Department of Computer Science
Ashoka University

Programming Language Design and Implementation: CS1319

Quiz 1

Assign Date: *Sep 23, 2023* | Marks: *100*

Instructions:

1. The quiz will be physical, on paper, and in the classroom.
2. Write your name and Ashoka ID on the answer-script and additional papers.
3. The quiz comprises two questions (totalling 100 marks) and one bonus question (for 10 marks). Each question has multiple parts with marks shown for each.
4. The quiz is Open book, Open notes, and Open Internet.
5. Laptops will be allowed during the quiz. No mobile phone will be allowed.
6. Any copy from peers will be dealt with zero tolerance - both to get zero in the question. Consultations or chats with others will lead to zero score for the entire quiz.
7. The assembly codes for Question 2 are given in pages 4 and 5. Use them to answer Q 2. Tear them, write your name and Ashoka ID on both sheets, and attach to your answer-script.
8. No question or doubt will be entertained. If you have any query, make your own assumptions, state them clearly in your answer and proceed.
9. Write in clear handwriting and in an unambiguous manner. If TAs have difficulty reading / understanding your answer, they will make assumptions at their best capacity to evaluate. You would not get an opportunity for explanation or rebuttal.

1. Consider the following Flex specification:

```
%{
/* C Declarations and Definitions */
%}
 /* Regular Expression Definitions */
INT      "int"
RET      "return"
ID       [a-z][a-z0-9]*
PUNC     [;]
CONST    [0-9]+
WS       [ \t\n]
/* Definitions of Rules \& Actions */
%%
"do"     { printf("<KEYWORD, do> "); /* Keyword Rule */ }
{INT}    { printf("<KEYWORD, int> "); /* Keyword Rule */ }
{RET}    { printf("<KEYWORD, return> "); /* Keyword Rule */ }
"while"  { printf("<KEYWORD, while> "); /* Keyword Rule */ }
{ID}     { printf("<ID, %s> ", yytext); /* Identifier Rule & yytext points to lexeme */}
"*"      { printf("<OPERATOR, *> "); /* Operator Rule */ }
"/"      { printf("<OPERATOR, /> "); /* Operator Rule */ }
"+"      { printf("<OPERATOR, +> "); /* Operator Rule */ }
"-"      { printf("<OPERATOR, -> "); /* Operator Rule */ }
"="      { printf("<OPERATOR, => "); /* Operator Rule */ }
">"      { printf("<OPERATOR, >>");  /* Operator Rule */ }
"{"      { printf("<SPECIAL SYMBOL, {> "); /* Scope Rule */ }
"}"      { printf("<SPECIAL SYMBOL, }> "); /* Scope Rule */ }
"("      { printf("<SPECIAL SYMBOL, (> "); /* Parenthesis Rule */ }
")"      { printf("<SPECIAL SYMBOL, )> "); /* Parenthesis Rule */ }
{PUNC}   { printf("<PUNCTUATION, ;> "); /* Statement Rule */ }
{CONST}  { printf("<INTEGER CONSTANT, %s> ",yytext); /* Literal Rule */ }
"\n"     { printf("\n"); /* New line Rule */ }
{WS}     /* White-space Rule */ ;
%%
/* C functions */
main() { yylex(); /* Flex Engine */ }
```

