Dokumentacja Projektu Kalkulator

Paweł Bielecki

12 listopada 2023

Spis treści

1 Założenia projektu

Celem projektu jest zaimplementowanie kalkulatora wykonującego działania na dowolnie dużych liczbach, w systemach liczbowych od 2 do 16.

Obsługiwane działania to:

- dodawanie
- mnożenie
- potęgowanie
- dzielenie całkowitoliczbowe
- dzielenie modulo
- konwersja systemu liczbowego

1.1 Używanie programu

Program wczytuje działania z pliku tekstowego i wypisuje wyniki do innego pliku.

Uruchamiany poprzez:

```
$ calc.exe <plik wejściowey> [plik wyjściowy]
```

Bez podania pliku wyjściowego program tworzy plik wyjściowy z nazwy pliku wejściowego.

Dodatkowo program można uruchomić bez argumentów, aby wejść w tryb interaktywny.

```
$ calc.exe
calc.exe repl
 Aby zakończyć: ctrl-d (ctrl-z na windowsie)

 [1]> + 10
 [2]> 15
 [3]> 20
 [4]> Wynik: 35
```

W przypadku podania więcej, niż 2 argumentów program wypisuje instrukcję używania.

```
$ calc.exe a b c
Użycie:
    calc.exe <ścieżka do pliku wejściowego> [ścieżka do
↪ pliku wyjściowego]

Plik wejściowy musi być w formacie `*.txt`
Plik wyjściowy jest opcjonalny (zostanie stworzony na
↪ podstawie nazwy pliku wejściowego `out_*.txt`)
```

1.2 Obsługiwanie błędów

Program wypisuje 3 rodzaje informacji dla użytkownika:

- *info* - informacja pomocnicza, program zadziałał poprawnie
- *warning* - dane działanie wykonało się, ale ze skutkiem możliwie niepożądanym
- *error* - działanie nie zostało wykonane - w takim wypadku program wypisuje do pliku wyjściowego jedynie działanie i argumenty, i przechodzi do kolejnego działania

Przykłady:

```
$ calc.exe add.txt
INFO: Nie podano pliku wyjściowego, stworzono `out_add.txt`
```

Jeżeli podamy za mało argumentów:

```
WARNING [7:1]: Operacja przyjmuje przynajmniej 2 argumenty,
↪ otrzymano 1
    + 10
    ^
```

W nawiasach kwadratowych mamy numer linii oraz number kolumny, w której wystąpił błąd w pliku wejściowym.

```
ERROR [1:4]: Baza może być tylko z przedziału [2, 16]
    + 22
      ^
```

Obsługiwane błędy:

- Nie można otworzyć pliku wejściowego
- Nie podano nazwy operacji (plik wejściowy zaczyna się od argumentu)
- Podano zbyt mało argumentów
- Nieoczekiwany znak
- Nieprawidłowy znak w liczbie o bazie x (np. 123 nie jest poprawną liczbą w systemie dwójkowym)
- Przekroczona długość argumentu (40 znaków)
- Baza nie jest z przedziału [2, 16]
- Dzielenie przez 0

2 Narzędzia

Program został napisany w języku C, kompilowany w *gcc*.

```
$ gcc -O2 -pedantic -Wall -Wextra
```

Używany edytor tekstu to *neovim* (na linuxie) oraz *visual studio code* (na windowsie), *clang-format* do formatowania kodu (z ustawionymi 4 spacajmi indentacji zamiast 2). Poza tym również *git* jako system kontroli wersji oraz proste skrypty w *bash*-u do testowania.

Dokumentacja została napisana w *latex*-u z wykorzystaniem biblioteki *minted* do wyświetlania kodu.

