Department of Computer Science
Ashoka University

Programming Language Design and Implementation (PLDI): CS-1319-1

Quiz - 4: Offline — *Marks: 100*
Date: *December 17, 2022* — Time: *16:30 - 18:30* **NO EXTENSION**

Instructions:

1. The quiz can be taken from anywhere.
2. The quiz comprises one question (totalling 100 marks) and one bonus question (for 10 marks). Each question has multiple parts with marks shown for each.
3. The quiz is Open book, Open notes, and Open Internet.
4. Any copy from peers will be dealt with zero tolerance - both to get zero in the question.
5. The quiz will be available from 16:25.
6. The quiz must be submitted within 18:10.
7. No late or email or physical submission will be entertained.
8. You may choose between two options to prepare your answers:
   (a) Write the answers on paper. Take snaps and prepare a single PDF.
   (b) Alternately, you may type your answer in an editor (Notepad / WORD etc.) and save as PDF.
   - Naturally, you may use a mixed style too (like drawing a diagram on paper and include its image in the file you are typing).
   - Whatever way you prepare the answers, all should be made into a single PDF file.
9. Remember to write Name & Roll Number on every page and put pages in right order.
10. The PDF should be named with roll number and uploaded.
11. No question or doubt will be entertained. If you have any query, make your own assumptions, state them clearly in your answer and proceed.
12. 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 strings comprising `a`'s and `b`'s. Such a string `s` is called a palindrome if it reads the same from left-to-right and right-to-left. For example, `s1` = `abba` and `s2` = `ababa` are palindromes while `s3` = `abb` is not.

   (a) Can we write regular definition for palindrome (as above) in Flex notation? If yes, write the regular definition. If no, justify. **[10]**

   BEGIN SOLUTION No.

   We consider the cases of odd (has a center-marker) and even (has no center-marker) palindromes separately.

   - **Even palindromes**: Let us assume that the length of the palindrome is $2n$, $n > 0$. To match the forward scan with the reverse, we need to remember the symbol scanned in every step of forward scan. For this we need an FSM M with at least $n + 1$ states.
     Once we reach the final state in this FSM, we can make reverse transitions if the input symbol matches the forward transition. If we reach the start state finally and the string is over, we have found a palindrome.
     Now consider, that we present an input of length $2n+2$ to M. Understandably, after reading the first n symbols M will assume that the string is folding on the $n+1^{st}$ symbol which will be wrong.
     Bottom-line, we need a M machine with $n+1$ states for $n > 0$. This is not finite (we at least need a stack) and hence not regular.
   - **Odd palindromes**: Similar to the earlier case.

   Note that the infeasibility can be formally argued using the Pumping Lemma of FSMs.

   END SOLUTION

   (b) Can we write a grammar for palindrome (as above) in Bison notation? If yes, write the grammar in Bison. If no, justify. **[10]**

   BEGIN SOLUTION

   Yes.

   The grammar for palindromes will be:

   $$
   \begin{array}{lcll}
   S & \rightarrow & aSa & \\
   S & \rightarrow & bSb & \\
   S & \rightarrow & a & \text{// Odd palindromes with centermarker } a \\
   S & \rightarrow & b & \text{// Odd palindromes with centermarker } b \\
   S & \rightarrow & \epsilon & \text{// Even palindromes}
   \end{array}
   $$

   Hence, in Bison notation we get:

   ```
   S : aSa | bSb | a | b | ;
   ```

   END SOLUTION

2. Suppose that you have designed a language recurC that supports recursive functions, arrays, `struct`'s and `if-else` statements (along with arithmetic and Boolean expressions), but no loop construct like `while`, `for` or `do-while`.

   Prove that any C function can be written in recurC language. **[10]**

   BEGIN SOLUTION

   Only loops are missing from recurC. So let us check if loops of C can be expressed using the features of recurC.

   Note that C language supports three loop constructs. Let us consider each below:

   - **FOR loop**: Consider:

     ```
     for(E1; E2; E3) B;
     ```

where each `E` is an expression of type `expression` and `B` is the loop-body – a single statement. This can be expressed using `while` as:

```
E1;
while (E2) {
    B;
    E3;
}
```

- **DO-WHILE loop**: Consider:

```
do {
    B;
} while E;
```

This can be expressed using `while` as:

```
B;
while (E) {
    B;
}
```

- **WHILE loop**: Consider:

```
while (E)
    B;
```

This can be expressed using a recursive function, say `fWhile()`, and `if` as:

```
void fWhile(expression E) {
    if (E) {
        B;
        fWhile(E);
    }
    return;
}
```

Hence, all loops can be simulated using recursive functions and `if`.