(a) A lexer is generated from the above specification. Write the lexer output for the following input file (preserve the lines in the output as generated). **[20]**

```
int abs(int);
int func(int value) {
    int root = value / 2 + 1;
    int temp;
    do {
        temp = root;
        root = (temp + value / temp) / 2;
    } while (abs(root - temp) > 1);
    return root;
}
```

```
<KEYWORD, int> <ID, abs> <SPECIAL SYMBOL, (> <KEYWORD, int> <SPECIAL SYMBOL, )> <
    PUNCTUATION, ;>
<KEYWORD, int> <ID, func> <SPECIAL SYMBOL, (> <KEYWORD, int> <ID, value> <SPECIAL SYMBOL,
     )> <SPECIAL SYMBOL, {>
<KEYWORD, int> <ID, root> <OPERATOR, => <ID, value> <OPERATOR, /> <INTEGER CONSTANT, 2> <
    OPERATOR, +> <INTEGER CONSTANT, 1> <PUNCTUATION, ;>
<KEYWORD, int> <ID, temp> <PUNCTUATION, ;>
<KEYWORD, do> <SPECIAL SYMBOL, {>
<ID, temp> <OPERATOR, => <ID, root> <PUNCTUATION, ;>
```

```
<ID, root> <OPERATOR, => <SPECIAL SYMBOL, (> <ID, temp> <OPERATOR, +> <ID, value> <
    OPERATOR, /> <ID, temp> <SPECIAL SYMBOL, )> <OPERATOR, /> <INTEGER CONSTANT, 2> <
    PUNCTUATION, ;>
<SPECIAL SYMBOL, }> <KEYWORD, while> <SPECIAL SYMBOL, (> <ID, abs> <SPECIAL SYMBOL, (> <
    ID, root> <OPERATOR, -> <ID, temp> <SPECIAL SYMBOL, )> <OPERATOR, >> <INTEGER
    CONSTANT, 1> <SPECIAL SYMBOL, )> <PUNCTUATION, ;>
<KEYWORD, return> <ID, root> <PUNCTUATION, ;>
```

(b) What will be the change in the output for each of the following changes in the Flex specification (one at a time): **[1 + 1 + 1 + (1 + 1) = 5]**

- If the action on the `Rule of Newline` is changed to just a semicolon (;)

  Everything prints on a single line instead of different lines.

- `Rule of RET` is removed

  All instances of the word `return` will change from `<KEYWORD, return>` to `<ID, return>` in the output since the string `return` matches the rule of {`ID`}

- `Rule of WS` is removed

  Every instance of a whitespace in the file to be tokenised is replicated in the output of the Flex program.

- Suppose in the input program the string `value` is replaced with the string `do`. What will be the output of the lexer
  - if the flex specification is as given?

    The Flex program would interpret the word `do` as a keyword. Hence the output would change from `<ID, value>` to `<KEYWORD, do>`.

  - if the flex specification is changed by swapping the rule of `DO` with the rule of `ID`?

    The Flex program would replace every instance of `<ID, %s>` where `%s` is some identifier with `<KEYWORD, do>`. The program would also replace any instance of `<KEYWORD, do>` with `<ID, do>`.

You do not need to write the entire output. Just mention the changes, if any, in every case.

2. Consider the C program:

```c
#include <stdio.h>

int f(int n) { // n >= 0
    int c = 10;
    if (0 == n)
        return n + c;
    else
        return f(f(n - 1)-c);
}
int main() {
    int n = 4, r;
    r = f(n);
    return 0;
}
```

The above program is compiled without optimization by MS-VC to generate an assembly file containing declarations and instructions. The assembly listing is given in the next two pages. Use the listing to answer the following questions.

(a) Draw the call graph of the program starting with `main()` as the root. Mark every call node on the graph with the name of the function called, the value of its parameter, the name of the caller and the value of the parameter of the caller. **[9 * 1.5 = 13.5]** For example, if `n = 3`, the inner call to `f` in `f(f(n - 1) - c)` is marked as "f(2): f(3)", while the outer call to `f` is marked as "f(<return value of inner call> - c): f(3)".

