ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA APLIKOVANÝCH VĚD
KATEDRA INFORMATIKY A VÝPOČETNÍ TECHNIKY

Dokumentace semestrální práce

Předmět: Programování v jazyce C (KIV/PC)

BigInt Calculator

Aritmetika s libovolnou přesností

Autor: Jáchym Nácovský
Osobní číslo: A25B0003P
Datum: 2. ledna 2026

Obsah

1 Zadání **2**
1.1 Požadavky na vstupní data . . . 2
1.2 Režimy spuštění . . . 2
1.3 Řídící příkazy . . . 2

2 Analýza úlohy **3**
2.1 Reprezentace velkých čísel . . . 3
2.2 Volba reprezentace a práce se znaménkem . . . 3
2.3 Analýza aritmetických algoritmů . . . 3
2.4 Zpracování výrazů . . . 4

3 Popis implementace **5**
3.1 Sčítání a odčítání . . . 6
3.2 Násobení . . . 7
3.3 Dělení a modulo . . . 8
3.4 Faktoriál . . . 9
3.5 Parser a syntaktická analýza . . . 9
3.6 Správa chyb a mezivýpočty . . . 9
3.7 Optimalizace . . . 10

4 Uživatelská příručka **11**
4.1 Překlad programu . . . 11
4.2 Ovládání . . . 11
4.3 Podporované operátory . . . 11

5 Testování a validace **12**
5.1 Jednotkové testy . . . 12
5.2 Kontrola paměti . . . 12

6 Závěr **12**

7 Zdroje **13**

1 Zadání

Cílem semestrální práce bylo vytvořit konzolovou aplikaci, která umožňuje výpočty s libovolně velkými celočíselnými hodnotami. Program musí podporovat různé číselné soustavy (binární, desítkovou, hexadecimální) a interpretovat aritmetické výrazy v infixovém zápisu.

1.1 Požadavky na vstupní data

Program musí správně rozpoznat čísla s prefixy `0b` pro binární, `0x` pro hexadecimální a bez prefixu pro desítkovou soustavu. Nezáleží na velikosti písmen v hexadecimálních číslech ani příkazech.

1.2 Režimy spuštění

- **Interaktivní režim** – zadávání výrazů přímo v konzoli, ukončení příkazem `quit`.
- **Dávkový režim** – načítání vstupu ze souboru řádek po řádku.

1.3 Řídící příkazy

Program musí umožnit změnu výstupní číselné soustavy a poskytovat kontrolní příkazy, viz tabulka níže.

Tabulka 1: Řídící příkazy interpretu

Příkaz	Popis činnosti
`dec`	Výstup v desítkové soustavě
`bin`	Výstup v binární soustavě (doplněk)
`hex`	Výstup v hexadecimální soustavě (doplněk)
`out`	Zobrazení aktuálního nastavení
`quit`	Ukončení programu

2 Analýza úlohy

Řešení bylo rozděleno na tři oblasti: reprezentace čísel, aritmetické algoritmy a syntaktická analýza výrazů. Cílem je, aby program zvládl libovolně dlouhá čísla bez ztráty přesnosti a aby algoritmy byly efektivní.

2.1 Reprezentace velkých čísel

Interně jsou čísla uložena jako dynamické pole 32-bitových slov `digits[]` a samostatné znaménko. Tato reprezentace je nezávislá na vstupní soustavě a usnadňuje implementaci aritmetiky.

Obrázek 1: Interní reprezentace čísla BigInt.

2.2 Volba reprezentace a práce se znaménkem

Při návrhu vnitřní reprezentace velkých čísel byly zvažovány dvě hlavní koncepce:

1. **Reprezentace v desítkové soustavě:** Uložení hodnot 0 až 10^9 do každého prvku pole. Tato varianta by sice usnadnila vstupně-výstupní operace, ale za cenu nižšího výkonu při aritmetických výpočtech.
2. **Binární reprezentace (zvolená):** Využití plné šířky 32bitového slova (`uint32_t`).

Binární varianta byla upřednostněna z důvodu **efektivnějšího využití systémových prostředků**. Tato reprezentace dovoluje využít nativní 64bitové mezivýpočty pro zpracování přenosu (*carry*), což je pro operace násobení a dělení řádově efektivnější než opakované výpočty modulo u desítkové reprezentace.

