INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

	Stamp / Signature of the Invigilator

EXAMINATION (End Semester)	SEMESTER (Autumn)

Roll Number									Section		Name	
Subject Number	C	S	3	1	0	0	3	Subject Name				*Compilers*
Department / Center of the student											Additional Sheets	

Important Instructions and Guidelines for Students

1. You must occupy your seat as per the Examination Schedule/Sitting Plan.

2. Do not keep mobile phones or any similar electronic gadgets with you even in the switched off mode.

3. Loose papers, class notes, books or any such materials must not be in your possession, even if they are irrelevant to the subject you are taking examination.

4. Data book, codes, graph papers, relevant standard tables/charts or any other materials are allowed only when instructed by the paper-setter.

5. Use of instrument box, pencil box and non-programmable calculator is allowed during the examination. However, exchange of these items of any other papers (including question papers) is not permitted.

6. Write on both sides of the answer script and do not tear off any page. **Use last page(s) of the answer script for rough work.** Report to the Invigilator if the answer script has torn or distorted page(s).

7. It is your responsibility to ensure that you have signed the Attendance Sheet. Keep your Admit Card/Identity Card on the desk for checking by the invigilator.

8. You may leave the examination hall for wash room or for drinking water for a very short period. Record your absence from the Examination Hall in the register provided. Smoking and the consumption of any kind of beverages are strictly prohibited inside the Examination Hall.

9. Do not leave the Examination Hall without submitting your answer script to the invigilator. **In any case, you are not allowed to take away the answer script with you.** After the completion of the examination, do not leave the seat until the invigilators collect all the answer scripts.

10. During the examination, either inside or outside the Examination Hall, gathering information from any kind of sources or exchanging Information with others or any such attempt will be treated as **'unfair means'**. Do not adopt unfair means and do not indulge in unseemly behavior.

Violation of any of the above instructions may lead to severe punishment.

Signature of the Student

To be filled in by the examiner

Question Number	1	2	3	4	5	6	7	8	9	10	Total
Marks obtained											

Marks Obtained (in words)	Signature of the Examiner	Signature of the Scrutineer

CS31003 COMPILERS
AUTUMN 2025 – 2026
END-SEMESTER EXAMINATION
19-NOVEMBER-2025, 2:00PM – 5:00PM
MAXIMUM MARKS: 100

Instructions to students

- Write your answers in the question paper itself.
- Answer **all** questions.
- Write in the blank spaces provided in the questions. Use the empty pages at the end for rough work. Please avoid supplementary sheets. The answers to all the questions must be written in this question paper only. If you continue some answer to the Rough-Work pages, please supply an appropriate pointer in your answer.
- Do not write anything on this page. Questions start from the next page (Page 3).
- If you need to make any assumptions in some questions, write them clearly in your respective answers.
- The Dragon-Book refers to the textbook followed in the class:

  Aho, Lam, Sethi, and Ullman, *Compilers*: *Principles*, *Techniques*, *and Tools*, Second Edition.

1. [Syntax-directed translation]

The following grammar generates non-empty sequences of bits (0's and 1's). Each such sequence is interpreted as a binary representation of a non-negative integer N. For example, the bit sequence 00101 is a binary representation of 5 (decimal).

$$N \rightarrow N\,B \mid B$$
$$B \rightarrow \mathbf{0} \mid \mathbf{1}$$

(a) The non-terminals N and B have a synthesized attribute `parity` (its only allowed values are `EVEN` and `ODD`). For example, the parity of 00101 (the number 5) is `ODD`, and the parity of 10100 (the number 20) is `EVEN`. Provide the semantic actions for computing this attribute. In all the parts, use no extra attributes (except what is given). **[4]**

Production	Semantic action
$N \rightarrow N_1\,B$	`N.parity = B.parity`
$N \rightarrow B$	`N.parity = B.parity`
$B \rightarrow \mathbf{0}$	`B.parity = EVEN`
$B \rightarrow \mathbf{1}$	`B.parity = ODD`

(b) Now, we want to determine whether the input number is a multiple of 3. To that effect, we store against N and B a synthesized attribute `rem3` (with allowed values 0, 1, and 2 only) which stands for the remainder of the number or bit upon division by 3. The input number is divisible by three if and only if `rem3` at the root of the parse tree is 0. Supply the semantic actions for computing `rem3`. Do not use `parity` in this part and in the next part. **[4]**

