CD LAB Manual - Cd notes

design thinking (JNTUA COLLEGE OF ENGINEERING ANANTAPURAM)

CHAITANYA ENGINEERING COLLEGE KOMMADI,
VISAKHAPATNAM - 530048

CERTIFICATE

This is to certify that _____

_____ is a student studying _____

Register no. _____ branch. _____

has done _____ no of experiments during the year _____

in the subject _____

Lecturer In-charge **Head of the Department**

External Examiner

1. Check the ouput of different compilers gcc, g++, clang, clang++, javac, python etc by running respective language programs with different flags. (purpose to understand preprocessor, optimizations, linker)

C / C++: gcc, g++, clang, clang++

Sample C Program (hello.c)

```
#include <stdio.h>
int main()
{
   printf("Hello, World!\n");
   return 0;
}
```

Output:
Hello, World!

Sample C++ Program (hello.cpp)

```
#include <iostream>
int main()
{
   std::cout << "Hello, C++ World!" << std::endl;
   return 0;
}
```

Output:
Hello, C++ World!

1. Preprocessing Phase

This only expands macros, includes, etc.
Run:
gcc -E hello.c -o hello.i
g++ -E hello.cpp -o hello.i
clang -E hello.c -o hello.i

Output:
File hello.i contains the **expanded code** (e.g., full content of <stdio.h> included).
No compiling or linking occurs.

2. Compilation Phase

Run:

Convert preprocessed code into assembly.
gcc -S hello.c -o hello.s
g++ -S hello.cpp -o hello.s

Output:

- File hello.s is **assembly code**.
- Observe how main and printf translate to instructions.

3. Compilation + Optimization

Try optimization flags to see changes in output.
gcc -O0 -S hello.c -o hello_O0.s # No optimization
gcc -O2 -S hello.c -o hello_O2.s # Aggressive optimization

Compare:
Use diff hello_O0.s hello_O2.s
With -O2, you may see **fewer instructions** or inlined code.

4. Linking Phase

Combine object files into an executable.
gcc -c hello.c -o hello.o # Only compile to object
gcc hello.o -o hello # Link manually

Output:

- Executable hello is created.

Java: javac, java

Sample Java Program (Hello.java)

```
public class Hello
 {
   public static void main(String[] args)
   {
      System.out.println("Hello, Java World!");
   }
}
```

Output:
Hello, Java World!

Run:

javac Hello.java
java Hello

Observations:

- javac compiles to Hello.class (bytecode).
- No preprocessing or linking.
- JVM performs runtime optimizations (JIT).

Python: python / python3

Sample Python Program (hello.py)

```
print("Hello, Python World!")
```

Output:

```
Hello, Python World!
```

Run:

```
python hello.py
```

Observations:

- Interpreted line-by-line.
- You can precompile using:

```
python -m py_compile hello.py
```

- Produces .pyc file in __pycache__/ (bytecode).

2. The Language called TinyCStr is described as follows

a) Every TinyCStr program has one or more functions and syntax of function declaration and function definition is similar to C, one function function must be main.

```
#include <stdio.h>
// Function declaration
void greet();

int main() {
    // Main function definition
    printf("Welcome to TinyCStr!\n");

    // Calling another function
    greet();

    return 0;
}

// Function definition
void greet() {
    printf("Hello from the greet function!\n");
}
```

Output

Welcome to TinyCStr!

Hello from the greet function!

b) Every TinyCStr function has zero or statements

```
#include <stdio.h>

// Function to print a message (one statement)
void printMessage() {
   printf("Welcome to TinyCStr!\n");
}

// Function to add two numbers (one statement)
int add(int a, int b) {
   return a + b;
}

// Function to check if a number is even (one statement)
int isEven(int num) {
   return num % 2 == 0;
}

// Main function (zero or one statement per function)
int main() {
   printMessage(); // Call to printMessage (one statement)
   int sum = add(5, 10); // Call to add (one statement)
   printf("Sum: %d\n", sum); // Print the result (one statement)
   printf("Is 10 even? %s\n", isEven(10) ? "Yes" : "No"); // Check even (one statement)
   return 0; // Return statement (one statement)
}
```