Rozhodnutí o oddělení znaménka od absolutní hodnoty čísla bylo zvoleno, protože:

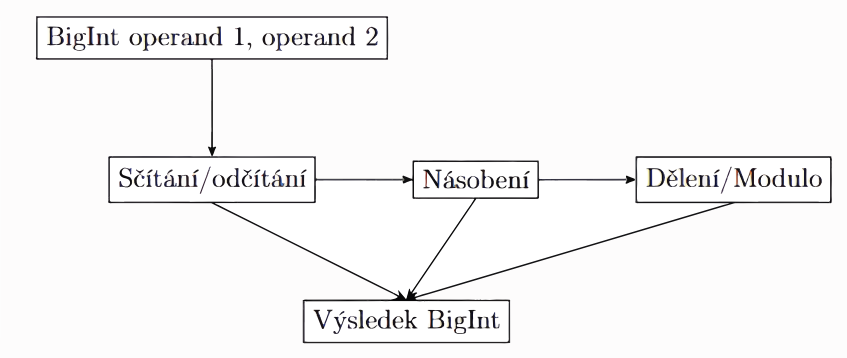
- usnadňuje implementaci aritmetických operací,
- zjednodušuje převody mezi číselnými soustavami,
- minimalizuje nutnost zpracovávat doplňkový kód během výpočtů.

2.3 Analýza aritmetických algoritmů

Sčítání a odčítání: po slovech s přenosem, při rozdílných znaménkách odčítání absolutních hodnot. Mezivýpočty používají 64-bitový typ `uint64_t`.

Násobení: školní algoritmus $O(N \cdot M)$, 64-bit mezivýpočty pro zabránění přetečení. Rychlejší algoritmy (Karatsuba) jsou jen zmíněny.

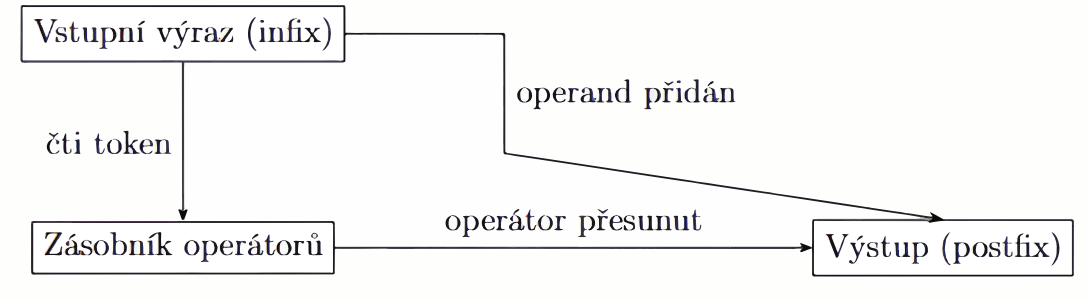
Dělení a modulo: metoda „Bitwise Long Division“ pro podíl a zbytek. Efektivní pro velká čísla.



Obrázek 2: Schéma aritmetických operací nad BigInt.

2.4 Zpracování výrazů

Pro syntaktickou analýzu byl zvolen Shunting-yard algoritmus, převádějící infix výrazy do postfixové notace (RPN). Následně se vyhodnocují pomocí zásobníku operandů.



Obrázek 3: Shunting-yard algoritmus.

3 Popis implementace

Program je rozdělen do modulů:

- `main.c` – hlavní program, zpracovává vstup, volá parser a aritmetické funkce,
- `parser.c` – převod infixového zápisu na postfix, kontrola syntaxe a tokenizace,
- `stack.c` – implementace zásobníku operandů a operátorů pro RPN,
- `bigint.c` – aritmetika s libovolně velkými čísly (sčítání, odčítání, násobení, dělení, modulo, faktoriál).

Paměť je alokována dynamicky a spravována funkcemi `malloc`, `realloc` a `free`. Struktura `BigInt` obsahuje pole `digits[]` a samostatné znaménko.

