ПРАКТИЧНА РОБОТА №1.
ЗМІШАНЕ ВИКОРИСТАННЯ МОВ СИСТЕМНОГО ПРОГРАМУВАННЯ ДЛЯ ВИРІШЕННЯ СИСТЕМНИХ ЗАДАЧ (2 год)

Мета: ознайомитись із принципами змішаного програмування — поєднання мов низького рівня (асемблер) та високого рівня (C/C++, Pascal, тощо) — для створення ефективних системних програм. Набути практичних навичок у розробці програмних модулів, що викликають асемблерні процедури з коду мов високого рівня.

1. Теоретичні відомості

1.1. Поняття змішаного програмування

Змішане програмування — це метод розробки програм, при якому використовуються декілька мов програмування з різним рівнем абстракції. Найчастіше комбінуються:

- C або C++ — для організації логіки, управління потоками, роботи з пам'яттю;
- Assembler (ASM) — для виконання критично швидких або апаратно-залежних операцій.

1.2. Переваги змішаного програмування

- Підвищення продуктивності програми на критичних ділянках.
- Використання специфічних інструкцій процесора (SIMD, MMX тощо).
- Зменшення обсягу машинного коду.
- Доступ до апаратних ресурсів (порти вводу/виводу, регістри, переривання).
- Гнучкість у налагодженні та оптимізації.

1.3. Способи інтеграції коду

1. Вставки асемблера (inline assembly)
2. Окремі асемблерні модулі (.asm)
3. Виклик з високорівневої мови через extern

2. Практичне завдання для виконання

1. Створити програму на мові C або C++, яка виконує обчислення певної функції (наприклад, y = a * b + c).
2. Винести частину обчислень до асемблерної процедури.
3. Викликати цю процедуру з основної програми.
4. Порівняти результати обчислень і оцінити швидкодію.

3. Приклад виконання

main.c

```
#include <stdio.h>
```

```c
extern int mul_asm(int, int);

int main() {
      int a = 6, b = 7, result;

      result = mul_asm(a, b);

      printf("Результат множення: %d\n", result);
      return 0;
}
```

mul.asm

```
; приклад для MASM(ml64)
PUBLIC mul_asm
mul_asm PROC
mov eax, ecx; перший параметр
imul eax, edx; множення
ret
mul_asm ENDP
END
```

4. Контрольні питання

1. Що таке змішане програмування?
2. Які способи інтеграції асемблерного коду ви знаєте?
3. Які переваги надає використання асемблера у системних програмах?
4. У яких випадках застосування асемблера недоцільне?
5. Які директиви використовуються для оголошення зовнішніх процедур у C і ASM?

ПРАКТИЧНА РОБОТА №2.
ОСНОВИ ПРОГРАМУВАННЯ ЗА ДОПОМОГОЮ API-ФУНКЦІЙ В СЕРЕДОВИЩІ WINDOWS (2 год)

Мета: ознайомитися з принципами використання Windows API для створення програм, що безпосередньо взаємодіють із компонентами операційної системи. Навчитися створювати прості програми, які використовують виклики системних функцій Windows.

1. Теоретичні відомості

Windows API (Application Programming Interface) — це набір функцій, структур та констант, що надаються операційною системою Windows для розробників. За допомогою API програми можуть виконувати такі дії, як робота з файлами, вікнами, повідомленнями, пристроями вводу-виводу та ін.

Основні групи API-функцій Windows:

1. **Kernel API** – функції для роботи з процесами, потоками, пам’яттю, файлами.
2. **User API** – функції для створення графічного інтерфейсу користувача, вікон, обробки подій.
3. **GDI (Graphics Device Interface)** – функції для малювання графіки, роботи зі шрифтами та зображеннями.
4. **Shell API** – функції для взаємодії із середовищем Windows (Провідник, панель завдань тощо).

2. Практичне завдання для виконання

1. Ознайомитися зі структурою типової Windows-програми (WinMain, WndProc).
2. Створити просту програму, яка створює вікно та обробляє повідомлення.
3. Додати до програми виклик декількох API-функцій (наприклад, створення діалогового вікна, виведення повідомлення, робота з файлами).
4. Додати меню або кнопку для виклику певної функції API.
5. Протестувати програму і описати її роботу.