Production	Semantic action
$N \rightarrow N_1\,B$	`N.rem3 = (2 * N₁.rem3 + B.rem3) % 3`
$N \rightarrow B$	`N.rem3 = B.rem3`
$B \rightarrow \mathbf{0}$	`B.rem3 = 0`
$B \rightarrow \mathbf{1}$	`B.rem3 = 1`

(c) In this part, S generates a signed integer by adding a sign bit (0 means positive, 1 means negative) at the beginning of a binary string (the magnitude of the integer) generated by N. That is, S generates integers in the signed-magnitude representation. We now add a new production $S \rightarrow B\,N$ (B generates the sign bit). Keep the productions and semantic actions for N and B as in Part (b). Write below the semantic action for computing `rem3` (allowed values are 0, 1, 2 only) of S. For example, 100101 stands for $-5 = -2 \times 3 + 1$, and has `rem3` value 1. **[2]**

Production	Semantic action

$S \rightarrow B\,N$	`If B.rem3 = 0, then S.rem3 = N.rem3`
	`else`
	    `if N.rem3 = 0, then S.rem3 = 0`
	    `else S.rem3 = 3 - N.rem3`

2. [Intermediate-code generation]

Translate the following C code snippet to three-address code. Here `a[][]` is a two-dimensional integer array of dimension 100 × 100, and `i`, `j`, and `n` are integers. Assume that each integer has width 4. Strictly follow C-style interpretations of pre- and post-increment operators (++), and of `break` in `switch` statements. **[10]**

```
do {
    switch (n%3) {
        case 0: a[i][++j] = n++; break;
        case 1: a[i++][j] = n;
        case 2: a[i][j] = ++n;
    }
} while (n < 100);
```

```
begin:  t1 = n % 3
        goto test
L1:     t2 = 400 * i
        t3 = j + 1
        j = t3
        t4 = 4 * j
        t5 = t2 + t4
        a[t5] = n
        t6 = n + 1
        n = t6
        goto next
L2:     t7 = 400 * i
        t8 = i + 1
        i = t8
        t9 = 4 * j
        t10 = t7 + t9
        a[t10] = n
L3:     t11 = 400 * i
        t12 = 4 * j
        t13 = t11 + t12
        t14 = n + 1
        n = t14
        a[t13] = t14
        goto next
test:   if t1 == 0 goto L1
        if t1 == 1 goto L2
        if t1 == 2 goto L3
next:   if n < 100 goto begin
```

3. [Backpatching]

Consider the following grammar of multi-way branching using the keywords **if**, **else**, **elif** (abbreviation of `else if`), and **endif**. A branching statement starts with the keyword **if**, and is followed by a parenthesized Boolean condition (B) and then by a list (L) of statements (S). An optional sequence of one or more else-if blocks (each consisting of the keyword **elif**, a parenthesized Boolean expression, and a list of statements, in that sequence) follows. Finally, there is an optional else block (one only). The combined else-if and else part is generated by E. In all the cases, the branching statement ends with the keyword **endif**. In the grammar below, A stands for an assignment statement, X for an expression, and R for a relational operator. The start symbol is L.

L	→	S | L S
S	→	A | **if** **(** B **)** L E **endif**
E	→	ε | **else** L | **elif** **(** B **)** L E
B	→	X R X

Here are some examples of multi-way branching statements generated by this grammar.

```
if (x >= 0) a = 1; endif
if (x >= 0) a = 1; y = x; else a = 0; y = -x; endif
if (x > 0) a = 1; y = x; elif (x < 0) a = -1; y = -x; else a = 0; y = 0; endif
```

(a) Prove/Disprove the following two assertions about this grammar.

