavě bude použit výhradně dvojkový doplňkový kód. Výstupem je pak odpovídající seznam výsledků vyhodnocení každého výrazu.

Program bude spouštěn příkazem `calc.exe [<vstupní-soubor>]`.

Symbol `<vstupní-soubor>` zastupuje nepovinný parametr – název vstupního souboru se seznamem aritmetických výrazů. Není-li symbol `<vstupní-soubor>` uveden, program bude fungovat v interaktivním módu, kdy se příkazy budou provádět přímo zadáním z konzole do interpretu.

1.1 Specifikace vyhodnocovaných výrazů

Výrazem může být pouze příkaz interpretu, nebo aritmetický výraz v infixové formě. Interpret je case-insensitive (tj. nerozlišuje velká a malá písmena). Minimální množina podporovaných příkazů:

- `quit ...` Ukončí interpret s návratovou hodnotou `EXIT_SUCCESS`.
- `bin ...` Výsledky výrazů budou vypisovány ve dvojkové soustavě (prefix `0b`).
- `dec ...` Výsledky výrazů budou vypisovány v desítkové soustavě (výchozí).
- `hex ...` Výsledky výrazů budou vypisovány v šestnáctkové soustavě (prefix `0x`).
- `out ...` Vypíše aktuálně vybranou číselnou soustavu (`bin`, `dec` nebo `hex`).

Tabulka 1: Minimální množina podporovaných funkcí a operátorů.

Operátor	Operace	Precedence	Arita	Asociativita
!	Faktoriál	4	unární	levá
^	Umocnění	4	binární	pravá
-	Číslo opačné	3	unární	pravá
*	Násobení	3	binární	
/	Celočíselné dělení	3	binární	levá
%	Zbytek po celočíselném dělení	2	binární	levá
+	Součet	1	binární	
-	Rozdíl	1	binární	levá
(Levá závorka	0		
)	Pravá závorka	0		

Celočíselné operandy mohou být ve zpracovávaných aritmetických výrazech zadány pomocí

- dvojkové soustavy (prefix `0b`),
- desítkové soustavy (bez prefixu),
- nebo šestnáctkové soustavy (prefix `0x`).

Pro kódování operandů bude použit výhradně dvojkový doplňkový kód, kde obecně platí:

$$\begin{aligned} 3 &= 0b000\dots00011 = \textcolor{red}{0b011} \neq \textcolor{red}{0b11} = 0b111\dots11111 = -1 \\ 11 &= 0b0001011 = 0x0000b = \textcolor{red}{0x0b} \neq \textcolor{red}{0xb} = 0xffffb = 0b1011 = -5 \end{aligned} \tag{1}$$

1.2 Specifikace výstupu programu

Program bude vždy vypisovat výsledek právě zadaného aritmetického výrazu či příkazu interpretu. Při načtení seznamu příkazů ze souboru musí výstup odpovídat přesně výstupu spuštění programu bez parametru (interaktivní režim).

Při výpisu výsledků pomocí jejich binární či hexadecimální reprezentace je, stejně jako u vstupu, použit dvojkový doplňkový kód. Vypsán bude vždy minimální potřebný počet binárních či hexadecimálních cifer – zvýrazněná část nerovnosti (1).

V případě chybného zadání příkazu či aritmetického výrazu vypíše interpret chybu dle tabulky 1 v úplném zadání [1] a pokračuje ve zpracování dalšího výrazu.

2 Analýza úlohy

2.1 Reprezentace čísel

Důležitým problémem k vyřešení je způsob uložení neomezeně přesných celých čísel v paměti. Nejprve je nutné si uvědomit, že neomezeně velká čísla není prakticky možné v počítači ukládat. Pro naše účely se tedy omezíme na ‘extrémně velká’ čísla, jejichž maximální, resp. minimální velikost bude diskutována dále v sekci 3.2. V jazyce C sám o sobě neexistuje datový typ, který by taková čísla dokázal ukládat (`long long` má nejvyšší přesnost a ani zdaleka nám nevystačí) a je tedy nutné vymyslet strukturu schopnou tato čísla reprezentovat, nejlépe s co nejmenší paměťovou zátěží.

