1 Question-1

Phase 1: Lexical Analysis

The lexical analyzer breaks the program into tokens, grouping lexemes and providing their token types. Below is the tokenization for the provided C program.

Line No.	Tokenized Output
1	<KW, INCLUDE> <SPECIAL SYMBOL, < > <ID, STDIO> <SPECIAL SYMBOL, >
2	<KW, CONST> <KW, INT> <ID, n> <OP, ASSIGN> <ICONST, 10> <PUNC, ;>
4	<KW, INT> <ID, f> <SPECIAL SYMBOL, (> <KW, INT> <ID, n> <SPECIAL SYMBOL,)> <SPECIAL SYME
5	<KW, IF> <SPECIAL SYMBOL, (> <ICONST, 0> <OP, EQ> <ID, n> <OP, OR> <ICONST, 1> <OP, EQ> <
6	<KW, RETURN> <ID, n> <PUNC, ;>
7	<KW, RETURN> <ID, f> <SPECIAL SYMBOL, (> <ID, n> <OP, MINUS> <ICONST, 2> <SPECIAL SYMBOL,
9	<SPECIAL SYMBOL, }>
10	<KW, INT> <ID, r> <OP, ASSIGN> <ID, f> <SPECIAL SYMBOL, (> <ID, n> <SPECIAL SYMBOL,)> <P
12	<KW, PRINTF> <SPECIAL SYMBOL, (> <STRING LIT, f(%d) = %d> <PUNC, ,> <ID, n> <PUNC, ,> <ID
13	<KW, RETURN> <ICONST, 0> <PUNC, ;>

Symbol Table

Lexeme	Description
n	Variable, Integer
f	Function, Returns Integer
r	Variable, Integer

Phase 2: Syntax Analysis

The syntax analyzer constructs a syntax tree to represent the grammatical structure of the token stream. Here's a portion of the tree for the key parts of the program:

```
- Function `f(n)`:
  f(n) -> if (n == 0 || n == 1) -> return n
        -> else -> return f(n - 2) + (n << 1) - 1

- Main Function:
  main() -> int r = f(n)
         -> printf("f(%d) = %d", n, r)
         -> return 0
```

This tree structure is based on the grammar rules of C for conditionals, function calls, and arithmetic expressions.

Phase 3: Semantic Analysis

In the semantic analysis phase, type checks and consistency evaluations are performed. The following checks are made:

- **Type Consistency**: All variables (`n`, `r`) and function `f` are of type `int`. There are no mismatches.
- **Function Call**: The recursive function `f(n)` is valid since the argument is of the correct type and it returns an integer.
- **Bitwise Shift Operation**: The left shift operator (`<<`) is applied correctly on an integer variable.

No semantic errors are present in the program, so the syntax tree remains unchanged.

Phase 4: Intermediate Code Generation

Here is the intermediate code in three-address format:

```
01 t1 = 10            // Load constant n = 10
02 if (t1 == 0) goto L1
03 if (t1 == 1) goto L1
04 t2 = t1 - 2        // Compute n - 2
05 t3 = f(t2)         // Call f(n - 2)
06 t4 = t1 << 1       // Compute n << 1
07 t5 = t4 - 1        // Compute (n << 1) - 1
08 t6 = t3 + t5       // Add f(n - 2) + (n << 1) - 1
09 r = t6             // Store result in r
10 printf("f(%d) = %d", t1, r)
11 return 0
L1: r = t1            // Return n if n == 0 or n == 1
12 printf("f(%d) = %d", t1, r)
13 return 0
```

This represents the exact steps of the program logic in a low-level representation.

Phase 5: Code Optimization

After optimization, redundant code and unnecessary computations are removed. The optimized three-address code is:

```
01 t1 = 10            // Load n = 10
02 if (t1 == 0 || t1 == 1) goto L1
03 t2 = t1 - 2        // n - 2
04 t3 = f(t2)         // Call f(n - 2)
05 t4 = (t1 << 1) - 1 // Compute (n << 1) - 1
06 r = t3 + t4        // r = f(n - 2) + (n << 1) - 1
07 goto L2
L1: r = t1            // Return n if n == 0 or n == 1
L2: printf("f(%d) = %d", t1, r)
08 return 0
```

This optimized version reduces unnecessary conditional jumps and merges operations.