(i) This grammar suffers from the dangling-else ambiguity (involving **if** and **else** only). **[2]**

False. The dangling-else ambiguity arises for a statement of the form `if (C1) if (C2) S1 else S2`. This can be interpreted in two ways: `if (C1) { if (C2) S1 else S2 }` and `if (C1) { if (C2) S1 } else S2`. In the current grammar, the second interpretation is invalid because of the keyword **endif**. These two interpretations now should be written in two different ways as follows.

```
if (C1) if (C2) S1 else S2 endif endif
if (C1) if (C2) S1 endif else S2 endif
```

(ii) A multi-statement **if**/**elif**/**else** block needs to be enclosed within braces or other delimiters. **[2]**

False. An **if** block must be followed immediately by one of the keywords **elif**, **else**, and **endif**. An **elif** block must be followed immediately by **elif**, **else** or **endif**. An **else** block must be followed by **endif**. These keywords act as delimiters for the blocks.

(b) We use backpatching (without fall-through optimization) to translate multi-way branching statements to 3-address codes in single passes. The non-terminals L (list of statements), S (statement), and E (combined else-if and else part) maintain a synthesized attribute `nextlist`, and a Boolean expression (B) has two synthesized attributes `truelist` and `falselist`. Each of these lists stores the instruction numbers containing jumps to an unspecified (yet unknown) instruction number. Later when the jump target is available, backpatching is done by adding this target to all the instructions stored in the list. Complete the following table by filling out the semantic actions on the right side, corresponding to the productions on the left side. Use marker non-terminals M and N only, where M.`inst` stores the next instruction number that can be obtained in the variable `nextinst`, and N has a `nextlist`. Use the functions `makelist()`, `merge()`, and `backpatch()` as explained in the Dragon-Book (as well as in the class). **[14]**

Production	Semantic action

L → S	`L.nextlist = S.nextlist`
	`backpatch( L.nextlist, nextinst )`

L → L_1 M S	`L.nextlist = S.nextlist`
	`backpatch( L₁.nextlist, M.inst )`

Production	Semantic action
$S \rightarrow A$	`S.nextlist = NULL`

Rewrite the following production by adding marker non-terminals at suitable places.
$S \rightarrow$ **if** (B) L E **endif**
 	
$S \rightarrow$ **if** (B) M_1 L N M_2 E **endif**
	`backpatch(B.truelist, M`$_1$`.inst)`
	`backpatch(B.falselist, M`$_2$`.inst)`
	`S.nextlist = merge(merge(L.nextlist, N.nextlist), E.nextlist)`

$E \rightarrow \varepsilon$	`E.nextlist = NULL`
$E \rightarrow$ **else** L	`E.nextlist = L.nextlist`

Rewrite the following production by adding marker non-terminals at suitable places.
$E \rightarrow$ **elif** (B) L E
 	
$E \rightarrow$ **elif** (B) M_1 L N M_2 E_1
	`backpatch(B.truelist, M`$_1$`.inst)`
	`backpatch(B.falselist, M`$_2$`.inst)`
	`E.nextlist = merge(merge(L.nextlist, N.nextlist), E`$_1$`.nextlist)`

$B \rightarrow X_1 R X_2$	`B.truelist = makelist(nextinst)`
	`B.falselist = makelist(nextinst + 1)`
	`gen(if X`$_1$`.addr R.op X`$_2$`.addr goto -)`
	`gen(goto -)`

$M \rightarrow \varepsilon$	`M.inst = nextinst`

$N \rightarrow \varepsilon$	`N.nextlist = makelist(nextinst)`
	`gen(goto -)`

4. [Basic blocks and flow graph]

The following C code computes the binomial coefficient $C(n, r)$ using the identity $C(i, j) = C(i-1, j) + C(i-1, j-1)$. Assume that $0 \leq r \leq n < 20$. The code uses a dynamic-programming approach to compute $C(n, r)$ in a two-dimensional array `C[20][20]` of integers. All basic variables are of type `int`.

```
C[0][0] = 1;
j = 1; while ( j <= r ) { C[0][j] = 0; j = j + 1; }
i = 1;
while ( i <= n ) {
    C[i][0] = 1; j = 1;
    while ( j <= r ) {
        if (j > i) C[i][j] = 0;
        else C[i][j] = C[i-1][j] + C[i-1][j-1];
        j = j + 1;
    }
    i = i + 1;
}
```