Čísla bychom mohli reprezentovat např. polem jednobytových znaků ‘1’ nebo ‘0’, jednalo by se ale o velice neefektivní techniku, jejíž použití je v úplném zadání [1] silně nedoporučeno. Mnohem lepší alternativa je použití bitového pole [2]. U obyčejného pole bychom ale v paměti alokovali statickou velikost, tím pádem by pole mohlo být buď zbytečně velké nebo moc malé. Namísto staticky alokovaného pole tedy použijeme pole dynamicky realokované (dále *vektor*), což nám umožní alokovat paměť nejen dostatečně velkou, ale i minimální potřebnou, pro reprezentaci čísla.

2.2 Matematické operace

Nejdůležitější operací je sčítání, která je nutným základem pro další operace. Lze se inspirovat skutečným způsobem sčítání ve výpočetní technice, které se hardwarově řeší pomocí polovičních a úplných sčítaček, jenž lze jednoduše implementovat i softwarově pomocí odpovídajících logických operací. Museli bychom ale ke každému bitu přistupovat jednotlivě, což by mohlo být lehce zdouhavé. Efektivnější by bylo využití aritmeticko-logické jednotky, aby tyto operace udělala za nás a sčítání tedy dělala byte po byte s tím, že si pouze ohlídáme příznakový bit `CARRY` mezi byty.

K výpočtu opačných čísel se nabízí dvojkový doplněk, tedy znegování všech bitů a přičtení jedničky. Tuto operaci pak lze jednoduše využít ve spojení se sčítáním pro zhotovení operace rozdílu.

Násobení čísel $n \cdot m$ můžeme implementovat naivně (n -krát sečteme m , nebo naopak), pro velká čísla by ale takové řešení trvalo příliš dlouho. Využijeme místo toho techniku násobení, která se vyučuje na základních školách při násobení dvou dekadických čísel. Tento proces je v binární soustavě ještě jednodušší než v dekadické.

Operaci celočíselného dělení lze také implementovat naivně (od dělece odečítáme dělitele dokud můžeme), znovu by se ale jednalo o nevhodné řešení, které má alternativu opět v technice dělení ze základních škol.

Řešení pro výpočet zbytku po celočíselném dělení je zmíněn v zadání [1], implementujeme ho tedy stejně jako je implementován překladačem jazyka ANSI C:

$$a \% m = a - m \left\lfloor \frac{a}{m} \right\rfloor \quad (2)$$

Umocnění základu exponentem je další naivně implementovatelnou operací, pro vylepšení zde ale žádnou techniku ze základní školy nevyužijeme. Pro efektivnější výpočet mocniny v binární soustavě existuje velice jednoduchý a rychlý algoritmus [4] popsáný dále v sekci 3.3.

Poslední operací je faktoriál, který je neproblematičtější a pravděpodobně neexistuje jiné řešení, než postupné násobení $n! = n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 1$. Možná optimalizace by spočívala v uložení vypočítaného výsledku do paměti, v níž bychom již vypočítané výsledky hledali, tuto optimalizaci ale vynecháme.

2.3 Interpretace infixové formy

Infixová forma zápisu aritmetického výrazu (např. $4 + 2 * 3$) je pro vyhodnocení výsledku výrazu nevhodná. Operátory mají různé priority (precedence v tab. 1) a v infixové formě se tato priorita těžko zjišťuje. Tento problém se standardně řeší převodem infixové formy do formy tzv. ‘Reverzní Polské notace’ (RPN) [5], v níž je priorita operátorů jednoznačně určená. Formu RPN lze z infixové formy získat pomocí algoritmu Shunting yard [6].

Shunting yard umí převést infixovou formu výrazu na formu RPN, nedokáže ale v infixové formě najít syntaktické chyby, a proto musíme kontrolu infixové syntaxe zařídit sami. Shunting yard zpracovává infixovou formu znak po znaku. Můžeme u každého znaku, reprezentující číslo nebo některý z operátorů (tab. 1), zjistit, zda-li se předchozí operátor nebo číslo může před aktuálním znakem vyskytovat. Povolené výskyty jsou popsány v tabulce 2.

Tabulka 2: Seznam povolených předchůdců operátoru nebo čísla

Operátor	Binární)	Unární, levá	Unární, pravá (Číslo
Povolení předchůdci	Unární, levá Číslo)	Unární, pravá Binární Číslo (Bez předchůdce		

2.4 Načítání výrazů ze vstupu

Vzhledem k tomu, že výraz může být libovolně dlouhý, číst ze vstupu statické množství znaků by bylo nevhodné. Namísto toho můžeme využít struktury typu vektor k postupnému načítání všech znaků až po znak ukončovací (nulový nebo `'\n'`), přičemž každý znak převedeme funkcí `tolower` na znak malý.