Phase 6: Code Generation

Finally, the intermediate code is translated into machine code. Assuming a basic architecture, the machine instructions would be:

```
LD R1, #10            // Load 10 into register R1
BEQZ R1, L1           // If R1 == 0, branch to L1
SUB R2, R1, #2        // Compute R1 - 2, store in R2
CALL f(R2)            // Call function f with R2
LSH R3, R1, #1        // Shift R1 left by 1, store in R3
SUB R3, R3, #1        // Subtract 1 from R3
ADD R4, R2, R3        // Add R2 and R3, store in R4
ST r, R4              // Store result in r
BR L2                 // Jump to L2
L1: ST r, R1          // Store R1 in r if R1 == 0 or R1 == 1
L2: PRNT "f(%d) = %d", R1, R4
HALT
```

This is the exact machine code corresponding to the program's logic.
[12pt,a4paper]article [utf8]inputenc [T1]fontenc listings color xcolor geometry fancyhdr
a4paper, margin=1in

2 Question-2

Assembly Code Annotation

PUBLIC Section

```
PUBLIC _n
PUBLIC _f
PUBLIC _main
```

- **PUBLIC _n, _f, _main:** Declares the symbols _n, _f, and _main as global, allowing them to be referenced across different modules in the program.

EXTERN Section

```
EXTRN __imp__printf:PROC
EXTRN __RTC_CheckEsp:PROC
EXTRN __RTC_InitBase:PROC
EXTRN __RTC_Shutdown:PROC
```

- **EXTRN __imp__printf:PROC:** Declares the `printf` function, imported from the C runtime library.
- **EXTRN __RTC_CheckEsp, __RTC_InitBase, __RTC_Shutdown:** External functions for runtime stack checking and initialization/shutdown, used to ensure the integrity of stack operations.

CONST Section

```
CONST SEGMENT
_n DD 0aH
??_C@_0L@HHEJLIDJ@f?$CI?$CFd?$CJ?5?$DN?5?$CFd?$AA@ DB 'f(%d) = %d', 00H
CONST ENDS
```

- **_n DD 0aH:** Defines the global variable _n and initializes it with the hexadecimal value `0x0a` (decimal 10).
- **??_C@... DB 'f(%d) = %d', 00H:** Declares the format string used in `printf` for printing the result of function `f`. This is stored as a constant string in memory.

TEXT Section

```
_TEXT SEGMENT
_r$ = -8
_main PROC ; COMDAT
```

- **_TEXT SEGMENT:** Begins the code segment where all assembly instructions are stored.
- **_r$ = -8:** Declares the local variable `r`, which is stored at the offset `-8` from the base pointer.

- **_main PROC:** Declares the start of the `main` function.

```
push ebp
mov ebp, esp
sub esp, 204
push ebx
push esi
push edi
lea edi, DWORD PTR [ebp-204]
mov ecx, 51
mov eax, -858993460
rep stosd
```

- **push ebp, mov ebp, esp:** Standard function prologue. Saves the old base pointer and sets up the new base pointer for the current stack frame.
- **sub esp, 204:** Allocates 204 bytes of local stack space for local variables.
- **push ebx, esi, edi:** Saves registers `ebx`, `esi`, and `edi`, as they may be used by the function.
- **lea edi, [ebp-204], mov ecx, 51, mov eax, -858993460, rep stosd:** Clears the local stack frame (204 bytes) by filling it with the value `0xcccccccc`, which is a common debugging pattern.

```
mov eax, DWORD PTR _n
push eax
call _f
add esp, 4
mov DWORD PTR _r$ [ebp], eax
```

- **mov eax, DWORD PTR _n:** Loads the value of global variable `n` into register `eax`.
- **push eax, call _f:** Pushes `n` onto the stack and calls the function `f`.
- **add esp, 4:** Cleans up the stack after the function call by adjusting the stack pointer.
- **mov DWORD PTR _r$ [ebp], eax:** Stores the return value of `f(n)` into the local variable `r`.

```
mov esi, esp
mov eax, DWORD PTR _r$[ebp]
push eax
mov ecx, DWORD PTR _n
push ecx
push OFFSET ??_C@_0L@...
call DWORD PTR __imp__printf
add esp, 12
cmp esi, esp
call __RTC_CheckEsp
```

- **mov esi, esp:** Stores the current stack pointer in `esi` for later comparison.
- **mov eax, DWORD PTR _r$ [ebp], push eax:** Loads the value of `r` and pushes it onto the stack as an argument to `printf`.
- **mov ecx, DWORD PTR _n, push ecx:** Loads the value of `n` and pushes it onto the stack as the second argument to `printf`.