3 Ogólny opis rozwiązania

Kod jest podzielony na 5 modułów.

```c
// scanner.h

#define ARG_SIZE 41
#define LINE_SIZE 256

typedef struct {
    char *line;
    unsigned int line_idx;
    unsigned int idx;
} scanner;

void parse_int(scanner *scanner, char *output, unsigned int len,
               unsigned short base, bool *ok);
oper read_instruction(scanner *scanner, bool *ok);
void read_arg(scanner *scanner, char *output, unsigned short base, bool *ok);
bool is_argument(scanner *scanner, char *line, bool *ok);

char advance(scanner *scanner);
void consume(scanner *scanner, char c);
void consume_spaces(scanner *scanner);
```

Scanner będzie przechodził po danej linii znak po znaku i próbował ją zinterpretować jako argument lub nazwę operacji. Będzie również wykrywał błędne formatowanie liczb (np. użycie znaku 'd' w liczbie o podstawie 11).

Sam scanner jest structem zawierającym daną linię pliku wejściowego oraz jej numer i swoją pozycję (wskazującą na następny znak do przeczytania).

Dwie najważniejsze funkcje to `read_arg`, która sczytuje argument i zapisuje go w podanym argumencie `char *output` oraz `read_instruction`, która zwraca struct `oper` reprezentujący typ operacji oraz jej bazę.

```
// operation.h

enum op_type {
    Add,
    Mul,
    Div,
    Mod,
    Pow,
    Con,
};

typedef struct {
    enum op_type op_type;
    unsigned short base;
} oper;
```

Po odczytaniu operacji i argumentów działania przekazywane są do wykonania:

```
// execute.h

dynStr execute(scanner *scanner, oper op, char *arg1, char
  ↪  *arg2, bool *ok);
```

Funkcja ta, patrząc na typ operacji rozdziela wykonanie do poszczególnych funkcji działań.

```
// execute.c

dynStr execute(scanner *scanner, oper op, char *arg1, char
  ↪  *arg2, bool *ok) {
    switch (op.op_type) {
    case Add:
        return exec_add(op.base, arg1, arg2);
    case Mul:
        return exec_mul(op.base, arg1, arg2);
    case Pow:
        return exec_pow(op.base, arg1, arg2);
    case Div:
```

```
12          return exec_div(scanner, op.base, arg1, arg2, NULL,
    ↪  ok);
13      case Mod:
14          return exec_mod(scanner, op.base, arg1, arg2, ok);
15      case Con:
16          return exec_convert(scanner, op.base, arg2);
17      default:
18          assert("Unreachable" && 0);
19      }
20  }
```

Funkcje te zwracają typ `dynStr`, czyli dynamicznie alokowaną tablicę znaków, zdefiniowaną w `helpers.h` razem z innymi przydatnymi operacjami.

```
1  // helpers.h
2  #define dbg(x, y) fprintf(stderr, "%d:" #y " = " x "\n",
    ↪  __LINE__, y);
3
4  typedef struct {
5      char *data;
6      size_t size;
7      size_t capacity;
8  } dynStr;
9
10 void init_dynStr(dynStr *str);
11 void dynStr_from(dynStr *str, char *s);
12 void append_char(dynStr *str, char c);
13 void free_dynStr(dynStr *str);
14
15 void print_help(char *name);
16 void reverse(char *str);
17 bool is_digit(char c, unsigned short base);
18 bool is_unknown_char(scanner *scanner, char c);
19 int char_to_dec(char c, short base, bool *ok);
20 char int_to_char(int v);
21 void trim_trailing(dynStr *num, char c);
22 void trim_leading(dynStr *num, char c);
23 const char *extract_name(const char *path);
```

Operacje które mogą napotkać błędne wejście posługują się funkcją `report`, która ma wypisać informację o błędzie i jego lokalizacji.

```
// report.h

typedef enum {
    error,
    warning,
    info,
} er_type;

void report(scanner *scanner, er_type t, char *msg, ...);
```

Program ma początek oczywiście w `main.c` gdzie sczytujemy od użytkownika nazwy plików i w pętli czytamy kolejne linie wejścia i wypisujemy rezultaty.

Zatem podsumowując, ogólna struktura kodu:
`main`: pobiera input, przechodzi przez każdą linię → `scanner`: przechodzi przez każdy znak linii → `execute`: wykonuje działania → `main`: wypisuje output.