END SOLUTION

3. Consider the following program:

```
int main() {
    int a[10], b[10], c[10];
    int n, i;

    n = 9;

    for (i = 0; i <= n; ++i) {
        a[i] = i;
        b[i] = i * i;
        c[i] = a[i] + b[i];
    }
    return 0;
}
```

(a) Translate the above program to three address codes:

i. Show the Global Symbol Table with the symbol name, data type, category, and size. Mark appropriate parent / child symbol table pointers to build the tree of symbol tables. [**2**]

BEGIN SOLUTION

ST.glb				Parent: *Null*
Name	**Type**	**Category**	**Size**	**Offset**
`main`	void → int	func	0	ST.main

END SOLUTION

ii. Write the array of quad codes starting at index 100. [**20**]

BEGIN SOLUTION

```
// TAC of main
L100: t00 = 9
L101: n = t00
L102: t01 = 0
L103: i = t01
L104: if i <= n goto L108
L105: goto L124
L106: i = i + 1
L107: goto L104
L108: t02 = i * 4
L109: t03 = a + t02
L110: *t03 = i
L111: t04 = i * 4
L112: t05 = b + t04
L113: t06 = i * i
L114: *t05 = t06
L115: t07 = i * 4
L116: t08 = c + t07
L117: t09 = i * 4
L118: t10 = a[t09]
L119: t11 = i * 4
L120: t12 = b[t11]
L121: t13 = t10 + t12
L122: *t08 = t13
L123: goto L106
L124: ret 0
```

END SOLUTION

iii. Show the Symbol Table for function `main` with the symbol name, data type, category, size, and offset. Mark appropriate parent / child symbol table pointers to build the tree of symbol tables. [**3**]

BEGIN SOLUTION

ST.main				Parent: *ST.glb*
Name	**Type**	**Category**	**Size**	**Offset**
a	array(10, int)	local	40	-4
b	array(10, int)	local	40	-44
c	array(10, int)	local	40	-84
n	int	local	4	-88
i	int	local	4	-92
t00	int	temp	4	-96
t01...t13	int	temp	4×13	-100...-152

END SOLUTION

(b) Peephole optimize the code of function `main` and renumber the optimized quads from `L100`. Optimize for the following: [**10**]

- propagating copies and removing dead code

Given Code	**Optimized Code**
`x = a + b` `y = x`	`y = a + b`

- folding constants

Given Code	**Optimized Code**
`x = 3 + 1`	`x = 4`

- short-cutting jump-to-jump

Given Code	**Optimized Code**

`[110]: goto 113`
`...`
`[113]: goto 119`
	`[110]: goto 119`

- eliminating jump-over-jump

Given Code	**Optimized Code**

`[110]: if a > b goto 112`
`[111]: goto 120`
`[112]: x = 0`
	`[110]: if a <= b goto 120`
	`[112]: x = 0`

- applying algebraic simplification

Given Code	**Optimized Code**
`x = p + 0`	`x = p`

- applying strength reduction

Given Code	**Optimized Code**
`a = 4 * i`	`a = i << 2`

BEGIN SOLUTION

Making a pass over the quad array, we peep and optimize the quads. The quads affected are marked with the optimization action in comments.

```
// Peep-optimized TAC of main
      L100: t00 = 9       // Deadcode
L100: L101: n = 9         // Copy propagate
      L102: t01 = 0       // Deadcode
L101: L103: i = 0         // Copy propagate
L102: L104: if i <= n goto L106 (L108)
L103: L105: goto L121 (L124)
L104: L106: i = i + 1
L105: L107: goto L102 (L104)
L106: L108: t02 = i << 2 // Strength reduction
L107: L109: t03 = a + t02
L108: L110: *t03 = i
L109: L111: t04 = i << 2 // Strength reduction
L110: L112: t05 = b + t04
      L113: t06 = i * i  // Deadcode
L111: L114: *t05 = i * i // Copy propagate
L112: L115: t07 = i << 2 // Strength reduction
L113: L116: t08 = c + t07
L114: L117: t09 = i << 2 // Strength reduction
L115: L118: t10 = a[t09]
L116: L119: t11 = i << 2 // Strength reduction
L117: L120: t12 = b[t11]
L118: L121: t13 = t10 + t12
L119: L122: *t08 = t13
L120: L123: goto L104 (L106)
L121: L124: ret 0
```