- **push OFFSET ??_C@...:** Pushes the address of the format string used in `printf`.
- **call __imp__printf:** Calls the external `printf` function to print the result.
- **add esp, 12:** Cleans up the arguments pushed onto the stack for `printf`.
- **cmp esi, esp, call __RTC_CheckEsp:** Compares the current stack pointer to the saved value and calls `CheckEsp` to verify stack consistency.

```
xor eax, eax
pop edi
pop esi
pop ebx
add esp, 204
cmp ebp, esp
call __RTC_CheckEsp
mov esp, ebp
pop ebp
ret 0
```

- **xor eax, eax:** Sets `eax` to 0, representing the return value `0` from `main`.
- **pop edi, pop esi, pop ebx:** Restores the registers `edi`, `esi`, and `ebx`.
- **add esp, 204:** Deallocates the local stack space.
- **cmp ebp, esp, call __RTC_CheckEsp:** Checks for stack consistency again before exiting.
- **mov esp, ebp, pop ebp, ret 0:** Standard function epilogue, restores the original base pointer and returns from `main`.

Function f()

```
_f PROC ; COMDAT
_n$ = 8
push ebp
mov ebp, esp
sub esp, 192
push ebx
push esi
push edi
lea edi, DWORD PTR [ebp-192]
mov ecx, 48
mov eax, -858993460
rep stosd
```

- **_f PROC:** Declares the start of the function `f`.
- **_n$ = 8:** The local variable `n` is stored at an offset of `8` from the base pointer.
- **push ebp, mov ebp, esp:** Standard function prologue, setting up the stack frame.
- **sub esp, 192:** Allocates 192 bytes for local variables.
- **push ebx, esi, edi:** Saves the registers `ebx`, `esi`, and `edi`.
- **lea edi, [ebp-192], mov ecx, 48, mov eax, -858993460, rep stosd:** Clears the local stack frame with the pattern `0xcccccccc`.

```
cmp DWORD PTR _n$ [ebp], 0
je SHORT $LN1@f
cmp DWORD PTR _n$ [ebp], 1
jne SHORT $LN2@f
$LN1@f:
mov eax, DWORD PTR _n$ [ebp]
jmp SHORT $LN3@f
$LN2@f:
mov eax, DWORD PTR _n$ [ebp]
sub eax, 2
push eax
call _f
add esp, 4
mov ecx, DWORD PTR _n$ [ebp]
lea eax, DWORD PTR [eax+ecx*2-1]
$LN3@f:
pop edi
pop esi
pop ebx
add esp, 192
cmp ebp, esp
call __RTC_CheckEsp
mov esp, ebp
pop ebp
ret 0
```

- **cmp DWORD PTR _n$ [ebp], 0, je SHORT $LN1@f:** Compares the local variable `n` to 0. If equal, jumps to the label `$LN1@f`.
- **cmp DWORD PTR _n$ [ebp], 1, jne SHORT $LN2@f:** If `n` is not equal to 1, jumps to the label `$LN2@f`.
- **$LN1@f: mov eax, DWORD PTR _n$ [ebp], jmp SHORT $LN3@f:** If `n` is 0 or 1, sets `eax` to `n` and jumps to `$LN3@f`.
- **$LN2@f: mov eax, DWORD PTR _n$ [ebp], sub eax, 2, push eax, call _f, add esp, 4:** If `n` is not 0 or 1, recursively calls `f(n-2)`.
- **mov ecx, DWORD PTR _n$ [ebp], lea eax, DWORD PTR [eax+ecx*2-1]:** Computes the final return value as `f(n-2) + (n < 1) - 1`.
- **$LN3@f: pop edi, pop esi, pop ebx, add esp, 192:** Restores the saved registers and deallocates local stack space.
- **cmp ebp, esp, call __RTC_CheckEsp:** Checks stack consistency.
- **mov esp, ebp, pop ebp, ret 0:** Function epilogue, restoring the base pointer and returning from `f`.