4 Szczegóły

4.1 Parsowanie argumentów programu

Tak jak było powiedziane w pierwszej części, porgram może przyjąć 0, 1 lub 2 argumenty, zatem dla większej ilości chcemy wypisać tekst pomocniczy.

```
// main.c

if (argc > 3) {
    print_help(argv[0]);
    exit(1);
}
```

Warto zaznaczyć, że `argv[0]` to zawsze nazwa programu, stąd sprawdzamy `argc > 3`.

Jeżeli nie podano żadnego argumentu plik wejściowy i wyjściowy, to odpowiednio standardowe wejście i wyjście.

```c
// main.c

FILE *in_file;
FILE *out_file;
bool is_repl = false;
if (argc == 1) {
    in_file = stdin;
    out_file = stdout;
    is_repl = true;
} else {
    ...
}
```

W przeciwnym razie otwieramy plik wejściowy i tworzymy/otwieramy wyjściowy.

```c
// main.c

else {
    in_file = fopen(argv[1], "r");
    if (in_file == NULL) {
        report(NULL, error, "Nie można otworzyć pliku `%s`\n",
    ↪  argv[1]);
        exit(1);
    }

    char *out_name;
    if (argc == 3) {
        out_name = argv[2];
    } else {
        char o_name[100] = "out_";
        out_name = strcat(o_name, extract_name(argv[1]));
        out_name = strcat(out_name, ".txt");
        report(NULL, info, "Nie podano pliku wyjściowego,
    ↪  stworzono `%s`\n",
                out_name);
    }
    out_file = fopen(out_name, "w");
}
```

Jeżeli nie podano pliku wyjściowego musimy z nazwy wejściowego utworzyć nazwę wyjściowego:

```
// helpers.c

const char *extract_name(const char *path) {
    const char *filename;

    char slash = '/';
#ifdef _WIN32
    slash = '\\';
#endif
    const char *last_slash = strrchr(path, slash);

    if (last_slash != NULL) {
        filename = last_slash + 1;
    } else {
        filename = path;
    }

    const char *last_dot = strrchr(filename, '.');

    if (last_dot != NULL) {
        size_t len = last_dot - filename;
        char *new_str = (char *)malloc(sizeof(char) * (len +
↪   1));
        strncpy(new_str, filename, len);
        new_str[len] = '\0';
        return strdup(new_str);
    } else {
        return strdup(filename);
    }
}
```

Najpierw odcinamy od ostatniego ’/’ (na windowsie ‘\‘) potem do ostatniej ’.’, aby dostać nazwę pliku.

4.2 Czytanie wejścia

Przed pętlą deklarujemy pewne zmienne przechowujące stan programu:

```
// main.c

dynStr result;
oper op;
bool ok = true;
unsigned int line_idx = 0;
char buffer[LINE_SIZE];

// do raportowania błędów
char op_buffer[LINE_SIZE];
int op_idx = 0;
int arg_count = 0;
int expected_args = 2;
```

4 ostatnie nie są niezbędne do funkcjonowania programu, lecz dają cenne informacje do bardziej precyzyjnego raportowania błędów. Będzie to często powtarzający się scenariusz - dla lepszych błędów musimy poświęcić nieco prostoty kodu (np. funkcje będą musiały przyjmować 1-2 argumenty więcej, aby dostać informacje do raportowania błędów). Mimo to uważam, że dobre błędy są warte lekkiego pogorszenia prostoty implementacji (np. pomocne wiadomości o błędzie / warningi są dla mnie największą zaletą danego kompilatora).