3. Приклад виконання

Приклад програми на C, яка створює просте вікно Windows:

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
      LPSTR lpCmdLine, int nCmdShow) {
      HWND hwnd;
      MSG msg;
      WNDCLASSW wc = { 0 };

      wc.lpszClassName = L"SimpleWindow";
```

```
	wc.hInstance = hInstance;
	wc.hbrBackground = GetSysColorBrush(COLOR_3DFACE);
	wc.lpfnWndProc = WndProc;
	wc.hCursor = LoadCursor(0, IDC_ARROW);

	RegisterClassW(&wc);
	hwnd = CreateWindowW(wc.lpszClassName, L"Hello Windows API",
		WS_OVERLAPPEDWINDOW | WS_VISIBLE,
		100, 100, 350, 250, 0, 0, hInstance, 0);

	while (GetMessage(&msg, NULL, 0, 0)) {
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
	return (int)msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam) {
	switch (msg) {
	case WM_DESTROY:
		PostQuitMessage(0);
		break;
	default:
		return DefWindowProcW(hwnd, msg, wParam, lParam);
	}
	return 0;
}
```

4. Контрольні питання

1. Що таке Windows API і для чого воно використовується?
2. Яка різниця між Kernel API, User API та GDI?
3. Які основні функції використовуються для створення вікон у Windows?
4. Які параметри передаються до функції WinMain?
5. Як відбувається обробка повідомлень у Windows-програмах?

ПРАКТИЧНА РОБОТА №3.
РОЗРОБКА ДИНАМІЧНИХ БІБЛІОТЕК. СТАТИЧНЕ І ДИНАМІЧНЕ КОМПОНУВАННЯ (2 год)

Мета: ознайомитися з принципами створення, використання та підключення бібліотек у середовищі програмування мовою C/C++.

Навчитися створювати статичні (.lib) та динамічні (.dll) бібліотеки, підключати їх до програм і аналізувати відмінності між статичним та динамічним компонуванням.

1. Теоретичні відомості

1.1. Поняття бібліотеки

Бібліотека — це набір функцій і процедур, які можуть повторно використовуватися у різних програмах.

Існує два основних типи бібліотек:

- Статичні бібліотеки (.lib або .a) — підключаються до виконуваного файлу під час компонування, і всі необхідні функції копіюються у програму.
- Динамічні бібліотеки (.dll або .so) — підключаються під час виконання програми, що зменшує розмір виконуваного файлу та дозволяє спільне використання коду.

1.2. Статичне компонування

Під час статичного компонування компоновщик (linker) додає код з бібліотеки безпосередньо до виконуваного файлу (.exe).

Переваги:

- Не потребує зовнішніх файлів під час запуску.
- Швидше завантаження.

Недоліки:

- Збільшення розміру програми.
- Необхідність перекомпіляції при зміні бібліотеки.

1.3. Динамічне компонування

Динамічні бібліотеки (.dll у Windows, .so у Linux) завантажуються у пам’ять лише під час виконання програми.

Вони можуть бути використані одночасно кількома процесами.

Переваги:

- Менший розмір програми.
- Можна оновлювати бібліотеку без перекомпіляції основної програми.

Недоліки:

- Необхідність наявності .dll файлу під час виконання.
- Потенційні конфлікти версій (“DLL Hell”).

2. Практичне завдання для виконання

1. Створити статичну бібліотеку (.lib) з кількома функціями (наприклад, обчислення факторіалу, середнього арифметичного тощо).
2. Створити програму, яка використовує цю бібліотеку та компонується статично.
3. Створити динамічну бібліотеку (.dll) з аналогічними функціями.
4. Створити програму, яка динамічно підключає бібліотеку під час виконання (через LoadLibrary() і GetProcAddress() у Windows).
5. Порівняти розмір виконуваних файлів і поведінку програм при зміні бібліотеки.

3. Приклад коду

3.1. Створення динамічної бібліотеки (DLL)

```
// mymath.c
__declspec(dllexport) int add(int a, int b) {
	return a + b;
}
```

3.2. Заголовний файл

```
// mymath.h
__declspec(dllimport) int add(int a, int b);
```

3.3 Використання DLL у програмі

```
#include <windows.h>
#include <stdio.h>

typedef int (*ADDPROC)(int, int);

int main() {
	HINSTANCE hLib = LoadLibrary("mymath.dll");
	if (!hLib) {
		printf("Cannot load library.\n");
		return 1;
	}