3 Popis implementace

3.1 Základní datové struktury

V programu jsou často využívány datové struktury zásobník (struktura `stack_type`, soubory `stack.c/.h`) a již zmíněný vektor (struktura `vector_type`, soubory `vector.c/.h`). V průběhu programu bude také nutné převést data z vektoru do zásobníku. Funkce převedení je v souboru `conversion.c/.h`. Vzhledem k tomu, že se jedná o běžné datové struktury, nebudeme je podrobně popisovat. Zmíněné soubory jsou umístěny samostatně ve složce `data_structures`.

3.2 Struktura pro reprezentaci čísel

Struktura pro reprezentaci ‘nekonečně’ velkých čísel (struktura `mpt`, soubory `multiple_precision_type.c/.h`) je pouze obalovací strukturou pro vektor prvků typu `unsigned int` (dále označovány jako segmenty), jehož velikost se dynamicky realokuje podle potřeby.

Veškeré soubory týkající se struktury `mpt` a práce s ní se nacházejí ve stejnojmenné složce `mpt`.

Instance `mpt` lze vytvořit na haldě funkcí `mpt_allocate`, nebo na zásobníku, kde ji poté inicializujeme funkcí `mpt_init`. Takto vytvořené instance `mpt` budou ve vektoru mít jediný segment se zadanou číselnou hodnotou v rozsahu datového typu `int`. Pro získání vyšších, resp. nižších, čísel je zapotřebí přidat do vektoru další segmenty a v nich nastavit příslušné bity (funkce `mpt_set_bit_to`). Každý i -tý prvek ve vektoru odpovídá i -tému segmentu čísla. Nultý segment má váhu nejmenší a s každým dalším segmentem se váha zvyšuje.

Je nutné počítat s tím, že čísla reprezentujeme dvojkovým doplňkem. Např. číslo $2^{32} = 0b11\dots11$ se sice vejde do jednoho segmentu, ale ve dvojkovém doplňku je rovno -1 , a proto je nutné přidat navíc segment samých nul, aby hodnota odpovídala 2^{32} .

Vezmeme-li v potaz, že bity indexujeme v rozsahu `size_t`, a budeme-li předpokládat, že program běží na 64-bitovém stroji, vidíme, že jedno číslo `mpt` může mít až 2^{64} bitů a mít tedy hodnotu v rozsahu $\langle -2^{(2^{63})}, 2^{(2^{63})} - 1 \rangle$. Jedná se o čísla, která mají v dekadické soustavě přibližně $2,8 \cdot 10^{18}$ číslic. Nedokážeme tedy reprezentovat ‘nekonečně velká’ čísla, na druhou stranu jsou tato čísla tak velká, že bychom na výsledky operací s nimi čekali příliš dlouho.

3.3 Matematické operace s MPT

Funkce pro matematické operace s instancemi `mpt` se nachází v souborech `multiple_precision_operations.c/.h`.

U všech operací je výsledná hodnota zapsána odpovídající funkcí do instance, na kterou ukazuje ukazatel v prvním parametru. Samotné funkce vracejí `int` 1 nebo 0 když nastane chyba nebo nelze operaci provést se zadanými hodnotami. Vstupní operandy zůstávají nezměněné. Na konci každé operace by měla výsledná hodnota vždy být optimalizovaná funkcí `mpt_optimize` tak, aby měla co nejmenší počet segmentů.

Pro některé operace bylo zapotřebí vytvořit navíc operace absolutní hodnoty (funkce `mpt_abs`) a bitového posunu (funkce `mpt_shift`).

Sčítání provádíme po segmentech a stav příznakového bitu `CARRY` zjišťujeme funkcí `addition_carry_`. Při sčítání musíme pro obecnou správnost výpočtu sečíst tolik segmentů, kolik má operand s nejvíce segmenty a poté ještě jeden segment navíc, kdyby došlo k nastavení MSB.