3 Question-3

3(a) Identification of `printf` in the Assembly Code

In the assembly code, `printf` is identified in several places through its external declaration, string literals, and the actual function call. Each of these lines plays a role in identifying and setting up the `printf` function.

- **Line 68:**

```
68 EXTRN __imp__printf:PROC
```

Purpose: This declares `printf` as an external function. The `EXTRN` directive tells the assembler that the `printf` function is defined elsewhere (in a standard library, such as the C runtime library), and the symbol `__imp__printf` is an imported procedure (`PROC`).

- **Line 85:**

```
85 ??_C@_0L@HHEJLIDJ@f?$CI?$CFd?$CJ?5?$DN?5?$CFd?$AA@ DB 'f(%d) = %d', 00H ; `string'
```

Purpose: This line defines the format string for `printf`. The string `'f(%d) = %d'` is the format string that will be used by `printf` to print the values of `n` and the result of the function `f(n)`. The `DB` (Define Byte) directive creates the string in memory with a null byte (`00H`) at the end, which terminates the string.

- **Lines 112–113:**

```
112 push eax      ; push first argument (n) onto stack
113 call _f       ; call the function f(n)
```

Purpose: While not directly related to `printf`, this pushes the argument to the function call to `f`. The result of `f(n)` will later be used in the call to `printf`.

- **Lines 68 and 71:**

```
71 push DWORD PTR __imp__printf ; pass printf function reference
```

Purpose: This line pushes the address of the `printf` function onto the stack. The program relies on dynamic linking to find the function at runtime.

3(b) Body of `printf` Located and Linked with the Call

- The **dynamic linking mechanism** is used to resolve the address of `printf`. In the assembly code, `printf` is declared as an external function with the symbol `__imp__printf` (line 68). At runtime, the dynamic linker (part of the operating system) searches for this symbol in the C runtime library (CRT), and once found, it sets the actual address of `printf` in memory.
- When `printf` is called in the assembly code, the program doesn't have a direct address for the function in the code. Instead, it pushes the address of `__imp__printf` onto the stack (line 71), which the dynamic linker resolves to the actual function body when the program is loaded and executed.
- This approach ensures that the `printf` function, which is part of a shared library, can be dynamically linked at runtime.

3(c) Passing Parameters to `printf`

In the x86 calling convention used here, parameters are passed to functions via the stack, in **right-to-left order**. For `printf`, the format string and the values to be printed (e.g., `n` and the result of `f(n)`) are pushed onto the stack before calling the function.

Here's the sequence of steps for passing the arguments:

- **Step 1: Push format string** In line 85, the format string `'f(%d) = %d'` is defined and stored in memory. Before calling `printf`, the address of this string is pushed onto the stack.

```
114 push OFFSET ??_C@_0L@HHEJLIDJ@f?$CI?$CFd?$CJ?5?$DN?5?$CFd?$AA@
```

- **Step 2: Push the first argument (value of `n`)** In line 112, the value of `n` (stored in the register `eax`) is pushed onto the stack as the first argument to `printf`.

```
112 push eax  ; push the value of n onto the stack
```

- **Step 3: Push the second argument (result of `f(n)`)** After calling `f(n)` and storing its result in `eax`, the value is pushed onto the stack as the second argument for `printf`.

```
113 push eax  ; push the result of f(n) onto the stack
```

- **Step 4: Call `printf`** Once all the arguments have been pushed onto the stack in reverse order, `printf` is called using the `call __imp__printf` instruction:

```
116 call DWORD PTR __imp__printf
```

  This calls the `printf` function, which uses the arguments passed on the stack to format and print the output.

- **Step 5: Clean up the stack** After the call to `printf`, the stack is cleaned up by adjusting the stack pointer:

```
117 add esp, 12
```

  Since three arguments (the format string and two integers) were pushed onto the stack (each occupying 4 bytes), `esp` is incremented by 12 to remove them from the stack and restore the stack pointer.