	ADDPROC add = (ADDPROC)GetProcAddress(hLib, "add");
	if (!add) {
		printf("Cannot find function.\n");
		return 1;
	}

	printf("Result: %d\n", add(3, 5));
	FreeLibrary(hLib);
	return 0;
}
```

4. Контрольні питання

1. У чому полягає різниця між статичною та динамічною бібліотеками?
2. Які переваги має динамічне компонування?
3. Що таке функції LoadLibrary() і GetProcAddress()?
4. Яким чином оновлення динамічної бібліотеки впливає на основну програму?
5. Як відбувається процес лінкування при створенні .dll?

ПРАКТИЧНА РОБОТА №4.
ПРИНЦИПИ ФУНКЦІОНУВАННЯ ТА ВАРІАНТИ ПОБУДОВИ ТРАНСЛЯТОРІВ І КОМПІЛЯТОРІВ (2 год)

Мета: Ознайомитися з основними принципами функціонування трансляторів і компіляторів, вивчити основні етапи процесу трансляції програм, зрозуміти різницю між інтерпретацією та компіляцією, а також розглянути основні варіанти архітектури побудови компіляторів.

1. Теоретичні відомості

1.1. Транслятор і компілятор

Транслятор — це програма, яка перетворює програму, записану однією мовою програмування, у програму іншою мовою (зазвичай машинною або проміжною).

Компілятор — це транслятор, який перетворює вихідний текст програми у виконуваний машинний код.

Інтерпретатор — програма, що виконує вихідний код без попередньої компіляції, аналізуючи його під час виконання.

1.2. Основні етапи роботи компілятора

Компілятор зазвичай складається з кількох послідовних фаз:

- Лексичний аналіз — розпізнавання послідовностей символів (лексем): ідентифікаторів, чисел, ключових слів тощо. Результатом є потік токенів.
- Синтаксичний аналіз — побудова синтаксичного дерева на основі граматики мови.
- Семантичний аналіз — перевірка типів, контекстних умов і логічної коректності програми.
- Проміжне представлення — створення внутрішньої форми програми, незалежної від конкретної архітектури (наприклад, тріади, чотвірки, тривіальні дерева).
- Оптимізація коду — покращення ефективності без зміни функціональності (усунення надлишкових обчислень, спрощення виразів тощо).
- Генерація коду — створення машинного коду або байт-коду.
- Компонування (linking) — об’єднання з бібліотеками та створення виконуваного файлу.

1.3. Типи трансляторів

- Однопрохідні компілятори — виконують усі фази в одному проході, використовуються для простих мов.
- Багатопрохідні компілятори — будують проміжні подання і поступово уточнюють результати.
- Інтерпретатори — не створюють виконуваний файл, а безпосередньо виконують команди.

- JIT-компілятори (Just-In-Time) — поєднують інтерпретацію і компіляцію під час виконання програми (наприклад, у Java або .NET).

2. Практичне завдання для виконання

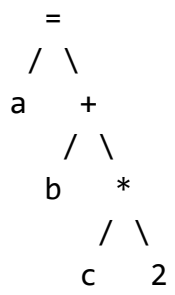
1. Ознайомитися з типовою архітектурою компілятора.
2. Скласти схему етапів роботи транслятора.
3. Реалізувати простий лексичний аналізатор для вхідного тексту (на будь-якій мові програмування).
4. Продемонструвати, як утворюється синтаксичне дерево для простого виразу.
5. Пояснити відмінність між інтерпретацією та компіляцією на прикладі (наприклад, Python і C++).

3. Приклад

Для прикладу розглянемо простий вираз: **`a = b + c * 2`**

Лексичний аналіз: `[id:a] [=] [id:b] [+] [id:c] [*] [num:2]`

Синтаксичне дерево:



Генерація коду (умовно):