Przed pętlą wypisujemy jeszcze wiadomość dla trybu interaktywnego.

```
// main.c

if (is_repl)
    printf("%s repl\nAby zakończyć: ctrl-d (ctrl-z na
    ↪ windowsie)\n\n",
           argv[0]);

while (repl_prompt(line_idx, is_repl),
       fgets(buffer, sizeof(buffer), in_file) != NULL) {
    ...
}
```

Pętla wykonuje się, dopóki `fgets` nie zwróci `NULL` (w przypadku błędu lub końca pliku). Przed każdą iteracją wypisujemy również prostego prompta, jeżeli jesteśmy w repl:

```
// main.c

void repl_prompt(int line_idx, bool is_repl) {
    if (is_repl)
        printf("[%d]> ", line_idx + 1);
}
```

Będziemy omijać wszystkie puste linie oraz zaczynające się na `'#'` (implementując w ten sposób możliwość komentarzy, przydatnych w debugowaniu).

```
// main.c

if (buffer[0] == '\n' || buffer[0] == '#') {
    continue;
}

for (int i = 0; buffer[i]; i++) {
    buffer[i] = toupper(buffer[i]);
}
```

Przy okazji zamieniamy wszystkie znaki na wielkie litery (aby nie odróżniać potem `'A'` od `'a'` w liczbach).

Następnie inicjujemy scanner.

```
// main.c

scanner scanner;
scanner.idx = 0;
scanner.line_idx = line_idx;
scanner.line = buffer;
```

Następnie będziemy potrzebowali dowiedzieć się, czy linia, którą właśnie zczytaliśmy jest argumentem (pojedynczą liczbą) czy nazwą operacji (czymś formatu [znak] [liczba] lub [liczba] [liczba]).

```
// scanner.c

bool is_argument(scanner *scanner, char *line, bool *ok) {
    int i = 0;
    while (line[i] != '\n' && line[i] != '\r' && line[i] !=
    ↪ '\0') {
        // traktujemy nieznane znaki jako argument.
        // za obsługę błędów będzie odpowiedzialny urzystkownik
    ↪ funkcji
        if (is_unknown_char(scanner, line[i])) {
            if (ok)
                *ok = false;
            return true;
        }

        if (!is_digit(line[i], 16))
            return false;

        i++;
    }

    return true;
}
```

Zauważmy, że funkcja ta przyjmuje jako argument `bool *ok`. Jest to flaga, która zostaje ustawiona na `false` jeżeli w którymś momencie wystąpi błąd. Flaga `ok` będzie pojawiać się jeszcze w wielu miejscach.

Jej wartość jest sprawdzana w każdej iteracji pętli:

```
// main.c

if (!ok) {
    ok = true;
    if (arg_count >= expected_args)
        dump_result(out_file, &result, arg_count,
↪   expected_args,
                    line_idx, op_buffer, is_repl, ok);
    // jeżeli nie zaczyna się jeszcze nowe działanie lub przy
    // parsowaniu wystąpił błąd to nadal pomijamy tą linię
    if (is_argument(&scanner, buffer, &ok) || !ok) {
        ok = false;
        continue;
    }
}
```

Jeżeli napotkaliśmy błąd to pomijamy wszystkie linie aż napotkamy nazwę następnej operacji, wtedy resetujemy stan programu (u nas tylko `ok = true;`) i możemy parsować dalej zapominając o wcześniejszych błędach.

W języku parserów nasza flaga ma reprezentować tak zwany *panic-mode* a powyższy `if` to *synchronizacja* z jedyną różnicą, że kompilator, który raz już wszedł w panic-mode nie będzie próbował dalej kompilować programu (z oczywistych powodów), może jednak nadal parsować kod, aby wykryć dla użytkownika ewentualne pozostałe błędy składniowe. My natomiast staramy się wykonać późniejsze działania, gdyż istnieją one niezależnie od pozostałych.