(a) Generate the 3-address code for the above snippet using fall-through optimization that reduces the number of `goto` statements (use `iffalse` in conditional jumps). Do not use any other optimization in this part. Assume that the size (width) of each `int` is 4 bytes. Use symbolic labels (`L1`, `L2`, `L3`, and so on) instead of instruction numbers. Name the temporaries as `t1`, `t2`, `t3`, and so on. Identify the basic blocks by enclosing them in rectangles. Name the basic blocks as `B1`, `B2`, `B3`, and so on. **[8]**

```
     +-------------------------+
     | t1 = 80 * 0             |     Block B1
     | t2 = 4 * 0              |
     | t3 = t1 + t2            |
     | C[t3] = 1               |
     | j = 1                   |
     +-------------------------+
L1:  | iffalse j <= r goto L2  |     Block B2
     +-------------------------+
     + t4 = 80 * 0             |     Block B3
     | t5 = 4 * j              |
     | t6 = t4 + t5            |
     | C[t6] = 0               |
     | t7 = j + 1              |
     | j = t7                  |
     | goto L1                 |
     +-------------------------+
L2:  | i = 1                   |     Block B4
     +-------------------------+
L3:  | iffalse i <= n goto L4  |     Block B5
     +-------------------------+
     | t8 = 80 * i             |     Block B6
     | t9 = 4 * 0              |
     | t10 = t8 + t9           |
     | C[t10] = 1              |
     | j = 1                   |
     +-------------------------+
L5:  | iffalse j <= r goto L6  |     Block B7
     +-------------------------+
     | iffalse j > i goto L7   |     Block B8
     +-------------------------+
     | t11 = 80 * i            |     Block B9
     | t12 = 4 * j             |
     | t13 = t11 + t12         |
     | t14 = i - 1             |
     | t15 = 80 * t14          |
     | t16 = 4 * j             |
     | t17 = t15 + t16         |
     | t18 = C[t17]            |
     | t19 = i - 1             |
     | t20 = 80 * t19          |
     | t21 = j - 1             |
     | t22 = 4 * t21           |
     | t23 = t20 + t22         |
     | t24 = C[t23]            |
     | t25 = t18 + t24         |
     | C[t13] = t25            |
     | goto L8                 |
     +-------------------------+
```

```
     +-------------------------+
L7:  | t26 = 80 * i            |     Block B10
     | t27 = 4 * j             |
     | t28 = t26 + t27         |
     | C[t28] = 0              |
     +-------------------------+
L8:  | t29 = j + 1             |     Block B11
     | j = t29                 |
     | goto L5                 |
     +-------------------------+
L6:  | t30 = i + 1             |     Block B12
     | i = t30                 |
     | goto L3                 |
     +-------------------------+
L4:
```

(b) Hand-optimize each basic block individually, by locating common subexpressions, by using constant propagation and algebraic identities (including strength reduction by applying distributivity), and by eliminating dead code. You do not have to use the DAG representation of the basic blocks. But give clear justifications (in plain English) for every optimization step that you use. Do not perform any global optimization here. Do not renumber the temporaries. **[8]**

The optimization steps are discussed block by block. The basic blocks that cannot be optimized are omitted in the discussion.

B_1: By algebraic identities, `t1`, `t2`, `t3` all evaluate to 0. We do not need to compute them, and instead straightaway set `C[0] = 1`.
Optimized block:

```
C[0] = 1
j = 1
```

B_3: `t4` evaluates to 0, and so `t6` = `t5`. So we do not need to compute `t4` and `t6`, and set `C[t5] = 0`.
Optimized block:

```
t5 = 4 * j
C[t5] = 0
t7 = j + 1
j = t7
goto L1
```