Summary

- **Identification**: `printf` is identified through the `EXTRN` declaration, the format string, and the function call.
- **Linking**: The actual body of `printf` is located at runtime via dynamic linking. The `EXTRN __imp__printf` declaration allows the dynamic linker to resolve the function's address.
- **Parameter Passing**: Arguments to `printf` (the format string and the values) are passed via the stack, in right-to-left order, with the format string pushed last.

4 Question-4

4(a) Roles of the Run-Time Checkers (RTC)

Run-Time Checkers (RTC) in Microsoft Visual C++ are debugging tools that help catch runtime errors related to memory and stack management, which are otherwise difficult to detect. These checks are particularly useful for identifying stack corruption, uninitialized variables, and buffer overruns during program execution. RTC assists in detecting issues that might not be visible during compile time but could lead to undefined behavior or crashes at runtime.

The primary role of RTC includes:

1. **Stack Integrity**: It monitors the stack for any corruption caused by mismatched pushes and pops or buffer overflows that might alter critical return addresses or local variables.

2. **Memory Access**: RTC detects illegal memory accesses, like reading uninitialized variables or writing beyond the allocated memory buffer.

3. **Base Initialization and Shutdown**: RTC ensures that necessary initialization for checking the stack and memory occurs before execution and cleans up after the program ends.

By inserting additional checks into the program, RTC aims to identify runtime errors that could lead to subtle bugs, crashes, or security vulnerabilities.

4(b) Explanation of RTC Functions

(i) `__RTC_CheckEsp:PROC`

Purpose: This procedure ensures that the **stack pointer (ESP)** remains consistent after a function call. It verifies that the amount of data pushed onto the stack has been correctly popped off, i.e., the stack is properly balanced before and after function execution.

Why it's important:

- If the ESP register is misaligned, it typically means that the function has altered the stack in an unintended way (e.g., by pushing more values than it pops off). This can lead to stack corruption, crashes, or incorrect function return values. `__RTC_CheckEsp` helps in detecting such stack corruption issues.

Related Assembly Line:

```
89 call __RTC_CheckEsp
```

Explanation: This instruction calls `__RTC_CheckEsp` at the end of the `main` function to verify that the stack pointer has the correct value and that no stack corruption occurred during the function execution.

(ii) `__RTC_InitBase:PROC`

Purpose: This procedure initializes the **RTC runtime environment** at the beginning of program execution. It sets up the necessary checks for stack, memory, and other runtime-related errors, enabling the runtime checkers to monitor the program.

Why it's important:

- Without this initialization, the RTC would not be able to monitor the stack or memory, making it impossible to catch issues such as stack corruption or illegal memory accesses during runtime.

Related Assembly Line:

```
80 __RTC_InitBase.rtc$IMZ DD FLAT:__RTC_InitBase
```

Explanation: This line sets up a segment (`rtc$IMZ`) for RTC initialization. `__RTC_InitBase` is used to prepare the environment for runtime checks before the main logic of the program starts.

(iii) `__RTC_Shutdown:PROC`

Purpose: This procedure cleans up and shuts down the **RTC environment** at the end of the program's execution. It ensures that all the RTC checks have completed successfully and frees any resources that might have been allocated for runtime checks.

Why it's important:

- After the program finishes, `__RTC_Shutdown` ensures a graceful shutdown of the runtime checking system. It also verifies that there are no lingering runtime errors (e.g., unfreed memory or stack imbalances) before the program fully exits.

Related Assembly Line:

```
75 __RTC_Shutdown.rtc$TMZ DD FLAT:__RTC_Shutdown
```

Explanation: This line sets up a segment (`rtc$TMZ`) for shutting down the RTC checks after the program's execution. `__RTC_Shutdown` ensures that all runtime checkers have completed their operations and finalizes the runtime environment gracefully.

Summary of RTC Roles and Procedures

- **RTC's Roles**: Ensure runtime checks for stack integrity, memory management, and initialization/shutdown of the RTC environment, thus catching runtime errors.
- **__RTC_CheckEsp:PROC**: Ensures the stack pointer (ESP) is balanced, detecting stack corruption.
- **__RTC_InitBase:PROC**: Initializes the RTC environment before the program starts, enabling runtime checks.
- **__RTC_Shutdown:PROC**: Shuts down and cleans up the RTC environment after the program ends, verifying that no issues were left unresolved.