```
LOAD b
MUL c, 2
ADD b, c
STORE a
```

4. Контрольні питання

1. Які основні етапи проходить програма під час трансляції?
2. Чим відрізняється транслятор від компілятора?
3. Яке призначення проміжного представлення коду?
4. У чому різниця між front-end і back-end частинами компілятора?
5. У чому полягає різниця між компіляцією і інтерпретацією?
6. Які переваги має багатопрохідна компіляція?

ПРАКТИЧНА РОБОТА №5.
МЕТОДИ СТВОРЕННЯ ЛЕКСИЧНИХ АНАЛІЗАТОРІВ (2 год)

Мета: ознайомитися з принципами побудови та функціонування лексичних аналізаторів (сканерів), навчитися застосовувати методи розпізнавання лексем, освоїти базові підходи до реалізації лексичного аналізу — як ручного, так і автоматизованого, з використанням інструментів типу Lex, Flex або мови програмування C/C++.

1. Теоретичні відомості

1.1. Лексичний аналізатор

Лексичний аналізатор (scanner, lexer) — це програма або частина компілятора, яка виконує попередню обробку вихідного тексту програми, розділяючи його на лексеми — найменші одиниці мови (ключові слова, ідентифікатори, числа, оператори тощо).

Результатом роботи лексичного аналізатора є послідовність токенів, яку надалі обробляє синтаксичний аналізатор.

1.2. Основні завдання лексичного аналізу

- Ігнорування пробілів, коментарів і непотрібних символів.
- Розпізнавання ключових слів, ідентифікаторів, чисел, операторів і роздільників.
- Перевірка правильності структури лексем (наприклад, формату числа або імені змінної).
- Формування лексем для наступних етапів компіляції.

1.3. Основні методи побудови лексичних аналізаторів

1. Ручне програмування (ad hoc)

Лексичний аналізатор пишеться безпосередньо мовою програмування (наприклад, на C або Python).

Для розпізнавання лексем використовуються умовні оператори, цикли або регулярні вирази.

Простий у реалізації, але важко масштабований.

2. Автоматична генерація (за допомогою Lex/Flex)

Використовується опис лексичної граматики у вигляді регулярних виразів.

Інструменти Lex (Unix) або Flex (GNU) автоматично створюють код лексичного аналізатора.

Зручно для мов із великою кількістю правил.

3. Використання регулярних виразів

Регулярні вирази дозволяють описувати шаблони лексем компактно і формально.

Наприклад:

```
[a-zA-Z_][a-zA-Z0-9_]*   – ідентифікатор
[0-9]+                   – ціле число
"if"|"while"|"for"       – ключові слова
```

1.4. Скінченні автомати в лексичному аналізі

Більшість лексичних аналізаторів базуються на детермінованих скінченних автоматах (DFA). DFA послідовно читає символи вхідного потоку і переходить між станами, визначаючи, до якої лексеми належить поточна послідовність символів.

Схематично процес можна подати як:

```
Початковий стан → (читання символів) → Проміжні стани →
(завершення лексеми) → Вихідний токен
```

2. Практичне завдання для виконання

1. Вивчити принципи роботи лексичних аналізаторів.
2. Розробити таблицю лексем і правил для розпізнавання.
3. Реалізувати програму на мові C, яка: читає вхідний рядок (вихідний код); розпізнає числа, ідентифікатори, оператори +, -, *, /, =; виводить результати розпізнавання.
4. Перевірити правильність роботи програми на різних вхідних даних.

4. Приклад коду

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void identify_token(char* token) {
    if (strcmp(token, "+") == 0 || strcmp(token, "-") == 0 ||
        strcmp(token, "*") == 0 || strcmp(token, "/") == 0)
        printf("Оператор: %s\n", token);
    else if (strcmp(token, "=") == 0)
        printf("Оператор присвоєння: %s\n", token);
    else if (isalpha(token[0]))
        printf("Ідентифікатор: %s\n", token);
    else if (isdigit(token[0]))
        printf("Число: %s\n", token);
    else
        printf("Невідома лексема: %s\n", token);
}

int main() {
    char code[256];
    printf("Введіть рядок коду: ");
    fgets(code, sizeof(code), stdin);

    char token[50];
    int i = 0, j = 0;

    while (code[i] != '\0') {
        if (isspace(code[i])) {
```

```
            if (j > 0) {
                token[j] = '\0';
                identify_token(token);
                j = 0;
            }
        }
        else if (strchr("+-*/=()", code[i])) {
            if (j > 0) {
                token[j] = '\0';
                identify_token(token);
                j = 0;
            }
            token[0] = code[i];
            token[1] = '\0';
            identify_token(token);
        }
        else {
            token[j++] = code[i];
        }
        i++;
    }

    if (j > 0) {
        token[j] = '\0';
        identify_token(token);
    }

    return 0;
}
```

Приклад виконання:

Вхідні дані: `x = a + b * 25`

Результат роботи:

```
Ідентифікатор: x
Оператор присвоєння: =
Ідентифікатор: a
Оператор: +
Ідентифікатор: b
Оператор: *
Число: 25
```

5. Контрольні питання

1. Яке призначення лексичного аналізатора?
2. Що таке лексема та токен?
3. Які методи побудови лексичних аналізаторів ви знаєте?
4. Як використовуються регулярні вирази у Lex/Flex?
5. Що таке скінченний автомат і як він застосовується у лексичному аналізі?
6. Які типи лексем зазвичай розпізнаються у мовах програмування?