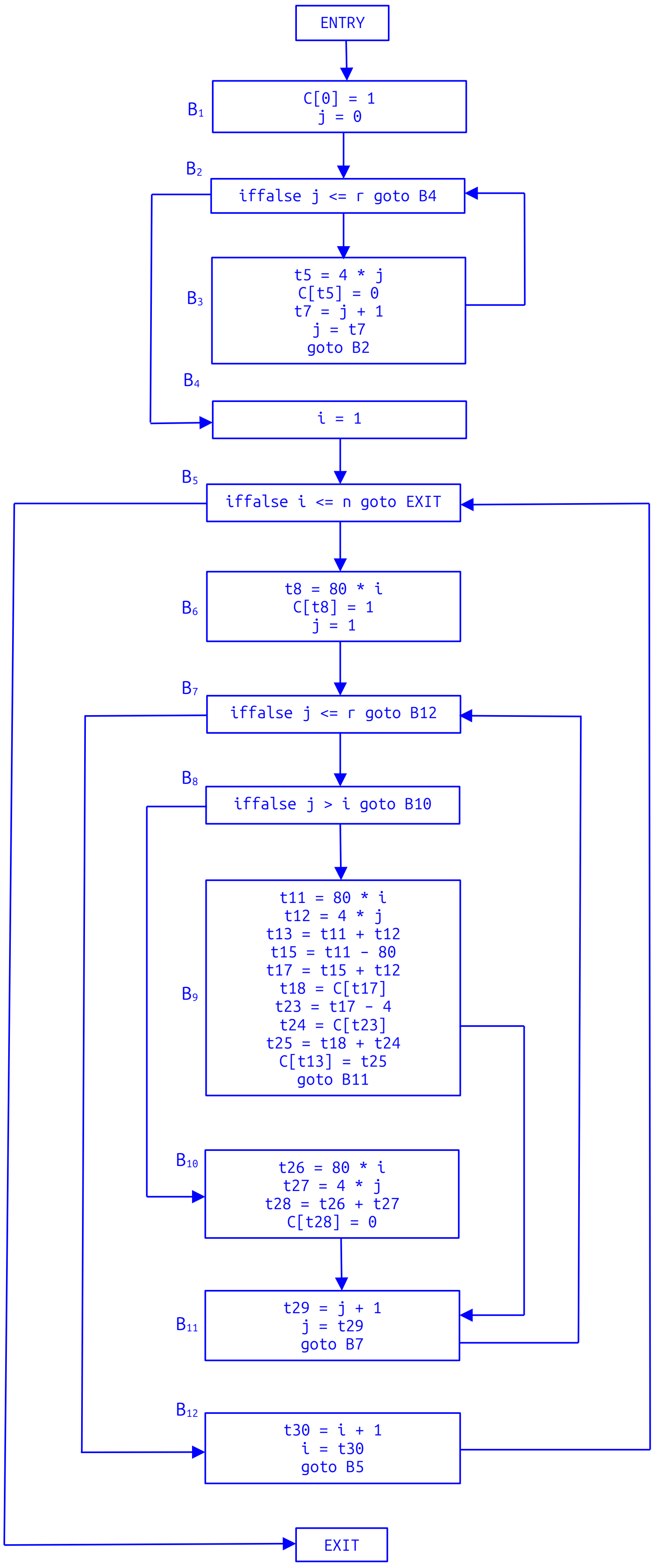
B_6: Again by using algebraic identities, we avoid computing `t9` and `t10`.
Optimized block:

```
t8 = 80 * i
C[t8] = 1
j = 1
```

B_9: This block offers several common subexpressions, and recalculation of formulas using the distributive law. `t15` can be computed as `t11 - 80`, so we can avoid computing `t14`. We can avoid computing `t16`, and use `t12` for it. Computations of `t19` and `t20` are redundant, we can use `t15` for `t20`. By distributive law, we can avoid computing `t21`, and take `t22 = t12 - 4`. We can do even better. We can avoid computing `t22` too, and take `t23 = t17 - 4`.
Optimized block:

```
t11 = 80 * i
t12 = 4 * j
t13 = t11 + t12
t15 = t11 - 80
t17 = t15 + t12
t18 = C[t17]
t23 = t17 - 4
t24 = C[t23]
t25 = t18 + t24
C[t13] = t25
goto L8
```

(c) Draw the flow graph on the optimized basic blocks. Write the 3-address instructions inside each optimized block. The instructions remaining in the optimized blocks should be in the same sequence as in the original blocks. Retain the old numbering of the temporaries (although some temporaries are not computed, and some use modified formulas). **[8]**



5. [Target-code generation]

Consider a single basic block consisting of the following seven 3-address instructions. Here, `a`, `b`, `c`, `d`, `e` are user-defined variables, and `t1`, `t2`, `t3` are compiler-generated temporaries.

```
t1 = a + d
t2 = b - c
t3 = t1 * t2
e = t3
a = t3 / t1
b = a + t3
c = t1 - c
```