Základní princip implementovaného násobení je ilustrován na obrázku 1. Násobení ve skutečnosti provádíme vždy nad dvojnásobkem segmentů operandu s nejvíce segmenty, aby byly výsledky správné i pro záporná čísla.

Obrázek 1: Ukázka postupu násobení $0b010101 \cdot 0b0101$

3.4 Převod řetězce na MPT

Funkce pro převody řetězců na `mpt` se nachází v souborech `multiple_precision_parsing.c/.h`.

Pro převod používáme hlavně funkci `mpt_parse_str`, která dokáže rozpoznat v jaké soustavě bylo číslo zadáno podle prefixu (`0b` / `0x`) nebo jeho absence. Funkce převádí číslo dokud jsou znaky pro danou soustavu platné a ve chvíli, kdy narazí na znak, který není součástí dané soustavy, ukončí činnost a vrátí hodnotu, kterou se podařilo převést. Během procházení znaků také mění ukazatel zadaný v parametru a na konci převodu bude ukazatel ukazovat na poslední znak, který číslu náležel.

Převod z čísel zadaných v binární soustavě probíhá následovně:

1. Vytvoříme instanci `mpt` s názvem `new` a hodnotou 0
2. Přečteme znak, a pokud nepatří do binární soustavy, ukončíme převod, jinak pokračujeme
3. Hodnotu `new` bitově posuneme o jednu pozici vlevo
4. Nastavíme nultý bit `new` na 1 nebo 0 podle aktuálního znaku
5. Změníme ukazatel na další znak a vrátíme se na krok 2

Převod z čísel zadaných v hexadecimální soustavě probíhá analogicky k binární soustavě s tím rozdílem, že bitový posun vlevo provádíme o čtyři pozice a místo nastavování nultého bitu, přičítáme hodnotu hexadecimálního znaku.

U těchto soustav musíme navíc ještě ošetřit, jestli byla čísla zadána v dvojkovém doplňku a mají být záporná. Např. `0b1101` by takto bylo převedeno na `0b00001101` a je tedy třeba nastavit potřebné bity zleva na jedničku, k čemuž slouží statická funkce `fill_set_bits_`.

Pro převod z dekadické soustavy používáme následující postup:

1. Vytvoříme instanci `mpt` s názvem `new` a hodnotou 0
2. Přečteme znak, a pokud nepatří do dekadické soustavy, ukončíme převod, jinak pokračujeme
3. Hodnotu `new` vynásobíme deseti
4. K `new` přičteme hodnotu znaku v dekadické soustavě.
5. Změníme ukazatel na další znak a vrátíme se na krok 2

3.5 Výpisy MPT

Funkce pro výpisy hodnot `mpt` se nachází v souborech `multiple_precision_printing.c/.h`.

Pro výpis používáme funkci `mpt_print`, která hodnotu vypíše v požadované soustavě.

Výpis v binární soustavě je nejjednodušší:

1. Vypíšeme `'0b'` a hodnotu MSB
2. Ignorujeme všechny bity před (s nižší vahou) MSB, které mají stejnou hodnotu jako MSB
3. Zbytek bitů vypíšeme v pořadí od nejvyššího po nultý

Výpis v hexadecimální soustavě provádíme po nibblech (čtveřice bitů) od nejvyššího nibblu po nultý.

1. Vypíšeme '0x'
2. Ignorujeme všechny nibbly s hodnotu 0xf (když MSB = 1) nebo 0x0 (když MSB = 0)
3. Pokud MSB = 0 a aktuální nibble ≥ 8 , vypíšeme '0', jinak další krok
4. Pokud MSB = 1 a aktuální nibble < 8 , vypíšeme '0', jinak další krok
5. Zbytek nibblů vypíšeme v pořadí od nejvyššího po nultý

Pro výpis v dekadické soustavě si nejprve vytvoříme absolutní hodnotu zadané instance `mpt`. Tu postupně dělíme deseti, kde zbytek po dělení nám postupně dá jednotlivé číslice v dekadické podobě. Číslice takto ale získáváme pozpátku, proto je nejprve ukládáme do vektoru, jehož obsah nakonec pozpátku vypíšeme, ještě před vypsáním vektoru ale vypíšeme znak mínusu, pokud bylo původní číslo záporné.