ПРАКТИЧНА РОБОТА №6.
МЕТОДИ СТВОРЕННЯ СИНТАКСИЧНИХ АНАЛІЗАТОРІВ (3 год)

Мета: ознайомитися з основними принципами побудови синтаксичних аналізаторів (парсерів); навчитися реалізовувати базові алгоритми синтаксичного розбору; засвоїти методи рекурсивного спуску та аналізу за допомогою таблиць переходів.

1. Теоретичні відомості

1.1. Синтаксичний аналіз

Синтаксичний аналізатор (parser) — це програма, що перевіряє, чи відповідає послідовність токенів граматиці мови програмування. Результатом його роботи є синтаксичне дерево (дерево розбору).

Синтаксичний аналізатор працює після лексичного аналізатора і використовує отримані токени для побудови структур вищого рівня — виразів, операторів, блоків тощо.

1.2. Типи синтаксичних аналізаторів

Рекурсивний спуск (Recursive Descent Parser) — простий у реалізації, використовує набір взаємно викликаючих функцій, підходить для LL(1)-граматик.

Висхідні аналізатори (Bottom-Up, LR) — складніші, але універсальніші; можуть обробляти більші класи граматик, реалізуються за допомогою автоматів і таблиць переходів.

Автоматична генерація (YACC/Bison) — граматика задається формальною мовою, інструмент генерує C-код парсера автоматично.

1.3. Приклад граматики

Розглянемо просту граматику арифметичних виразів:

```
E → T { (‘+’ | ‘–’) T }
T → F { (‘*’ | ‘/’) F }
F → '(' E ')' | id | num
```

2. Практичне завдання для виконання

1. Вивчити принцип роботи рекурсивного спуску.
2. Реалізувати синтаксичний аналізатор для виразів, що містять +, -, *, /, дужки, числа та ідентифікатори.
3. Перевірити правильність розбору кількох прикладів виразів.
4. За бажанням — побудувати текстове дерево розбору.

3. Приклад реалізації синтаксичного аналізатора на C

Нижче наведено приклад рекурсивного спуску для розбору арифметичних виразів виду

```
E = T (+|-) T, T = F (*|/) F, F = (E) | num.
```

4.1. Код програми

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

char* input;  // Вхідний рядок
char lookahead; // Поточний символ

void next() {
    lookahead = *input++;
}

void error() {
    printf("Синтаксична помилка!\n");
    exit(1);
}

void match(char t) {
    if (lookahead == t)
        next();
    else
        error();
}

// Прототипи
void E(); void T(); void F();

void E() {
    T();
    while (lookahead == '+' || lookahead == '-') {
        char op = lookahead;
        next();
        T();
        printf("Оператор: %c\n", op);
    }
}

void T() {
    F();
    while (lookahead == '*' || lookahead == '/') {
        char op = lookahead;
        next();
        F();
        printf("Оператор: %c\n", op);
    }
}

void F() {
    if (isdigit(lookahead)) {
        printf("Число: %c\n", lookahead);
        next();
    }
    else if (lookahead == '(') {
        match('(');
        E();
        match(')');
    }
    else {
        error();
    }
```

```
}

int main() {
    char expr[100];
    printf("Введіть арифметичний вираз: ");
    fgets(expr, sizeof(expr), stdin);
    input = expr;
    next();

    E();

    if (lookahead == '\n' || lookahead == '\0')
        printf("Вираз синтаксично правильний.\n");
    else
        error();

    return 0;
}
```

4.2. Приклад виконання

Вхід: `3+(4*5)-2`

Результат:

```
Число: 3
Число: 4
Число: 5
Оператор: *
Оператор: +
Число: 2
Оператор: -
Вираз синтаксично правильний.
```

5. Контрольні питання

1. Що таке синтаксичний аналіз?
2. Які основні типи синтаксичних аналізаторів ви знаєте?
3. У чому полягає різниця між LL і LR аналізаторами?
4. Як реалізується метод рекурсивного спуску?
5. Які переваги має автоматичне створення аналізаторів (YACC/Bison)?
6. Що таке синтаксичне дерево?

ПРАКТИЧНА РОБОТА №7.
МЕТОДИ СТВОРЕННЯ ГЕНЕРАТОРІВ КОДУ (2 год)

Мета: ознайомитися з принципами побудови генераторів коду — складових частин компілятора, що перетворюють проміжне представлення програми у машинний або асемблерний код. Навчитися створювати прості програми, які формують низькорівневі інструкції на основі синтаксичних дерев або тріад.

1. Теоретичні відомості

1.1. Генерація коду

Генератор коду (Code Generator) — це етап компіляції, на якому проміжне представлення програми (дерево розбору, тріади, постфіксний запис тощо) перетворюється на низькорівневі інструкції — асемблерні або машинні.

Основна мета — створити коректний і ефективний код, який можна виконати на цільовій архітектурі.

1.2. Основні етапи генерації коду

1. Вибір адресних регістрів (визначення, які змінні розміщувати в регістрах, а які — у пам’яті).
2. Генерація інструкцій (створення команд MOV, ADD, MUL, SUB тощо).
3. Оптимізація коду (спрощення виразів, усунення дублювань).
4. Формування вихідного файлу (у форматі .asm, .obj, .exe або байткоду).

1.3. Види проміжного представлення

Тип представлення	Приклад	Переваги
Постфіксна форма	`a b + c *`	Простота реалізації
Тріади	`(1) t1 = a + b` `(2) t2 = t1 * c`	Легкість оптимізації
Дерево виразів	`дерево з вузлами +, *, id`	Наглядність структури

2. Практичне завдання для виконання

1. Вивчити принцип роботи генераторів коду.
2. Розробити програму на C, яка: приймає вираз у постфіксній формі; генерує еквівалентні асемблерні інструкції для процесора x86; виводить результат на екран.
3. Перевірити роботу програми на кількох прикладах.

3. Приклад реалізації генератора коду

Нижче наведено простий приклад генерації асемблерних інструкцій для арифметичних виразів у постфіксному (зворотному польському) записі.

3.1. Код програми

```
#include <stdio.h>
#include <ctype.h>

char stack[100];
int top = -1;
int tempCount = 1;

void push(char c) {
	stack[++top] = c;
}

char pop() {
	return stack[top--];
}

void generate(char op) {
	char op2 = pop();
	char op1 = pop();
	printf("MOV R1, %c\n", op1);
	switch (op) {
	case '+': printf("ADD R1, %c\n", op2); break;
	case '-': printf("SUB R1, %c\n", op2); break;
	case '*': printf("IMUL R1, %c\n", op2); break;
	case '/': printf("IDIV R1, %c\n", op2); break;
	}
	printf("MOV T%d, R1\n", tempCount);
	push('A' + tempCount - 1); // імітація тимчасового результату
	tempCount++;
}

int main() {
	char expr[100];
	printf("Введіть вираз у постфіксній формі (наприклад: ab+c*):
");
	scanf("%s", expr);

	for (int i = 0; expr[i] != '\0'; i++) {
		char c = expr[i];
		if (isalnum(c)) {
			push(c);
		}
		else if (c == '+' || c == '-' || c == '*' || c == '/') {
			generate(c);
		}
	}

	printf("\nГенерація завершена. Отримано %d тимчасових
змінних.\n", tempCount - 1);
	return 0;
}
```

3.2. Приклад виконання

Вхід: ab+c*

Результат:

```
MOV R1, a
ADD R1, b
MOV T1, R1
MOV R1, T1
IMUL R1, c
MOV T2, R1

Генерація завершена. Отримано 2 тимчасові змінні.
```

3.3. Пояснення роботи програми

1. Вираз у постфіксній формі аналізується посимвольно.
2. Коли програма зустрічає операнд (a, b, c) — вона додає його у стек.
3. Коли зустрічає оператор (+, *, /), із стека вилучаються два операнди, і генерується відповідна послідовність асемблерних інструкцій.
4. Результат кожної операції зберігається у тимчасовій змінній (T1, T2, ...).

4. Контрольні питання

1. Що таке генератор коду і яке його місце у структурі компілятора?
2. Які типи проміжних представлень використовуються у процесі компіляції?
3. Як формується асемблерний код для арифметичних виразів?
4. У чому полягає різниця між оптимізованою та неоптимізованою генерацією коду?
5. Які переваги має використання постфіксної форми для генерації коду?
6. Як можна розширити програму для підтримки більш складних виразів?