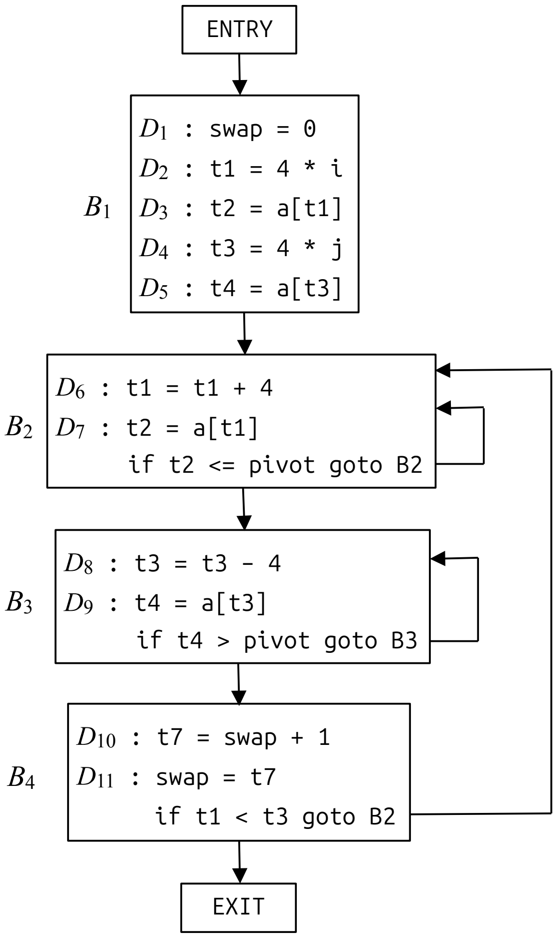
You have only four registers `R1`, `R2`, `R3`, `R4`. For every operation, the operands must stay in registers, and the result is computed in a register. For each 3-address instruction `I`, use the simple register-selection algorithm `getreg(I)` of the Dragon-Book (also taught in the class). At every step, show the register-descriptor table, the address-descriptor table, and the target code generated. Justify the necessity of load and store instructions, and the choice of each register by `getreg()`. For a 3-address instruction `x = y op z`, the registers are selected for `y`, `z`, and `x` in that order. Avoid storing temporaries in memory. Assume that the temporaries in a basic block are not used in other blocks (but user-defined variables are). All ties are broken by giving preference to lower-numbered registers. **[18]**

3-address code	Target code	Register descriptor				Address descriptor							
		R1	R2	R3	R4	a	b	c	d	e	t1	t2	t3
Initialize	No code generated					a	b	c	d	e			
		Justification: Initially, all the registers are empty, all variables are in memory, and no temporaries are computed.											
`t1 = a + d`	`LD R1,a` `LD R2,d` `ADD R3,R1,R2`	a	d	t1		a,R1	b	c	d,R2	e	R3		
		Justification: All registers are empty, so `a` is loaded to `R1`, `d` to `R2`, and `t1` is computed in `R3`.											
`t2 = b - c`	`LD R4,b` `LD R1,c` `SUB R2,R4,R1`	c	t2	t1	b	a	b,R4	c,R1	d	e	R3	R2	
		Justification: `R4` is empty, so `b` is loaded to `R4`. Now, all registers are loaded. However, `t1` will be used later in the block, whereas `a` is not live here and `d` will not be used later (and the latest values of `a` and `d` can be found in the memory), so `c` is loaded to `R1`, and `t2` is computed in `R2`.											
`t3 = t1 * t2`	`MUL R2,R3,R2`	c	t3	t1	b	a	b,R4	c,R1	d	e	R3		R2
		Justification: Since `t1` and `t2` are available in registers, they need not be loaded. In order to store the result `t3`, we note that `t1` and `c` will be used later, whereas `t2` and `b` will not be used later (and the latest value of `b` can be found in memory). Using our tie-breaking policy, we choose `R2` to store the `t3`.											