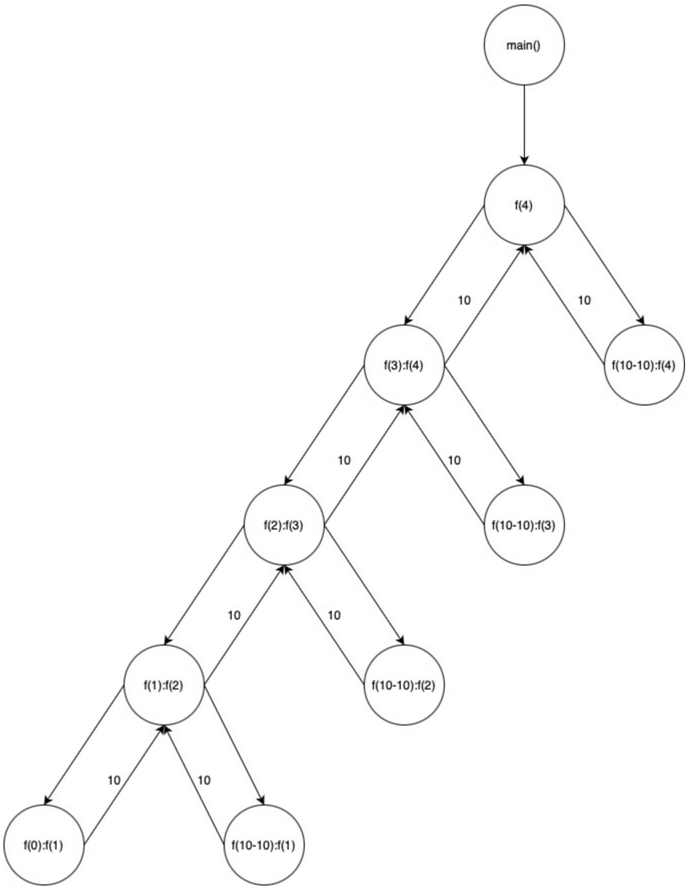

Figure 1: Call graph

This can also be written as: f(4): main()
f(3): f(4)
f(2): f(3)
f(1): f(2)
f(0): f(1)
f(0): f(1)
f(0): f(2)
f(0): f(3)
f(0): f(4)

(b) Generate the Symbol Tables for global scope and functions `main()` & `f()`. **[3 * 1.5 = 4.5]**

ST.glb		*Parent = None*	
`f`	int → int		
		func	0
`main`	void → int		
		func	0
ST.f()		*Parent = ST.glb*	
`n`	int	param	4
`c`	int	local	4
`f`	int → int		
		func	0
ST.main()		*Parent = ST.glb*	
`n`	int	local	4
`r`	int	local	4
`f`	int → int		
		func	0

Note: This is a simplified answer which does not use TAC form of given code.

(c) Assume that `esp = 0x00001000` when `f(2)` has been called. Show the state of the stack with the layout of activation records when the invocation of `f(1)`, called from `f(2)`, is in execution. Show every entry of the activation records with addresses, and the values of `esp` and `ebp`. Repeat when `f(0)`, called from `f(2)`, is in execution. **[6 + 6 = 12]**

You may skip the leading 4 nibbles of addresses for brevity. That is, write `0x00001000` *simply as* `0x1000`.

Addr	**Offset**	**Item**	esp
	ebp = 0x0FE4 = 4068		
0x0FE4 = 4068		ebp	0x0FE4
0x0FE0 = 4064		RA	0x0FE0
0x0FE8 = 4072	+8	n=0	
0x0FEC = 4076	+12	c=10	

AR of `f(0)` called by `f(2)`

Addr	**Offset**	**Item**	esp
0x0FF0 = 4080	−8	n=1	
0x0FF4 = 4084	−4	c=10	0x0FEC
	ebp = 0x0FF8 = 4088		
0x0FF8 = 4088		ebp	0x0FF8
0x0FFC = 4092		RA	0x0FFC

AR of `f(1)` called by `f(2)`

0x1000 = 4096			0x1000

Note:

- The leading 4 nibbles of addresses for brevity: `0x00001000` is written simply as `0x1000`.
- The addresses are in hex format (`0x00001000` = `4096`). Hence, `0x1000 - 4` = `0x1000 - 0x4` = `0x0FFC` and so on. Addresses are also shown in decimal for understanding. Consistent use of either hexadecimal or decimal is acceptable.
- The stack should be read from bottom to top (upwards).

(d) Annotate each line of the assembly listing of function `f()` (lines #50 to #107) to explain the functionality of the instruction and the connection to the original C program. Skip annotations for comment lines, blank lines, C source lines, labels and assembly directives (like `_TEXT`).

Mark the activation record definitions, prologue and epilogue for function `f()` clearly. Highlight the call and return mechanisms. [**35 + (7 + 3) = 45**]

The assembly instructions have been discussed in the class and / or the workout example.

3. **Bonus Problem**: Consider the function `f()` in Question 2.