Jeżeli zdecydujemy że dana linia jest nazwą działania:

```
// main.c

if (is_argument(&scanner, buffer, &ok)) {
    ...
} else {
        if (last_op_idx != 0 && line_idx != 1)
            dump_result(out_file, &result, arg_count,
  ↪  expected_args, last_op_idx, op_buffer, is_repl, ok);

    op = read_instruction(&scanner, &ok);
    if (!ok)
        continue;

    expected_args = op.op_type == Con ? 1 : 2;
    arg_count = 0;
    strcpy(op_buffer, buffer);
    last_op_idx = line_idx;
}
```

Po pierwsze, jeżeli nie jest to pierwsza operacja to printujemy rezultat poprzedniej (funkcja `dump_result`, która wypisuje rezultat sprawdzając czy operacja dostała wystarczającą ilość argumentów). Następnie sczytujemy operację jako `struct oper` zawierający typ opracji i jej bazę. Na koniec updatujemy potrzebne zmienne, m.in. `expected_args` mając na uwadze, że operacja `Con` (Convert) oczekuje tylko 1 argument.

Jeżeli zdecydujemy że linia jest argumentem działania:

```
// main.c

if (is_argument(&scanner, buffer, &ok)) {
    if (!ok)
        continue;

    if (last_op_idx == 0) {
        report(&scanner, error, "Nie podano nazwy operacji\n");
        ok = false;
        continue;
    }
    ...
}
```

Sama funkcja `is_argument` może napotkać błąd, jeżeli tak się stanie - `continue;`. Musimy również sprawdzić, czy nasz argument występuje już po jakiejś operacji (czyli czy `last_op_idx != 0`). Idąc dalej:

```
// main.c

if (is_argument(&scanner, buffer, &ok)) {
    ...

    int base = op.op_type == Con ? (op.base >> 4) + 2 :
    ↪ op.base;

    char arg[ARG_SIZE];
    read_arg(&scanner, arg, base, &ok);
    arg_count++;

    if (!ok)
        continue;
}
```

Funkcja `read_arg` potrzebuje znać bazę argumentu, aby go odczytać, znajduje się ona w `op.base` ale, jako że operacja konwersji systemów operuje na dwóch bazach jednocześnie musiałem zapisać je razem do jednej zmiennej, uprzednio shiftując jedną z nich i odejmując 2 (aby nie były równe 16,

co zajęłoby 5 bitów). Nie jestem w pełni zadowolony z tego rozwiązania, ale wynika ono z tego, że jedynie jedna operacja posiada odmienną strukturę ([liczba liczba] zamiast [znak] [liczba]). Gdyby było więcej działań, które również mają inną strukturę, pewnie zdecydowałbym się na inne rozwiązanie.

Idąc dalej:

```
// main.c

if (is_argument(&scanner, buffer, &ok)) {
    ...

    if (arg_count >= expected_args) {
        result = execute(&scanner, op, result.data, arg, &ok);
        if (!ok)
            continue;

        if (op.op_type == Con) {
            if (!is_repl)
                fprintf(out_file, "%s\n\n", arg);
            dump_result(out_file, &result, arg_count,
↪   expected_args, last_op_idx, op_buffer, is_repl, ok);
            continue;
        }
    } else {
        dynStr_from(&result, arg);
    }
}
```

Jeżeli mamy już wystarczająco argumentów (przynajmniej 2 lub przynajmniej 1 dla konwersji) to wykonujemy działanie - funkcja `execute` zwraca wynik typu `dynStr` (więcej w kolejnej sekcji). W przeciwnym razie zapisujemy argument w zmiennej przechowującej rezultat - `dynStr_from(&result, arg);`.

4.3 Scanner

Zacznijmy od parsowania nagłówka operacji.

```
// scanner.c

oper read_instruction(scanner *scanner, bool *ok) {
    oper op;
    op.op_type = Add;
    op.base = 0;
    char c = advance(scanner);

    if (is_digit(c, 10)) {
        scanner->idx--;

        unsigned short base1 = parse_base(scanner, ok);
        if (!(*ok))
            return op;

        consume_spaces(scanner);

        unsigned short base2 = parse_base(scanner, ok);
        if (!(*ok))
            return op;

        op.base = ((base1 - 2) << 4) | (base2 - 2);
        op.op_type = Con;
        return op;
    }
    ...
}
```

Funkcja `advance` zwraca aktualnie czytany znak i zwiększa wskaźnik scannera. Następnie sprawdzamy czy znak ten jest liczbą (wtedy mamy doczynienia z nagłówkiem operacji zamiany bazy). Parsujemy wtedy pierwszą bazę, konsumujemy dowolną ilość spacji (czyli [liczba] [liczba] również będzie poprawne) i parsujemy drugą bazę. Następnie shiftujemy pierwszą w lewo o 4 i dodajemy obie do `op.base`.