3.1 Sčítání a odčítání

Sčítání se provádí po jednotlivých 32-bitových slovech s přenosem do vyšších řádů. Při rozdílných znaménkách operandů se provádí odčítání absolutních hodnot, přičemž výsledku je přiřazeno správné znaménko podle velikosti operandů. Mezivýpočty používají 64-bitový typ `uint64_t` pro bezpečné zachycení přetečení.

```
BigInt* bi_add(const BigInt* a, const BigInt* b)
{
    int cmp;
    BigInt* result;

    if (!a || !b) return NULL;

    if (a->sign == 0) return bi_copy(b);
    if (b->sign == 0) return bi_copy(a);

    if (a->sign == b->sign)
    {
        result = bi_add_abs(a, b);
        if (!result) return NULL;
        result->sign = a->sign;
        return result;
    }

    cmp = bi_compare_abs(a, b);

    if (cmp == 0)
    {
        return bi_create();
    }

    if (cmp > 0)
    {
        result = bi_sub_abs(a, b);
        if (!result) return NULL;
        result->sign = a->sign;
    }
    else
    {
        result = bi_sub_abs(b, a);
        if (!result) return NULL;
        result->sign = b->sign;
    }

    return result;
}
```

Listing 1: Funkce pro sčítání BigInt

3.2 Násobení

Násobení využívá klasický algoritmus $O(N \cdot M)$. Každý součin slov je přičten k odpovídající pozici ve výsledném poli. Přenos do vyšších řádů je správně zpracován, čímž se zabraňuje přetečení.

```
BigInt* bi_mul(const BigInt* a, const BigInt* b)
{
    ...

    memset(res->digits, 0, total_len * sizeof(uint32_t));
    res->length = total_len;

    for (i = 0; i < a->length; i++)
    {
        carry = 0;
        for (j = 0; j < b->length; j++)
        {
            current = (uint64_t)a->digits[i] * b->digits[j] +
                res->digits[i + j] + carry;
            res->digits[i + j] = (uint32_t)current;
            carry = current >> BITS_IN_WORD;
        }

        if (carry)
        {
            sum = (uint64_t)res->digits[i + b->length] + carry;
            res->digits[i + b->length] = (uint32_t)sum;

            k = i + b->length + 1;
            ripple = sum >> BITS_IN_WORD;
            while (ripple > 0 && k < res->length)
            {
                sum = (uint64_t)res->digits[k] + ripple;
                res->digits[k] = (uint32_t)sum;
                ripple = sum >> BITS_IN_WORD;
                k++;
            }
        }
    }
    ...
}
```

Listing 2: Funkce pro násobení BigInt

3.3 Dělení a modulo

Dělení využívá metodu *Bitwise Long Division*. Dělitel se zarovná s nejvyšším bitovým řádem dělence a postupně se odečítá. Výstupem je současně podíl i zbytek.

```
BigInt* bi_div(const BigInt* a, const BigInt* b)
{
    BigInt *q, *r;

    if (!a || !b) return NULL;
    if (b->sign == 0) return NULL;

    bi_div_mod_abs(a, b, &q, &r);
    bi_destroy(r);

    if (q)
    {
        if (a->sign == b->sign)
        {
            q->sign = 1;
        }
        else
        {
            q->sign = -1;
        }
        bi_normalize(q);
    }
    return q;
}
```

Listing 3: Funkce pro dělení BigInt

3.4 Faktoriál

Faktoriál se počítá iterativně a využívá již implementovanou násobicí funkci. Program detekuje záporné vstupy a vypisuje chybové hlášení.

```
BigInt* bi_fact(uint32_t n)
{
    BigInt* res;
    uint32_t i;

    if (n == 0 || n == 1)
    {
        return bi_from_dec("1");
    }
    res = bi_create();
    res->digits[0] = 1;
    res->sign = 1;

    for (i = 2; i <= n; i++)
    {
        bi_mul_digit_into(res, i);
    }

    return res;
}
```

Listing 4: Výpočet faktoriálu BigInt

3.5 Parser a syntaktická analýza

Parser převádí infixový výraz na postfix (RPN) a kontroluje syntaxi. Každý token je analyzován:

- čísla se převádějí do interní reprezentace BigInt,
- operátory se řadí podle priority a asociativity do zásobníku,
- závorky jsou správně párovány a chybné uzávorkování vyvolá hlášení.

3.6 Správa chyb a mezivýpočty

Program detekuje chybové stavy:

- dělení nulou,
- přetečení při násobení/faktoriálu,
- záporný faktoriál.