END SOLUTION

(c) Construct the Control Flow Graph (CFG) for function `main`. **[3 + 7 = 10]**

i. Identify the leader quads

BEGIN SOLUTION

```
// Headers in TAC of main
L100: n = 9
L101: i = 0
L102: if i <= n goto L106
L103: goto L121
L104: i = i + 1
L105: goto L102
L106: t02 = i << 2
L107: t03 = a + t02
L108: *t03 = i
L109: t04 = i << 2
L110: t05 = b + t04
L111: *t05 = i * i
L112: t07 = i << 2
L113: t08 = c + t07
L114: t09 = i << 2
L115: t10 = a[t09]
L116: t11 = i << 2
L117: t12 = b[t11]
L118: t13 = t10 + t12
L119: *t08 = t13
L120: goto L104
L121: ret 0
```

END SOLUTION

ii. Construct the basic blocks and build the CFG.

BEGIN SOLUTION

```
// CFG of main
```

Block B1

```
L100: n = 9
L101: i = 0
```

goto B2

Block B2

```
L102: if i <= n goto L106 (goto B5)
```

goto B3

Block B3

```
L103: goto L121 (goto B6)
```

Block B4

```
L104: i = i + 1
L105: goto L102 (goto B2)
```

Block B5

```
L106: t02 = i << 2
L107: t03 = a + t02
L108: *t03 = i
L109: t04 = i << 2
L110: t05 = b + t04
L111: *t05 = i * i
L112: t07 = i << 2
L113: t08 = c + t07
L114: t09 = i << 2
L115: t10 = a[t09]
L116: t11 = i << 2
L117: t12 = b[t11]
L118: t13 = t10 + t12
L119: *t08 = t13
L120: goto L104 (goto B4)
```

Block B6

```
L121: ret 0
```

END SOLUTION

(d) For every block that contains a reference to array `a`, `b`, or `c`, optimize the block for local common sub-expression, copy propagation and dead-code elimination. Present the CFG with the optimized code. **[8 + 2 = 10]**

BEGIN SOLUTION

Since all references to the arrays are limited only within **Block B5**, we consider this block for optimizations by value-numbering.

```
// Block B5 of main - I/p for optimization
L106: t02 = i << 2
L107: t03 = a + t02
L108: *t03 = i
L109: t04 = i << 2
L110: t05 = b + t04
L111: *t05 = i * i
L112: t07 = i << 2
L113: t08 = c + t07
L114: t09 = i << 2
L115: t10 = a[t09]
L116: t11 = i << 2
L117: t12 = b[t11]
L118: t13 = t10 + t12
L119: *t08 = t13
```

VN Table

Name	VN
i	1
t02	2
a	3
t03	4
*t03	5
t04	2
b	6
t05	7
*t05	8
t07	2
c	9
t08	10
t09	2
t10	11
t11	2
t12	12
t13	13
*t08	14

Name Table

Index	Name	Val
1	i	
2	t02, t04, t07, t09, t11	
3	a	
4	t03	
5	*t03	
6	b	
7	t05	
8	*t05	
9	c	
10	t08	
11	t10	
12	t12	
13	t13	
14	*t08	

Hash Table

Expr	VN
i << 2	2
a + t02	4
*t03	5
b + t02	7
i * i	8
*t05	8
c + t02	10
a[t02]	11
b[t02]	12
t10 + t12	13
*t08	14