(a) Prove by induction that `f(n) = c` for any `n >= 0`. [**4**]
We know $f(0)$, base case is true. Using Inductive Hypothesis, we know $f(n)$ is true.
Hence,

$$f(n) = c$$

Thus,

$$f(n+1) = f(f(n) - c) = f(c - c) = f(0) = c$$

(b) What will be the behaviour of `main()` in the following cases? [**3 * 2 = 6**]

i. `int c = 10` in line #4 is changed to `int c`? That is, `c` is left uninitialized in the function scope.
Accessing an uninitialized variable is **undefined behaviour**.

ii. `int c = 10` from line #4 is moved to the global scope before `int f(int n)`? That is, `c` is moved to the global scope and initialized.
No change in behaviour

iii. `int c = 10` from line #4 is moved to the global scope before `int f(int n)` and changed to `int c`? That is, `c` is moved to the global scope and left uninitialized.
No change in behaviour, except $f(n) = 0, \forall n \geq 0$ as global variables are implicitly initialized to 0.

*The credit for a bonus problem (**10** here) is not counted in the total of **100**. However, marks scored in a bonus problem will be added to total score (capped, of course, at **100**).*

Assembly Listing

Use the following assembly language translation of the program from Q 2 where the C source lines are interspersed as comments.

Tear down this and the next page and use to answer Q 2. Write your name and Ashoka ID on both sheets and attach to your answer-script.

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 18.00.21005.1
; Several lines skipped for brevity -- PPD

PUBLIC _f
PUBLIC _main
; Several extern information lined skipped for brevity -- PPD
; File fakefact.c
_TEXT SEGMENT
_r$ = -20                               ; size = 4
_n$ = -8                                ; size = 4
_main PROC                              ; COMDAT

; 10   : int main() {
    push    ebp
    mov     ebp, esp
    sub     esp, 216                    ; 000000d8H
    push    ebx
    push    esi
    push    edi
    lea     edi, DWORD PTR [ebp-216]
    mov     ecx, 54                     ; 00000036H
    mov     eax, -858993460             ; ccccccccH
    rep stosd

; 11   :    int n = 4, r;
    mov     DWORD PTR _n$[ebp], 4

; 12   :    r = f(n);
    mov     eax, DWORD PTR _n$[ebp]
    push    eax
    call    _f
    add     esp, 4
    mov     DWORD PTR _r$[ebp], eax

; 13   :    return 0;
    xor     eax, eax

; 14   : }
    pop     edi
    pop     esi
    pop     ebx
    add     esp, 216                    ; 000000d8H
    cmp     ebp, esp
    call    __RTC_CheckEsp
    mov     esp, ebp
    pop     ebp
    ret     0
_main   ENDP
_TEXT   ENDS
```

```
; File fakefact.c
_TEXT SEGMENT
_c$ = -8                                ; size = 4
_n$ = 8                                 ; size = 4
_f PROC                                 ; COMDAT

; 3    : int f(int n) { // n >= 0
    push    ebp
    mov     ebp, esp
    sub     esp, 204                    ; 000000ccH
    push    ebx
    push    esi
    push    edi
    lea     edi, DWORD PTR [ebp-204]
    mov     ecx, 51                     ; 00000033H
    mov     eax, -858993460             ; ccccccccH
    rep stosd

; 4    :    int c = 10;
    mov     DWORD PTR _c$[ebp], 10      ; 0000000aH

; 5    :    if (0 == n)
    cmp     DWORD PTR _n$[ebp], 0
    jne     SHORT $LN2@f

; 6    :        return n + c;
    mov     eax, DWORD PTR _n$[ebp]
    add     eax, DWORD PTR _c$[ebp]
    jmp     SHORT $LN3@f

; 7    :    else
    jmp     SHORT $LN3@f
$LN2@f:

; 8    :        return f(f(n - 1)-c);
    mov     eax, DWORD PTR _n$[ebp]
    sub     eax, 1
    push    eax
    call    _f
    add     esp, 4
    sub     eax, DWORD PTR _c$[ebp]
    push    eax
    call    _f
    add     esp, 4
$LN3@f:

; 9    : }
    pop     edi
    pop     esi
    pop     ebx
    add     esp, 204                    ; 000000ccH
    cmp     ebp, esp
    call    __RTC_CheckEsp
    mov     esp, ebp
    pop     ebp
    ret     0
_f ENDP
_TEXT ENDS
```

The annotations start with ;;
You will receive points if you have done the same annotations but have used different terminology such as `comparison bit` instead of `condition flag/code`