Output

```
Welcome to TinyCStr!
Sum: 15
Is 10 even? Yes
```

c) The possible statements are declaration, assignment, conditional statements (if,else, for, while) except switch

```
#include <stdio.h>
int main() {
   // Declaration and assignment
   int a = 10, b = 20, sum = 0;

   // If-else conditional statement
   if (a > b) {
      printf("a is greater than b\n");
   } else {
      printf("b is greater than or equal to a\n");
   }

   // For loop
   for (int i = 1; i <= 5; i++) {
      sum += i; // Assignment inside the loop
   }
   printf("Sum of first 5 natural numbers: %d\n", sum);

   // While loop
   int count = 5;
   while (count > 0) {
      printf("Countdown: %d\n", count);
      count--; // Assignment inside the loop
   }
   printf("Program executed successfully!\n");
   return 0;
}       print(i);   // Print the current value of i
      i = i + 1;  // Increment i by 1
   }
}

// Main Function
int main() {
   // Call the function to print numbers using while loop
   printNumbersUsingWhile();

   return 0;  // Return 0 indicating successful execution
}
```

Output

```
b is greater than or equal to a
Sum of first 5 natural numbers: 15
Countdown: 5
Countdown: 4
Countdown: 3
Countdown: 2
Countdown: 1
Program executed successfully!
```

d) TinyCStr supports primitive data types of C and a string datatype

i) Implement a lexical analyser for TinyCStr usingflex/lex

Lex File: tinycstr.l

```
%{
#include <stdio.h>
%}

%%

"int"          { printf("Keyword: int\n"); }
"float"        { printf("Keyword: float\n"); }
"char"         { printf("Keyword: char\n"); }
"string"       { printf("Keyword: string\n"); }
[0-9]+         { printf("Number: %s\n", yytext); }
[a-zA-Z_][a-zA-Z0-9_]* { printf("Identifier: %s\n", yytext); }
"+"|"-"|"*"|"/"  { printf("Operator: %s\n", yytext); }
"="            { printf("Assignment Operator: %s\n", yytext); }
";"            { printf("Semicolon\n"); }
"("            { printf("Left Parenthesis\n"); }
")"            { printf("Right Parenthesis\n"); }
"{"            { printf("Left Brace\n"); }
"}"            { printf("Right Brace\n"); }
[ \t\n]+       { /* Ignore whitespace */ }
.              { printf("Unknown character: %s\n", yytext); }

%%

int main() {
   printf("Enter TinyCStr code (Ctrl+D to end):\n");
   yylex();
   return 0;
}

int yywrap() {
   return 1;
}
```

Steps to Run the Program

1. **Install Flex**:
    - On Linux: Use sudo apt install flex or equivalent for your package manager.
    - On Windows: Install Flex via MinGW or Cygwin.
2. **Save the Lex File**: Save the above code in a file named tinycstr.l.
3. **Generate the Lexical Analyzer**: Run the following command in your terminal:

   flex tinycstr.l

This generates a lex.yy.c file.

4. **Compile the Generated C File**: Use a C compiler like GCC to compile the generated file:

   gcc lex.yy.c -o tinycstr –lfl

   The -lfl flag links the Flex library.

5. **Run the Lexical Analyzer**: Execute the compiled program:

   ./tinycstr

   Enter your TinyCStr code as input, and the program will analyze it. Use Ctrl+D (Linux/Mac) or Ctrl+Z (Windows) to signal the end of input.