Block B5 of main

```
// O/p of VN Optimization
L106: t02 = i << 2
L107: t03 = a + t02
L108: *t03 = i
L109: t04 = i << 2 // Deadcode
L110: t05 = b + t02
L111: *t05 = i * i
L112: t07 = i << 2 // Deadcode
L113: t08 = c + t02
L114: t09 = i << 2 // Deadcode
L115: t10 = a[t02]
L116: t11 = i << 2 // Deadcode
L117: t12 = b[t02]
L118: t13 = t10 + t12
L119: *t08 = t13
```

```
// Renumbered w/o deadcode
L106: L106: t02 = i << 2
L107: L107: t03 = a + t02
L108: L108: *t03 = i
L109: L110: t05 = b + t02
L110: L111: *t05 = i * i
L111: L113: t08 = c + t02
L112: L115: t10 = a[t02]
L113: L117: t12 = b[t02]
L114: L118: t13 = t10 + t12
L115: L119: *t08 = t13
```

```
// Optimized Block B5
L106: t02 = i << 2
L107: t03 = a + t02
L108: *t03 = i
L109: t05 = b + t02
L110: *t05 = i * i
L111: t08 = c + t02
L112: t10 = a[t02]
L113: t12 = b[t02]
L114: t13 = t10 + t12
L115: *t08 = t13
```

No change in other blocks of CFG

END SOLUTION

4. Consider the following code snippet:

```
a = 2
b = 3
c = a * b
b = a + 1
d = 4
c = c + d
a = b + c
```

(a) Convert the code to symbolic registers (`s1`, `s2`, etc.) so that multiple definitions to the same variable are treated as separate symbolic registers. [**2**]

BEGIN SOLUTION

END SOLUTION

(b) Compute the live intervals for the symbolic registers. [**2**]

BEGIN SOLUTION

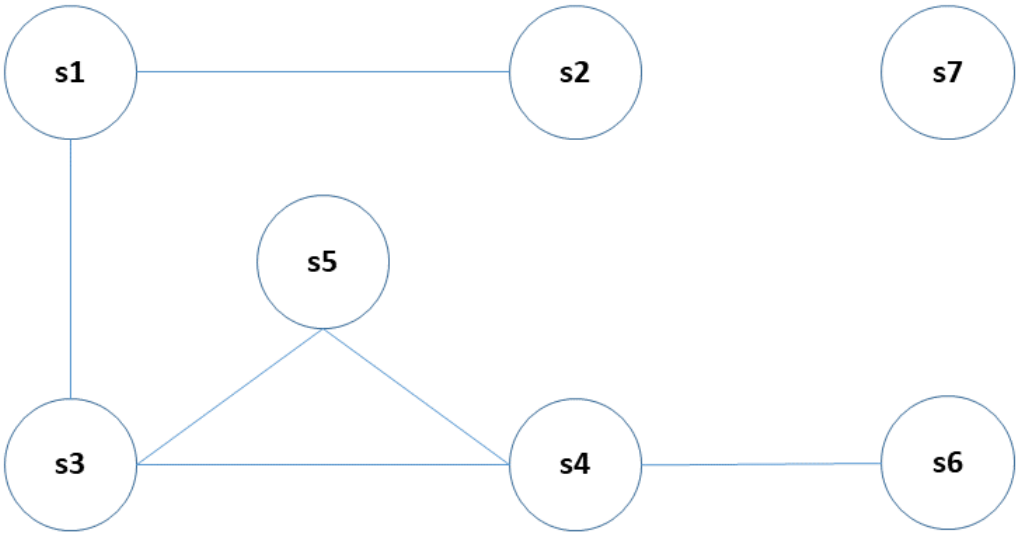
`1: s1 = 2`	`s1:` **1-4**
`2: s2 = 3`	`s2:` **2-3**
`3: s3 = s1 * s2`	`s3:` **3-6**
`4: s4 = s1 + 1`	`s4:` **4-7**
`5: s5 = 4`	`s5:` **5-6**
`6: s6 = s3 + s5`	`s6:` **6-7**
`7: s7 = s4 + s6`	`s7:` **7-**

END SOLUTION

(c) Construct the register interference graph for the symbolic registers. [**2**]

BEGIN SOLUTION

Based on the liveness intervals, here is the RIG:

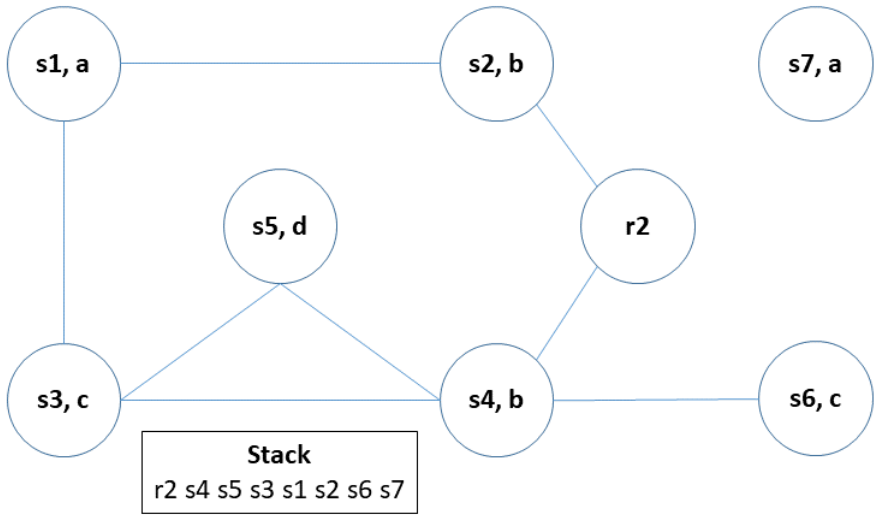


END SOLUTION

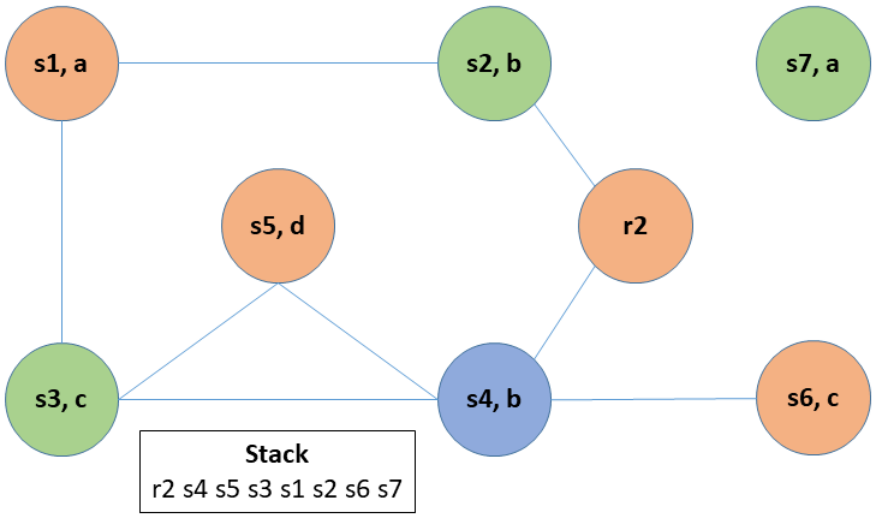
(d) You are given `k` registers - `r1`, `r2`, ... `rk`. Using Chaitin's graph coloring algorithm, find the minimum value of `k` for a register assignment without spilling if variable `b` is not allowed to be assigned to register `r2`. [**5**]

BEGIN SOLUTION

We modify the RIG by adding a node for `r2` and edges connecting symbolic registers that hold values of `b`.



This RIG has a 3-clique. So we know $k \geq 3$ and try coloring with $k = 3$. The trace of the stack (left end is the top) is shown in the diagram. We find that it is 3-colorable. So $k_{min} = 3$.



Based on the color, here is the assignment:

Variable	Symbolic Register	Machine Register
a, d, c	s1, s5, s6	r_2
b, c, a	s2, s3, s7	r_1
b	s4	r_0

END SOLUTION

(e) Translate the code for the snippet using the registers as assigned by you. [4]

BEGIN SOLUTION

```
a = 2          1: s1 = 2       a: s1                  1: r2 = 2
b = 3          2: s2 = 3       a: s1 b: s2            2: r1 = 3
c = a * b      3: s3 = s1 * s2 a: s1 b: s2 c: s3      3: r1 = r2 * r1
b = a + 1      4: s4 = s1 + 1  a: s1 b: s4 c: s3      4: r0 = r2 + 1
d = 4          5: s5 = 4       b: s4 c: s3 d: s5      5: r2 = 4
c = c + d      6: s6 = s3 + s5 b: s4 c: s3/s6 d: s5   6: r2 = r1 + r2
a = b + c      7: s7 = s4 + s6 a: s7 b: s4 c: s6      7: r1 = r0 + r2
```

END SOLUTION