3-address code	Target code	Register descriptor				Address descriptor							
		R1	R2	R3	R4	a	b	c	d	e	t1	t2	t3
`e = t3`	No code generated	c	e,t3	t1	b	a	b,R4	c,R1	d	R2	R3		R2
		Justification: Since `t3` is available in `R2`, it is not reloaded. The copy is effected by letting `e` be added to the register descriptor for `R2`. Moreover, the address descriptor for `e` should store the information that the latest value of `e` can be found in `R2`. No code is generated.											
`a = t3 / t1`	`DIV R4,R2,R3`	c	e,t3	t1	a	R4	b	c,R1	d	R2	R3		R2
		Justification: The operands `t3` and `t1` are available in registers (no loads needed). For storing the result, note that `t3`, `t1`, and `c` are live at this point, whereas `b` is not. Moreover, the latest value of `b` can be found in memory, so we replace `b` in `R4` by the result. We also adjust the address descriptor for `a` to `R4`.											
`b = a + t3`	`ADD R1,R4,R2`	b	e,t3	t1	a	R4	R1	c	d	R2	R3		R2
		Justification: Again the operands `a` and `t3` are available in registers, and need not be loaded. In order to store the result (in the name of b), we note that R1 stores c. Although c is used later, its latest value can still be obtained from memory (so the score of R1 is 0). But the latest values of a, t1, and e are only in registers, so the score of (each of) R2, R3, and R4 is 1. So we choose R1 for b.											
`c = t1 - c`	`ST b,R1` `LD R1,c` `SUB R1,R3,R1`	c	e,t3	t1	a	R4	b	R1	d	R2	R3		R2
		Justification: Although `t1` is available in `R3`, `c` needs to be reloaded. All the registers are of score 1 (containing `b`, `e`, `t1`, and `a`, respectively). By our tie-breaking policy, we choose `R1` to load `c`. But before that, we need to store `b`. The result (which is `c` again) can be stored in `R1` itself.											
End of block	`ST a,R4` `ST c,R1` `ST e,R2`	c	e,t3	t1	a	a,R4	b	c,R1	d	e,R2	R3		R2
		Justification: We need to write back `a`, `c`, and `e` to memory.											

6. [Data flow analysis]

Consider the following (optimized) intermediate code which (only) finds the number of swaps required while partitioning a one-dimensional array `a[]` (with respect to a pivot element), during quick sort (swaps are not done). The basic blocks (denoted by B_i) and the data flow graph are shown below. The definitions are denoted by D_j.



(a) Write below the sets gen(B) for all basic blocks B. **[4]**

gen(B_1) = $\{ D_1, D_2, D_3, D_4, D_5 \}$

gen(B_2) = $\{ D_6, D_7 \}$

gen(B_3) = $\{ D_8, D_9 \}$

gen(B_4) = $\{ D_{10}, D_{11} \}$

(b) Write below the sets kill(B) for all basic blocks B. **[4]**

kill(B_1) = $\{ D_6, D_7, D_8, D_9, D_{11} \}$

kill(B_2) = $\{ D_2, D_3 \}$

kill(B_3) = $\{ D_4, D_5 \}$

kill(B_4) = $\{ D_1 \}$

(c) For a basic block B, present the generic transfer equations for computing in(B) and out(B). **[2]**

in(B) = Union of out(A) over all blocks A connected to the input of B

out(B) = gen(B) $\cup$ (in(B) – kill(B))

(d) In order to compute all reaching definitions (that is, in(B) and out(B) for all basic blocks B), we initialize out(B) = $\emptyset$ for all the basic blocks B (including `ENTRY` and `EXIT`). We then carry out a sequence of iterations for updating in(B) and out(B) for all basic blocks B. Show the iterations (use the format given below for the first iteration). Also write the reason/criterion for stopping the updating loop. Show your calculations. **[10]**

Iteration 1

in(B_1) = out(ENTRY) = Φ

in(B_2) = out(B_1) $\cup$ out(B_2) $\cup$ out(B_4) = Φ

in(B_3) = out(B_2) $\cup$ out(B_3) = Φ

in(B_4) = out(B_3) = Φ

out(B_1) = gen(B_1) $\cup$ (in(B_1) – kill(B_1)) = $\{ D_1, D_2, D_3, D_4, D_5 \}$

out(B_2) = gen(B_2) $\cup$ (in(B_2) – kill(B_2)) = $\{ D_6, D_7 \}$

out(B_3) = gen(B_3) $\cup$ (in(B_3) – kill(B_3)) = $\{ D_8, D_9 \}$

out(B_4) = gen(B_4) $\cup$ (in(B_4) – kill(B_4)) = $\{ D_{10}, D_{11} \}$

Show the remaining iterations.