3.6 Operátory

Podporované operátory a k nim přidružené obslužné funkce, precedence a asociativity jsou ve struktuře `func_oper_type` v souborech `operators.c/.h`, spolu s funkcí `get_func_operator` pro získání struktury `func_oper_type` s daty, odpovídajícími požadovanému znaku operátoru.

3.7 Shunting yard

Funkce pro zpracování zadaných výrazů se nachází v souborech `shunting_yard.c/.h`.

Shunting yard je implementován standardně podle článku [6] a jeho činnost voláme funkcí `shunt` s tím, že po úspěšném provedení algoritmu nejsou čísla ve výsledné RPN formě zapsána přímo, ale jsou reprezentována znakem 'n' a jejich skutečné `mpt` hodnoty jsou uloženy ve výsledném zásobníku `values`. Při vyhodnocování RPN pak při každém přečtení znaku 'n' skutečnou hodnotu čísla z tohoto zásobníku odebereme.

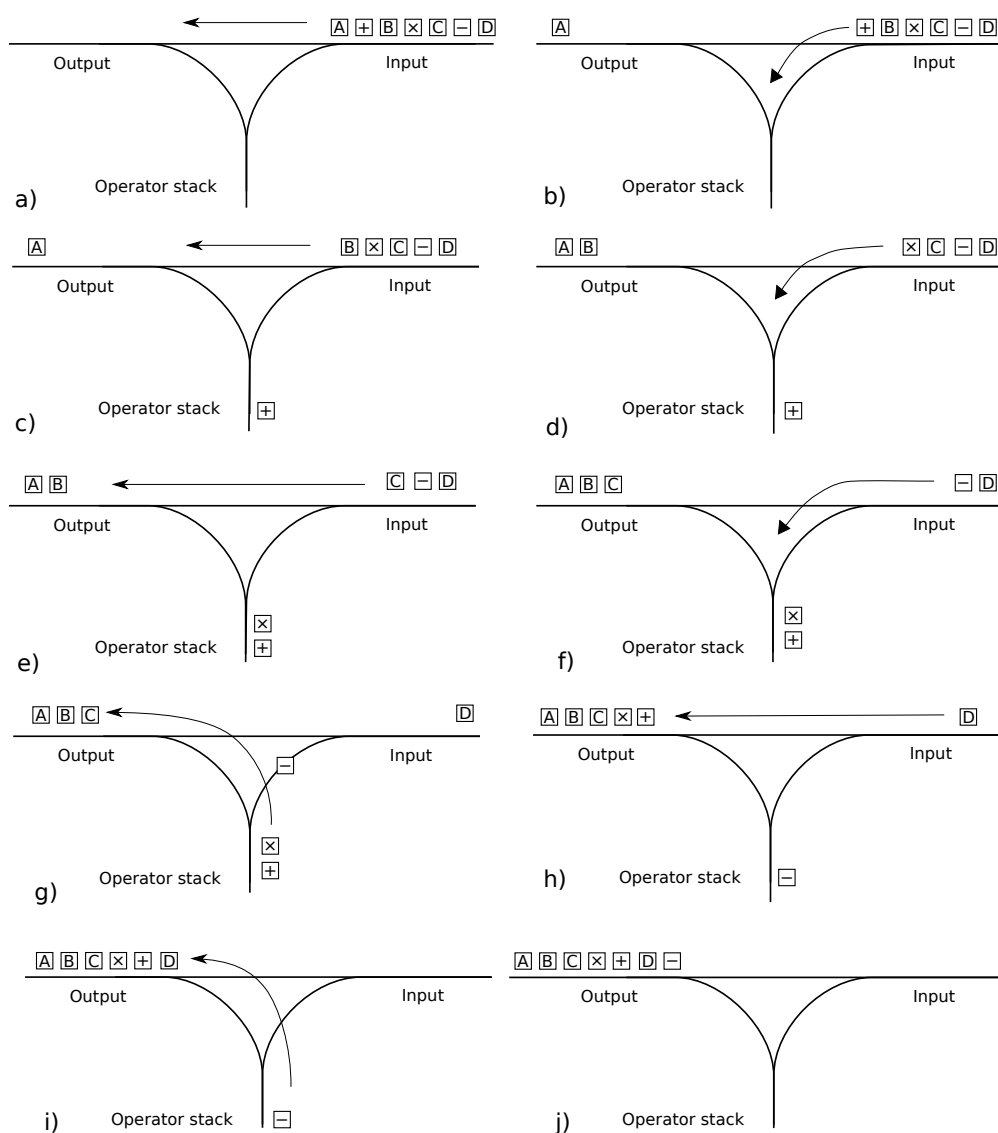
Existuje speciální případ infixové formy, kde se, i přes správnou precedenci a asociativitu (podle tab. 1), RPN forma vytvoří v nesprávném tvaru, a to umocnění základu zápornou hodnotou např. ve tvaru '4^-2'. Správná forma RPN by byla '42-^', s naším algoritmem by ale výsledná RPN forma byla '4^2-', která by při vyhodnocování výsledku vrátila chybu. Z toho důvodu je tento speciální případ ošetřen ve statické funkci `shunt_minus_` tak, že se operátor mínusu vynechá, zpracuje se výraz za mínusem a minus se přidá do výsledné RPN formy naposled.