```
; File fakefact.c
_TEXT SEGMENT        ;; DESCRIPTION OF STACK FRAME OF f begins from next line
_c$ = -8                         ; size = 4 ;; -8 is offset for c. Address of c is ebp-8
_n$ = 8                          ; size = 4 ;; 8 is offset for n. Address of n is ebp+8
_f PROC                          ; COMDAT ;; Function f code starts

; 3     : int f(int n) { // n >= 0 ;; PROLOGUE OF f STARTS from next line
    push     ebp       ;; save ebp to remember the frame information for caller
    mov      ebp, esp     ;; initialize ebp to esp so frame of this function will be allocated from here
    sub      esp, 204                ; 000000ccH ;; reserve 204 bytes on the stack for the frame of f
    push     ebx       ;; save ebx on stack
    push     esi       ;; save esi on stack
    push     edi       ;; save edi on stack
    lea      edi, DWORD PTR [ebp-204]     ;; stores address [ebp-204] in the edi register
    mov      ecx, 51                 ; 00000033H ;;store 51 in ecx since we reserved 51 * 4 bytes on stack
    mov      eax, -858993460         ; ccccccccH ;; set eax to uninitialized value (garbage) marker
    rep stosd    ;; stores value stored in eax in the address stored in edi, increments edi by 4 bytes, decrements value in ecx by 1, repeats till ecx is 0 ;; this basically marks all values in stack as uninitialized

; 4     :     int c = 10;
    mov      DWORD PTR _c$[ebp], 10   ; 0000000aH ;; initialize c to 10, ;; PROLOGUE OF f ENDS

; 5     :     if (0 == n)
    cmp      DWORD PTR _n$[ebp], 0    ;; compares the value of n with 0 and sets condition flag/code
    jne      SHORT $LN2@f     ;; checks condition flag/code and if n != 0, jump to label $LN2@f

; 6     :         return n + c;
    mov      eax, DWORD PTR _n$[ebp] ;; stores n in eax
    add      eax, DWORD PTR _c$[ebp] ;; adds c to value in eax (that is, n) and stores result (n + c) in eax
    jmp      SHORT $LN3@f     ;; jump to label $LN3@f (function epilogue) since there is a return in the C code

; 7     :     else
    jmp      SHORT $LN3@f     ;; jump to label $LN3@f used to skip the code for line 8 of the C code
$LN2@f:

; 8     :         return f(f(n - 1)-c);
    mov      eax, DWORD PTR _n$[ebp] ;; stores n in eax
    sub      eax, 1                  ;; eax = eax - 1, so eax = n - 1
    push     eax                     ;; push eax (n - 1) to stack to use as parameter for f
    call     _f           ;; call f, which uses the topmost value (n - 1) in the stack as parameter
```

```
    add     esp, 4                  ;; increment stack pointer by 4 to remove used parameter (n - 1) from stack
    sub     eax, DWORD PTR _c$[ebp] ;; eax has return value from previous call to f, so it has f(n - 1) and this instruction does eax = f(n - 1) - c
    push    eax         ;; push eax (f(n - 1) - c) to stack as parameter for f
    call    _f          ;; call f, which uses parameter (f(n - 1) - c)
    add     esp, 4      ;; increment esp by 4 to remove used parameter from stack
$LN3@f:

; 9    : }      ;; EPILOGUE OF f STARTS from next line
    pop     edi     ;; restore edi from stack
    pop     esi     ;; restore esi from stack
    pop     ebx     ;; restore ebx from stack
    add     esp, 204                ; 000000ccH ;; release 204 bytes of the frame of f
    cmp     ebp, esp        ;; compare ebp (in which we stored value of esp before reserving frame of f) with esp and set condition flag/code
    call    __RTC_CheckEsp  ;; check the condition code to confirm that esp matches its value before call. This is a system check for correctness
    mov     esp, ebp        ;; restore esp
    pop     ebp             ;; restore ebp (the frame of the caller function)
    ret     0               ;; Return 0. Control returns through indirect jump ;; EPILOGUE OF f ENDS
_f ENDP                 ;; Function f code ends
_TEXT ENDS
```