Iteration 2

in(B_1) = out(ENTRY) = Φ
in(B_2) = out(B_1) $\cup$ out(B_2) $\cup$ out(B_4) = $\{ D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_{10}, D_{11} \}$
in(B_3) = out(B_2) $\cup$ out(B_3) = $\{ D_6, D_7, D_8, D_9 \}$
in(B_4) = out(B_3) = $\{ D_8, D_9 \}$
out(B_1) = gen(B_1) $\cup$ (in(B_1) – kill(B_1)) = $\{ D_1, D_2, D_3, D_4, D_5 \}$
out(B_2) = gen(B_2) $\cup$ (in(B_2) – kill(B_2)) = $\{ D_1, D_4, D_5, D_6, D_7, D_{10}, D_{11} \}$
out(B_3) = gen(B_3) $\cup$ (in(B_3) – kill(B_3)) = $\{ D_6, D_7, D_8, D_9 \}$
out(B_4) = gen(B_4) $\cup$ (in(B_4) – kill(B_4)) = $\{ D_8, D_9, D_{10}, D_{11} \}$

Continue the iterations on this page.

Iteration 3

$\text{in}(B_1) = \text{out(ENTRY)} = \Phi$

$\text{in}(B_2) = \text{out}(B_1) \cup \text{out}(B_2) \cup \text{out}(B_4) = \{ D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{in}(B_3) = \text{out}(B_2) \cup \text{out}(B_3) = \{ D_1, D_4, D_5, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{in}(B_4) = \text{out}(B_3) = \{ D_6, D_7, D_8, D_9 \}$

$\text{out}(B_1) = \text{gen}(B_1) \cup \big(\text{in}(B_1) - \text{kill}(B_1)\big) = \{ D_1, D_2, D_3, D_4, D_5 \}$

$\text{out}(B_2) = \text{gen}(B_2) \cup \big(\text{in}(B_2) - \text{kill}(B_2)\big) = \{ D_1, D_4, D_5, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{out}(B_3) = \text{gen}(B_3) \cup \big(\text{in}(B_3) - \text{kill}(B_3)\big) = \{ D_1, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{out}(B_4) = \text{gen}(B_4) \cup \big(\text{in}(B_4) - \text{kill}(B_4)\big) = \{ D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

Iteration 4

$\text{in}(B_1) = \text{out(ENTRY)} = \Phi$

$\text{in}(B_2) = \text{out}(B_1) \cup \text{out}(B_2) \cup \text{out}(B_4) = \{ D_1, D_2, D_3, D_4, D_5, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{in}(B_3) = \text{out}(B_2) \cup \text{out}(B_3) = \{ D_1, D_4, D_5, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{in}(B_4) = \text{out}(B_3) = \{ D_1, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{out}(B_1) = \text{gen}(B_1) \cup \big(\text{in}(B_1) - \text{kill}(B_1)\big) = \{ D_1, D_2, D_3, D_4, D_5 \}$

$\text{out}(B_2) = \text{gen}(B_2) \cup \big(\text{in}(B_2) - \text{kill}(B_2)\big) = \{ D_1, D_4, D_5, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{out}(B_3) = \text{gen}(B_3) \cup \big(\text{in}(B_3) - \text{kill}(B_3)\big) = \{ D_1, D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

$\text{out}(B_4) = \text{gen}(B_4) \cup \big(\text{in}(B_4) - \text{kill}(B_4)\big) = \{ D_6, D_7, D_8, D_9, D_{10}, D_{11} \}$

Write below the reason why the iterations stop.

Iteration 4 does not encounter a change in out(B) for any basic block B, so we have reached a fixed point (that is, running the iterations further will change neither in(B) nor out(B) for any basic block B). So the iterations stop at this point.

Rough Work

Rough Work

Rough Work

Rough Work