Dokumentacja Projektu Kalkulator

Paweł Bielecki

11 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);

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 zczytuje 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 zczytujemy od użytkownika nazwy plików i w pętli czytamy kolejne linie wejścia i wypisujemy rezulataty.

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 musały 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 okacji 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) {
    // traktujemy nieznane znaki jako argument.
    // za obsługę błędów będzie odpowiedzialny użytkownik funkcji

    if (is_unknown_char(scanner, line[0]))
        return true;

    int i = 0;
    while (line[i] != '\n' && line[i] != '\r' && line[i] != '\0') {
        if (!is_digit(line[i], 16))
            return false;
        i++;
    }

    return true;
}
```

5 Podsumowanie