Output:
Input:
int x = 10;
string name = "John";
float y = 3.14;

Output:
Keyword: int
Identifier: x
Assignment Operator: =
Number: 10
Semicolon
Keyword: string
Identifier: name
Assignment Operator: =
Unknown character: "
Identifier: John
Unknown character: "
Semicolon
Keyword: float
Identifier: y
Assignment Operator: =
Number: 3
Unknown character: .
Number: 14
Semicolon

ii) Implement a parser for TinyCStr using bison/yacc and generate AST(Abstract Syntax Tree)

1.Bison/Yacc File (tinycstr.y**)**

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// AST Node Structure
typedef struct ASTNode {
    char *type;
    char *value;
    struct ASTNode *left;
    struct ASTNode *right;
} ASTNode;

// Function to create a new AST node
ASTNode* createNode(char *type, char *value, ASTNode *left, ASTNode *right) {
    ASTNode *node = (ASTNode *)malloc(sizeof(ASTNode));
    node->type = strdup(type);
    node->value = value ? strdup(value) : NULL;
    node->left = left;
    node->right = right;
    return node;
}

// Function to print the AST
void printAST(ASTNode *node, int depth) {
    if (!node) return;
    for (int i = 0; i < depth; i++) printf("  ");
    printf("%s: %s\n", node->type, node->value ? node->value : "NULL");
    printAST(node->left, depth + 1);
    printAST(node->right, depth + 1);
}

void yyerror(const char *s);
int yylex();
ASTNode *root;
%}

%union {
    char *str;
    ASTNode *node;
}

%token <str> IDENTIFIER STRING_LITERAL
%token INT FLOAT CHAR STRING
%type <node> declaration type_specifier initializer

%%
```

```
program:
   declaration { root = $1; }
   ;

declaration:
   type_specifier IDENTIFIER '=' initializer ';' {
      $$ = createNode("Declaration", $2, $1, $4);
   }
   ;

type_specifier:
   INT { $$ = createNode("Type", "int", NULL, NULL); }
   | FLOAT { $$ = createNode("Type", "float", NULL, NULL); }
   | CHAR { $$ = createNode("Type", "char", NULL, NULL); }
   | STRING { $$ = createNode("Type", "string", NULL, NULL); }
   ;

initializer:
   STRING_LITERAL { $$ = createNode("Initializer", $1, NULL, NULL); }
   | IDENTIFIER { $$ = createNode("Initializer", $1, NULL, NULL); }
   ;

%%

void yyerror(const char *s) {
   fprintf(stderr, "Error: %s\n", s);
}
```

2. Lex File (tinycstr.l**)**

```
%{
#include "tinycstr.tab.h"
%}

%%

"int"       { return INT; }
"float"     { return FLOAT; }
"char"      { return CHAR; }
"string"    { return STRING; }
[a-zA-Z_][a-zA-Z0-9_]* { yylval.str = strdup(yytext); return IDENTIFIER; }
\"[^\"]*\"  { yylval.str = strdup(yytext); return STRING_LITERAL; }
"="         { return '='; }
";"         { return ';'; }
[ \t\n]+    { /* Ignore whitespace */ }
.           { return yytext[0]; }

%%

int yywrap() {
```

```
    return 1;
}
```

3. Main File (main.c**)**

```
#include <stdio.h>
#include "tinycstr.tab.h"

extern int yyparse();
extern ASTNode *root;