Funkcja `parse_base` opiera się głównie na `parse_int` (opisane później). Sprawdza również czy liczba jest z odpowiedniego przedziału.

Idąc dalej, jeżeli wiemy, że nie mamy doczynienia z zamianą bazy, wystarczy jedynie sczytać bazę:

```
// scanner.c

oper read_instruction(scanner *scanner, bool *ok) {
    ...
    consume_spaces(scanner);

    op.base = parse_base(scanner, ok);
    if (!(*ok))
        return op;

    consume(scanner, '\n');
}
```

I zamienić znak działania na odpowiedni enum.

```
// scanner.c

oper read_instruction(scanner *scanner, bool *ok) {
    ...
    switch (c) {
    case '+':
        op.op_type = Add;
        break;
    case '*':
        op.op_type = Mul;
        break;
    case '^':
        op.op_type = Pow;
        break;
    case '/':
        op.op_type = Div;
        break;
    case '%':
        op.op_type = Mod;
```

```
20          break;
21      default:
22          report(scanner, error, "Nieoczekiwany znak `%c`\n", c);
23          if (ok)
24              *ok = false;
25      }
26
27      return op;
28  }
```

Parsowanie opiera się na funkcji `parse_int`.

```
1  // scanner.c
2
3  void parse_int(scanner *scanner, char *output, unsigned int
   ↪  len, unsigned short base, bool *ok) {
4      char *start = scanner->line + scanner->idx;
5      unsigned int i = 0;
6
7      // ucinamy 0 poprzedzające liczbę (np. 002 -> 2, 00 -> 0)
8      while (start[0] == '0' && start[2] != '\0') {
9          start++;
10     }
11     ...
12 }
```

Najpierw ustawiamy pointer na początek liczby i następnie ucinamy poprzedzające zera. Następnie kopiujemy do outputu kolejne znaki dopóki nie napotkamy na koniec linii.

```
1  // scanner.c
2
3  void parse_int(scanner *scanner, char *output, unsigned int
   ↪  len, unsigned short base, bool *ok) {
4      ...
5      while (start[i] != '\0' && start[i] != '\n' && start[i] !=
   ↪  '\r' && start[i] != ' ') {
6          char c = toupper(start[i++]);
7          advance(scanner);
```

```
        ... // sprawdzanie błędów

        output[i - 1] = c;
    }

    output[i] = '\0';
}
```

Funkcja do czytania argumentu korzysta po prostu z `parse_int`.

```
// scanner.c

void read_arg(scanner *scanner, char *output, unsigned short
  ↪ base, bool *ok) {
    parse_int(scanner, output, ARG_SIZE - 1, base, ok);
}
```

4.4 Dynamiczna alokacja

Wyniki działań mogą mieć (w przeciwieństwie do argumentów) dowolną długość, zatem niezbędne jest alokowanie ich dynamicznie.

```
// helpers.h

typedef struct {
    char *data;
    size_t size;
    size_t capacity;
} dynStr;

void init_dynStr(dynStr *str);
void dynStr_from(dynStr *str, char *s);
void append_char(dynStr *str, char c);
void free_dynStr(dynStr *str);
```

Inicjujemy z `capacity = 8`.

```
// helpers.c

void init_dynStr(dynStr *str) {
    str->size = 0;
    str->capacity = 8;
    str->data = (char *)malloc(sizeof(char) * str->capacity);
    str->data[0] = '\0';
}
```

Przy dodawaniu sprawdzamy czy przekorczyliśmy aktualne `capacity`, jeżeli tak to musimy przealokować pamięć. Wybieramy wtedy dwukrotnie większą pojemność.

```
// helpers.c

void append_char(dynStr *str, char c) {
    if (str->size + 1 >= str->capacity) {
        str->capacity *= 2;
        str->data = (char *)realloc(str->data, str->capacity);
    }

    str->data[str->size++] = c;
    str->data[str->size] = '\0';
}
```

4.5 Wykonywanie operacji

Wykonywaniem operacji zajmuje się `execute.c`. Funkcja `dynStr execute(...)` rozdziela zadanie wykonania operacji do osobnych funkcji ze względu na rodziaj działania.