Mezivýpočty jsou prováděny 64-bitovou aritmetikou a výsledky se postupně zapisují do 32-bitových slov.

3.7 Optimalizace

Pro zvýšení výkonu byla implementována:

- dynamická alokace paměti pouze dle potřeby,
- odklad přenosů a meziproměnných výsledků, aby se minimalizovaly kopie,
- kontrola nulových vyšších řádů pro zkrácení pole BigInt.

4 Uživatelská příručka

4.1 Překlad programu

Linux/Unix nebo WSL:

```
make
```

Vyčištění objektových souborů:

```
make clean
```

Windows (MinGW):

```
mingw32-make -f Makefile.win
```

4.2 Ovládání

Výrazy lze zadávat např.:

```
10 + 0x0A * 0b10
```

Tabulka 2: Příklad převodu mezi soustavami

Desítková	Binární	Hexadecimální
42	0b101010	0x2A
255	0b11111111	0xFF
1024	0b10000000000	0x400

Přepínání soustav: `dec`, `bin`, `hex`, zobrazení nastavení: `out`. Ukončení: `quit`.

4.3 Podporované operátory

Tabulka 3: Operátory

Znak	Význam	Priorita	Arita	Asociativita
!	Faktoriál	4	unární	levá
^	Umocnění	4	binární	pravá
-	Unární minus	3	unární	pravá
*	Násobení	2	binární	levá
/	Celočíselné dělení	2	binární	levá
%	Modulo	2	binární	levá
+	Sčítání	1	binární	levá
-	Odčítání	1	binární	levá

5 Testování a validace

5.1 Jednotkové testy

Testy funkcí z modulu `bigint.c` kontrolují správnost operací a chybové stavy. Dělení nulou vyvolá hlášení `Division by zero!`. Faktoriál 1000! byl vypočten v čase pod 100 ms.

Tabulka 4: Příklad chybových stavů operátorů

Operace	**Vstup**	**Očekávaný výstup / Chyba**
Dělení nulou	123 / 0	Division by zero!
Přetečení	$2^{1024} * 2^{1024}$	Správně spočítá BigInt
Faktoriál záporného čísla	`-5!`	Invalid input for factorial

5.2 Kontrola paměti

Valgrind potvrzuje, že veškerá dynamická paměť je korektně uvolněna.

6 Závěr

Cílem práce bylo navržení a implementace kalkulačky pro práci s celými čísly s libovolnou přesností. Zadání bylo splněno v plném rozsahu – výsledná aplikace spolehlivě interpretuje infixové výrazy, podporuje práci ve třech číselných soustavách (binární, desítkové a šestnáctkové) a korektně ošetřuje uživatelem zadané chybové stavy.

Během vývoje se jako klíčová ukázala volba reprezentace čísel pomocí pole 32bitových slov, která umožnila efektivní implementaci aritmetických operací s využitím nativní šířky registrů procesoru. I přes drobné odchýlení od původně zamýšleného standardu ANSI C směrem k modernějšímu **C99** zůstala zachována plná přenositelnost kódu mezi systémy Windows a Linux.

Během testování byla ověřena stabilita výpočtů i pro náročné vstupy (např. výpočet 10 000! proběhl bez znatelné prodlevy), což potvrzuje správnost zvolených algoritmů a efektivní správu paměti bez úniků. Možné budoucí rozšíření by se mohlo zaměřit na optimalizaci násobení pro extrémně velké operandy.

7 Zdroje

V průběhu vývoje a při návrhu algoritmů byly využity následující zdroje:

1. **W3Schools C Tutorial**
   Základy jazyka C, práce s ukazateli, dynamická alokace paměti.

2. **Brilliant.org – Shunting-yard Algorithm**
   Detailní popis a vysvětlení algoritmu pro převod infixového zápisu na postfix.

3. **FreeCodeCamp – Exponentiation by Squaring**
   Algoritmus pro efektivní umocňování velkých čísel.

4. **GeeksforGeeks – Big Integer Division**
   Popis algoritmu dělení pro velmi velká čísla (bitová dlouhá dělení).

5. **The C99 Standard (ISO/IEC 9899:1999)**
   Specifikace jazyka C99, použitá pro práci s typy jako `uint32_t` a `uint64_t`.

6. **Valgrind Documentation**
   Nástroj pro detekci úniků paměti a profiling.