int main() {
    printf("Enter TinyCStr code:\n");
    if (yyparse() == 0) {
        printf("Parsing successful! Abstract Syntax Tree:\n");
        printAST(root, 0);
    } else {
        printf("Parsing failed.\n");
    }
    return 0;
}
```

4. Compilation and Execution

1. **Install Bison and Flex**: Ensure you have Bison and Flex installed on your system. Use sudo apt install bison flex (Linux) or equivalent for your OS.
2. **Compile the Parser**:

   bison -d tinycstr.y

   flex tinycstr.l

   gcc -o tinycstr tinycstr.tab.c lex.yy.c main.c –lfl
3. **Run the Program**:
   ./tinycstr
4. **Input Example**:
   string name = "Hello, World!";
   ``

iii) Generate a 3-address code from theAST

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

// Define node types for AST
typedef enum { NODE_INT, NODE_FLOAT, NODE_CHAR, NODE_STRING, NODE_OP }
NodeType;

// Define structure for AST nodes
typedef struct ASTNode {
   NodeType type;
   char op; // Operator for NODE_OP
   char *value; // Value for literals (int, float, char, string)
   struct ASTNode *left, *right; // Left and right children for operations
} ASTNode;

// Function to create a new AST node
ASTNode* createNode(NodeType type, char *value, char op, ASTNode *left, ASTNode *right) {
   ASTNode *node = (ASTNode *)malloc(sizeof(ASTNode));
   node->type = type;
   node->value = value ? strdup(value) : NULL;
   node->op = op;
   node->left = left;
   node->right = right;
   return node;
}

// Function to generate 3-address code
int tempVarCount = 0;
void generateTAC(ASTNode *node) {
   if (!node) return;

   if (node->type == NODE_OP) {
      generateTAC(node->left);
      generateTAC(node->right);

      char tempVar[10];
      sprintf(tempVar, "t%d", tempVarCount++);

      printf("%s = ", tempVar);
      if (node->left->type == NODE_OP) printf("t%d", tempVarCount - 2);
      else printf("%s", node->left->value);

      printf(" %c ", node->op);

      if (node->right->type == NODE_OP) printf("t%d\n", tempVarCount - 1);
      else printf("%s\n", node->right->value);
   } else {
      // Leaf nodes (literals) do not generate TAC
```

```
    }
}

// Main function
int main() {
    // Example AST: (5 + 3) * 2
    ASTNode *node1 = createNode(NODE_INT, "5", '\0', NULL, NULL);
    ASTNode *node2 = createNode(NODE_INT, "3", '\0', NULL, NULL);
    ASTNode *node3 = createNode(NODE_OP, NULL, '+', node1, node2);
    ASTNode *node4 = createNode(NODE_INT, "2", '\0', NULL, NULL);
    ASTNode *root = createNode(NODE_OP, NULL, '*', node3, node4);

    printf("3-Address Code:\n");
    generateTAC(root);

    // Free memory (not shown for brevity)
    return 0;
```

Output:

```
3-Address Code:
t0 = 5 + 3
t1 = t0 * 2
```

iv) Generate MIPS instructions from 3-address code and run it on SPIM simulator

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

#define MAX_LINES 100
#define MAX_LEN 100

// Function to generate MIPS instructions
void generateMIPS(char tac[MAX_LINES][MAX_LEN], int n) {
   printf(".data\n");
   printf("str: .asciiz \"Hello, World!\"\n"); // Example string
   printf(".text\n");
   printf(".globl main\n");
   printf("main:\n");

   for (int i = 0; i < n; i++) {
      char op[10], arg1[10], arg2[10], result[10];
      int num = sscanf(tac[i], "%s %s %s %s", result, op, arg1, arg2);

      if (num == 4) {
         if (strcmp(op, "+") == 0) {
            printf("    lw $t0, %s\n", arg1);
            printf("    lw $t1, %s\n", arg2);
            printf("    add $t2, $t0, $t1\n");
            printf("    sw $t2, %s\n", result);
         } else if (strcmp(op, "-") == 0) {
            printf("    lw $t0, %s\n", arg1);
            printf("    lw $t1, %s\n", arg2);
            printf("    sub $t2, $t0, $t1\n");
            printf("    sw $t2, %s\n", result);
         } else if (strcmp(op, "*") == 0) {
            printf("    lw $t0, %s\n", arg1);
            printf("    lw $t1, %s\n", arg2);
            printf("    mul $t2, $t0, $t1\n");
            printf("    sw $t2, %s\n", result);
         } else if (strcmp(op, "/") == 0) {
            printf("    lw $t0, %s\n", arg1);
            printf("    lw $t1, %s\n", arg2);
            printf("    div $t2, $t0, $t1\n");
            printf("    sw $t2, %s\n", result);
         }
      } else if (num == 2 && strcmp(op, "print") == 0) {
         printf("    li $v0, 4\n");
         printf("    la $a0, %s\n", result);
         printf("    syscall\n");
      }
   }

   printf("    li $v0, 10\n"); // Exit syscall
   printf("    syscall\n");
```

```
}

int main() {
    char tac[MAX_LINES][MAX_LEN];
    int n;

    printf("Enter the number of three-address code lines: ");
    scanf("%d", &n);
    getchar(); // Consume newline character

    printf("Enter the three-address code (one per line):\n");
    for (int i = 0; i < n; i++) {
        fgets(tac[i], MAX_LEN, stdin);
        tac[i][strcspn(tac[i], "\n")] = '\0'; // Remove newline
    }

    printf("\nGenerated MIPS Assembly:\n");
    generateMIPS(tac, n);

    return 0;
}
```

Output:

```
Enter the number of three-address code lines: 3
Enter the three-address code (one per line):
t1=a+b
t2=t1*c
print str

Generated MIPS Assembly:
.data
str: .asciiz "Hello, World!"
.text
.globl main
main:
    li $v0, 10
    syscall
```

3 Write a program illustrating code optimization techniques:

i) Constant folding

```
 #include <stdio.h>

int main() {
    // Example of constant folding
    int a = 10;
    int b = 20;
    int result = (a * 2) + (b / 2); // Compiler will optimize this

    printf("Result: %d\n", result);

    // Equivalent to:
    // int result = 20 + 10; // Compiler evaluates this at compile time

    return 0;
}
```

Output:
Result: 30

ii) Copy propagation

```
#include <stdio.h>

int main() {
   int a, b, c;

   // Before copy propagation
   a = 10;  // 'a' is assigned 10
   b = a;   // 'b' is assigned the value of 'a'
   c = b + 5; // 'c' uses 'b', which is redundant

   printf("Value of c: %d\n", c);

   // After copy propagation
   // Directly replace 'b' with 'a' in the computation of 'c'
   c = a + 5;

   printf("Optimized value of c: %d\n", c);

   return 0;
}
```

Output:
Value of c: 15
Optimized value of c: 15

iii) Common sub expression elimination

```
#include <stdio.h>

int main() {
    int a = 5, b = 10, c = 15;
    int result1, result2;

    // Without Common Subexpression Elimination
    result1 = (a * b) + (a * b) + c;

    // With Common Subexpression Elimination
    int temp = a * b; // Common subexpression stored in 'temp'
    result2 = temp + temp + c;

    // Output results
    printf("Result without optimization: %d\n", result1);
    printf("Result with optimization: %d\n", result2);

    return 0;
}
```

Output:

```
Result without optimization: 115
Result with optimization: 115
```

iv)Loop unrolling

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

#define SIZE 1000000

// Function without loop unrolling
void sumWithoutUnrolling(int *arr, int size, long long *result) {
    *result = 0;
    for (int i = 0; i < size; i++) {
        *result += arr[i];
    }
}

// Function with loop unrolling
void sumWithUnrolling(int *arr, int size, long long *result) {
    *result = 0;
    int i;
    for (i = 0; i < size - 3; i += 4) { // Process 4 elements per iteration
        *result += arr[i] + arr[i + 1] + arr[i + 2] + arr[i + 3];
    }
    // Handle remaining elements
    for (; i < size; i++) {
        *result += arr[i];
    }
}

int main() {
    int arr[SIZE];
    for (int i = 0; i < SIZE; i++) {
        arr[i] = i + 1; // Initialize array with values 1 to SIZE
    }

    long long result;
    clock_t start, end;

    // Without loop unrolling
    start = clock();
    sumWithoutUnrolling(arr, SIZE, &result);
    end = clock();
    printf("Sum without unrolling: %lld, Time: %f seconds\n", result, (double)(end - start) /
CLOCKS_PER_SEC);

    // With loop unrolling
    start = clock();
    sumWithUnrolling(arr, SIZE, &result);
    end = clock();
    printf("Sum with unrolling: %lld, Time: %f seconds\n", result, (double)(end - start) /
CLOCKS_PER_SEC);
```

```
    return 0;
}
```

Output:

```
Sum without unrolling: 500000500000, Time: 0.001503 seconds
Sum with unrolling: 500000500000, Time: 0.000730 seconds
```

v) Dead code elimination

```
#include <stdio.h>

int main() {
   int a = 10, b = 20, c = 0;

   // Dead code: This block has no effect on the program's output
   if (a > b) {
      c = a + b;  // This assignment is never used
   }

   // Useful code
   printf("The value of b is: %d\n", b);

   return 0;
}
```

Output:

The value of b is: 20