4.5.1 Dodawanie

Jako algorytm dodawania będziemy wykorzystywać zwykłe dodawanie pisemne

```
// execute.c

```

```
3  dynStr exec_add(short base, char *arg1, char *arg2) {
4      ...
5      for (i = n1 - 1, j = n2 - 1; i >= 0 || j >= 0; i--, j--) {
6          // Możemy dać NULL za ok, bo sprawdziliśmy już błędny
   ↪   input
7          int digit1 = (i >= 0) ? char_to_dec(arg1[i], base,
   ↪   NULL) : 0;
8          int digit2 = (j >= 0) ? char_to_dec(arg2[j], base,
   ↪   NULL) : 0;
9          int sum = digit1 + digit2 + carry;
10
11         char c = int_to_char(sum % base);
12         append_char(&result, c);
13
14         carry = sum / base;
15     }
16
17     if (carry) {
18         char c = int_to_char(carry);
19         append_char(&result, c);
20     }
21
22     reverse(result.data);
23     return result;
24 }
```

Iterujemy liczby od końca (uzupełniając zerami w miarę potrzeby) i dodajemy odpowiednie cyfry z uwzględnieniem przeniesienia.

Zauważmy, że po takiej operacji musimy jeszcze odwrócić rezultat.

```c
// helpers.c

void reverse(char *str) {
    int length = strlen(str);
    for (int i = 0, j = length - 1; i < j; i++, j--) {
        char temp = str[i];
        str[i] = str[j];
        str[j] = temp;
    }
}
```

Być może dało by się uniknąć odwracania ale nie ma to większego znaczenia gdyż złożoność nadal będzie rzędu $O(n)$ gdzie n to długość dłuższej liczby i będzie znacznie szybszy niż mnożenie czy potęgowanie.

W dodawaniu wykorzystujemy również funkcje `char_to_dec` oraz `int_to_char` aby zamienić pojedyncze znaki na dany system liczbowy (poniżej `int_to_char`, `char_to_dec` działa analogicznie).

```c
// helpers.c

char int_to_char(int v) {
    assert(v >= 0 && v <= 15);

    if (v >= 0 && v <= 9) {
        return '0' + v;
    }
    return 'A' + v - 10;
}
```

Obie funkcje wykorzystują dodawanie/odejmowanie znaków aby uzyskać odpowiednio znak reprezentujący liczbę w danym systemie lub z jakiegoś systemu zamienić na liczbę dziesiętną.

4.5.2 Odejmowanie

Działanie odejmowania nie było w liście działań do zaimplementowania ale będzie ono niezbędne do późniejszego dzielenia.

Algorytm jest bardzo podobny do dodawania, z zasadniczą różnicą, że możemy odejmować jedynie liczbę mniejszą od większej:

```
// execute.c

dynStr exec_sub(short base, char *arg1, char *arg2) {
    ...
    if (n1 < n2) {
        append_char(&result, '0');
        return result;
    }
    ...
}
```

4.5.3 Mnożenie

Algorytm mnożenia będzie nieco bardziej skomplikowany ale nadal jest to po prostu dzielenie pisemne (tylko krok dodawania będziemy wykonywać na bierząco zamiast przechowywać wyniki i dodawać pod koniec jak robi się to na kartce).