3.8 Vyhodnocení RPN

Funkce pro vyhodnocení RPN se nachází v souborech `shunting_yard.c/.h`.

Vyhodnocení výrazů ve formě RPN je také implementováno standardně. V algoritmu se využívá zásobník `stack_values` pro mezivýsledky. Znaky výrazu zpracováváme jeden po druhém. Pokud narazíme na znak 'n', do `stack_values` vložíme hodnotu `mpt` odebranou ze zásobníku `values`. Pokud je znak operátor, provedeme odpovídající operaci nad hodnotou, resp. hodnotami odstraněnými ze zásobníku `stack_values` a výsledek do stejného zásobníku vložíme.

Na konci vyhodnocení by se v zásobníku `stack_values` měla nacházet jediná (výsledná) hodnota, a pokud tomu tak není, obsahoval výraz syntaktickou chybu.



Obrázek 3: Ilustrace postupu algoritmu Shunting yard (převzato z [6]).

4 Uživatelská příručka

4.1 Přeložení a sestavení programu

Postup pro přeložení a sestavení je na všech platformách stejný. Je nutné mít nainstalovaný překladač `gcc` a nástroj `make` a mít zprovozněné jejich ekvivalentní příkazy v konzoli. Samotný překlad a sestavení provedeme v příkazovém řádku v kořenové složce programu příkazem `make`. Po úspěšném sestavení se ve stejném adresáři vytvoří spustitelný binární soubor `calc.exe`.

```

~/Stažené/Semestralka > ls
build data doc src CMakeLists.txt Makefile Makefile.win
~/Stažené/Semestralka > make &>/dev/null && ls
build data doc src calc.exe CMakeLists.txt Makefile Makefile.win
~/Stažené/Semestralka >

```

Obrázek 4: Ukázka překladač a sestavení programu

4.2 Spuštění a ovládání programu

Program spouštíme, podle zadání, dvěma způsoby.

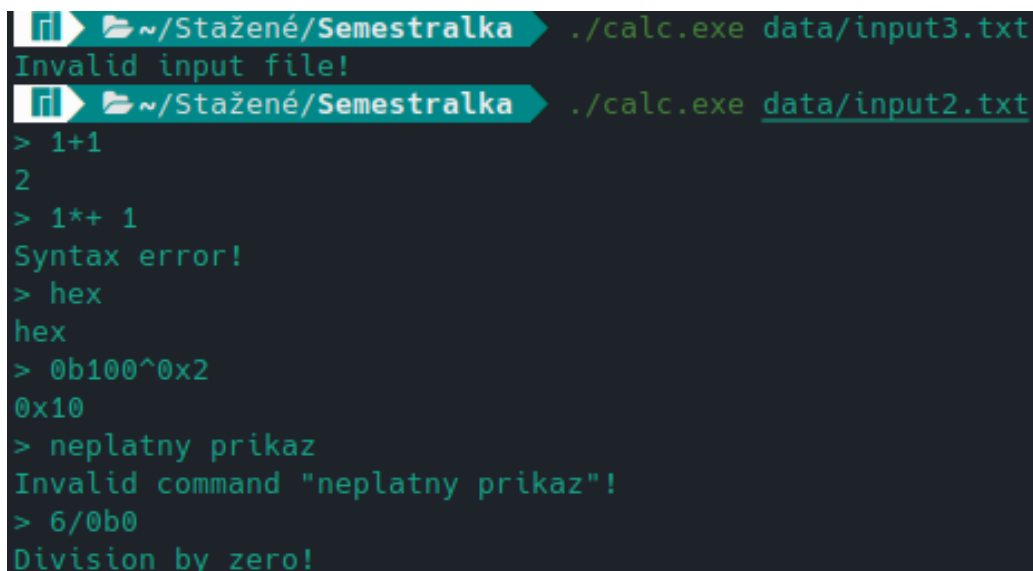
Příkazem `calc.exe` program otevřeme v interaktivním režimu (viz obr. 5) pro postupné zadávání příkazů nebo infixových výrazů z příkazové řádky. Zadáním nepodporovaného příkazu nebo výrazu se špatnou syntaxí vypíše program odpovídající chybovou hlášku a čeká na nový příkaz. Příkazem `quit` program ukončíme.