Zaczynamy od wypełnienia rezultatu $n1 + n2$ zerami (gdzie $n1$ i $n2$ to długości argumentów) gdyż jest to maksymalna możliwa wielkość wyniku (nadmiarowe zera utniemy pod koniec).

```
// execute.c

dynStr exec_mul(short base, char *arg1, char *arg2) {
    int n1 = strlen(arg1), n2 = strlen(arg2);

    dynStr result;
    init_dynStr(&result);
    for (int i = 0; i < n1 + n2; i++) {
        append_char(&result, '0');
    }
    ...
}
```

Następnie dla każdej cyfry pierwszej liczby rozważamy każdą drugiej (wszystko zaczynając od najmniej znaczącej pozycji):

```
// execute.c

dynStr exec_mul(short base, char *arg1, char *arg2) {
    ...
    for (int i = n1 - 1; i >= 0; i--) {
        int carry = 0;
        int digit1 = char_to_dec(arg1[i], base, NULL);

        for (int j = n2 - 1; j >= 0; j--) {
            int digit2 = char_to_dec(arg2[j], base, NULL);
            ...
        }
    ...
}
```

Wykonujemy mnożenie danych cyfr z uwzględnieniem przeniesienia oraz dodajemy do wyniku odpowiednią cyfrę z rezultatu (dlatego musieliśmy uprzednio wypełnić go zerami).

```
// execute.c

dynStr exec_mul(short base, char *arg1, char *arg2) {
    ...
        for (int j = n2 - 1; j >= 0; j--) {
            ...
            int product =
                digit1 * digit2 + carry +
                char_to_dec(result.data[n1 + n2 - (i + j) - 2],
↪   base, NULL);
        }
    ...
}
```

Dalej (podobnie jak w dodawaniu) liczymy przeniesienie, odwracamy rezultat oraz usuwamy nadmiarowe zera.

4.5.4 Dzielenie

Do dzielenia przyda się nam prosta funkcja `compare`, która będzie zwracać wartość > 0 gdy pierwszy argument będzie większy od drugiego.

```
// execute.c

int compare(char *arg1, char *arg2) {
    int len1 = strlen(arg1);
    int len2 = strlen(arg2);

    if (len1 > len2) {
        return 1;
    } else if (len1 < len2) {
        return -1;
    }
    return strcmp(arg1, arg2);
}
```

Będziemy dodawać kolejne cyfry z argumentu do `remaining` i porównywać go z dzielnikiem. Jeżeli będzie większy to wykonujemy sekwencję odejmowań aż będzie mniejszy od dzielnika.

```
// execute.c

dynStr exec_div(scanner *scanner, short base, char *arg1, char
  ↪  *arg2, char *mod, bool *ok) {
    ...
    while (arg1[i] != '\0') {
        append_char(&remaining, arg1[i]);
        trim_leading(&remaining, '0');

        int quotient = 0;
        while (compare(remaining.data, arg2) >= 0) {
            remaining = exec_sub(base, remaining.data, arg2);
            quotient++;
        }
        ...
    }
```

```
    ...
}
```

Po wykonaniu dzielenia w zmiennej `remaining` pozostaje nam reszta z dzielenia, którą wykorzystamy później zatem kopiujemy ją do jednego z argumentów funkcji.

```
// execute.c

dynStr exec_div(scanner *scanner, short base, char *arg1, char
  ↪  *arg2, char *mod, bool *ok) {
    ...
    if (mod) {
        strcpy(mod, remaining.data);
    }
}
```

4.5.5 Dzielenie modulo

Dzielenie modulo wykorzystuje po prostu zaimplementowane wcześniej dzielenie całkowitoliczbowe. Wynik dzielenia modulo będzie zawsze mniejszy lub równy rozmiarowi argumentu, zatem możemy przechować go w tablicy `mod[ARG_SIZE]`.

```
// execute.c

dynStr exec_mod(scanner *scanner, short base, char *arg1, char
  ↪  *arg2, bool *ok) {
    char mod[ARG_SIZE];
    exec_div(scanner, base, arg1, arg2, mod, ok);

    dynStr result;
    dynStr_from(&result, mod);

    return result;
}
```

5 Podsumowanie