Příkazem `calc.exe [<vstupni-soubor>]` program načte vstupní soubor, každý řádek v něm zpracuje jako zadaný výraz, řádek vypíše jako bychom ho zadali v interaktivním režimu a následně vypíše výsledek zpracování (viz obr. 6). Po zpracování všech řádků v souboru se program ukončí s návratovou hodnotou `EXIT_SUCCESS`. Pokud zadaný soubor neexistuje vypíše chybovou hlášku a skončí s návratovou hodnotou `EXIT_FAILURE`.



```
~/Stažené/Semestralka > ./calc.exe
> 1+1
2
> 0b0111^0x2
49
> out
dec
> some_missing_command
Invalid command "some_missing_command"!
> quit
```

Obrázek 5: Ukázka interaktivního režimu



```
~/Stažené/Semestralka > ./calc.exe data/input3.txt
Invalid input file!
~/Stažené/Semestralka > ./calc.exe data/input2.txt
> 1+1
2
> 1*+ 1
Syntax error!
> hex
hex
> 0b100^0x2
0x10
> neplatny prikaz
Invalid command "neplatny prikaz"!
> 6/0b0
Division by zero!
```

Obrázek 6: Ukázka načtení a zpracování souboru

5 Závěr

Vytvořená kalkulačka dokáže provádět všechny požadované operace, často i za relativně krátkou dobu běhu. Výrazy v infixové formě zpracovat umí a během testování nebyl nalezen žádný výraz, který by se vyhodnotil nesprávně. Speciální případ výrazu s operátorem mocniny, po kterém následuje operátor negace, je ošetřen, správnost tohoto ošetření nemusí být ale jednoznačná pro některé složitější výrazy. Implementovány jsou jak režim interaktivní, tak režim zpracování souboru, včetně chybových hlášení nepodporovaných příkazů, či nedefinovaných matematických operací. ‘Nekonečně velká’ čísla sice kalkulačka zpracovat neumí, čísla ale mohou být tak velká, že je lze označit za ‘prakticky nekonečně velká’.

Jedním z problémů k vyřešení byla příliš dlouhá doba kontroly programem `valgrind`, která byla způsobena zbytečnými dynamickými alokacemi a realokacemi struktury `mpt` a bylo nutné celý program přepsat tak, aby se struktury alokovaly na zásobníku.

Rychlost vyhodnocování některých výrazů může u větších čísel být relativně malá a určitě existují algoritmy, které by byly pro provedení vybraných matematických operací lepší. Právě takové algoritmy pravděpodobně využívá např. knihovna *GMP*, která je lepší alternativou našeho řešení.

Odkazy

- [1] Úplné zadání [Online]
<https://www.kiv.zcu.cz/studies/predmety/pc/data/works/sw2022-03.pdf>
- [2] Bitové pole [Online]
https://en.wikipedia.org/wiki/Bit_array
- [3] Binární sčítačka [Online]
https://cs.wikipedia.org/wiki/Bin%C3%A1rn%C3%AD_s%C4%8D%C3%ADta%C4%8Dka
- [4] Algoritmus binárního umocňování [Online]
https://cs.wikipedia.org/wiki/Algoritmus_bin%C3%A1rn%C3%ADho_umoc%C5%88ov%C3%A1n%C3%AD
- [5] Reverzní Polská notace [Online]
https://en.wikipedia.org/wiki/Reverse_Polish_notation
- [6] Algoritmus Shunting yard [Online]
https://en.wikipedia.org/wiki/Shunting_yard_algorithm