SPECIAL EDITION FOR CSEDU STUDENTS

TOUCH-N-PASS EXAM CRAM GUIDE SERIES

COMPILERS

Prepared By

Sharafat Ibn Mollah Mosharraf

CSE, DU

12th Batch (2005-2006)

Table of Contents

CHAPTER 4
SYNTAX ANALYSIS

Theories

4.1	**What are the error recovery strategies generally used by parser? [*2006. Marks: 2*]** 1. Panic-Mode Recovery 2. Phrase-Level Recovery 3. Error Productions 4. Global Correction
4.2	**What is ambiguous grammar? [*In-course 1, 2008-2009. Marks: 1*]** A grammar is ambiguous if it can have more than one parse tree generating a given string of terminals.
4.3	**What is meant by left recursion in a grammar? [*In-course 2005-2006. Marks: 1*]** A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α. For example: $A \rightarrow A\alpha \mid \beta$
4.4	**What is the problem with a production with an immediate left-recursion in a grammar? How can we eliminate left recursion? [*2005. Marks: 3*]** A production with an immediate left-recursion can cause recursive-descent and top-down predictive parsers to loop forever. **Removing immediate left recursion:** If $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \dots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$ Then we can rewrite the grammar as below: $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_n A'$ $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \in$
4.5	**Can we create LR parser table from an ambiguous grammar? When or why would one prefer to use ambiguous grammar? Justify your answer with example. [*2008, In-course 1, 2008-2009. Marks: 1 + 3*]**
4.6	**Give an unambiguous grammar that is not LR(1). [*2006. Marks: 3*]**
4.7	**Define LL(k) and LR(k) grammars. Write down the basic properties of LL(1) and LR(1) grammars. [*2007. Marks: 2 + 4*]**
4.8	**What is a synchronizing set? Write down the procedure for panic mode error recovery in LL parsing. [*2005. Marks: 4*]**
4.9	**What is the difference between LR(0) grammar and SLR grammar? [*2008. Marks: 2*]**
4.10	**What is $\in$-production? How $\in$-productions are handled in bottom-up parsing? Explain it from the grammar below: [*2007. Marks: 4*]** **S → aB** **B → ∈ | x**
4.11	**What are inadequate states? How do we try to resolve the problem associated with such states in SLR and LR(1) parsing? [*2005. Marks: 4*]**
4.12	**What do you understand by shift-reduce and reduce-reduce conflicts? Why these conflicts arise? Explain rules for resolving these conflicts in parsing table. [*2007. Marks: 6*]**
4.13	**Show that the construction of an LALR parser by merging LR(1) item sets with common core of a LR(1) grammar will not introduce any shift/reduce conflict. [*2004. Marks: 4*]**

4.14	**Write down the general structure of LR parsers. [*2003. Marks: 4*]**
4.15	**What does the characters L, A, L, R and 1 in the name LALR(1) parser stand for? [*In-course. Marks: 1*]**
4.16	**Which of the LL(1), SLR(1) and LR(1) can parse the following grammar? Why? [*In-course. Marks: 2.5*]** `S → A ∣ B` `A → b ∣ c` `B → c ∣ a`
4.17	**What does a lexical analyzer do when prefixes of the input string matches more than one patterns? [*2007, 2004. Marks: 2*]**

Exercises

4.1	**Consider the grammar:** `S → (L) ∣ a` `L → L,S ∣ S` **i. Find parse trees for the sentencs (a, (a,a)) and (a,((a,a),(a,a))).** **ii. Construct a leftmost derivation for each of the sentences in part (i).** **iii. Construct a rightmost derivation for each of the sentences in part (i). [*2003. Marks: 6*]**
4.2	**Eliminate ambiguities for the grammar: [*In-course 1, 2008-2009. Marks: 5*]** `E → E + E ∣ E * E ∣ (E) ∣ id`
4.3	**Eliminate left recursion from the following grammar:** `S → SX ∣ SSb ∣ XS ∣ a` `X → Xb ∣ Sa ∣ b` **When does the above algorithm of eliminating left recursion fail? [*2008. Marks: 3 + 2*]**
4.4	**Consider the grammar with the set of terminals:** `S → (L) ∣ a ∣ b` `L → L,S ∣ S` **(i) Remove left-recursion from the grammar and find the First and Follow sets for each non-terminal of the modified grammar.** **(ii) Write down a recursive descent parser (i.e. parsing algorithm) for the modified grammar. [*2007. Marks: 5 + 5*]**
4.5	**Consider the following grammar for arithmetic expressions.** `E → E + T ∣ E – T ∣ T` `T → T * F ∣ T / F ∣ F` `F → (E) ∣ id` **i. Write the above grammar eliminating immediate left recursion.** **ii. Draw the simplified transition diagram for the grammar. [*2005. Marks: 4*]**
4.6	**Consider the grammar with the set of terminals { (,), , , a, b}:** `S → (L) ∣ a ∣ b` `L → L,S ∣ S` **a. Remove left-recursion from the grammar and find the First(1) and Follow(1) sets for each non-terminal of the modified grammar.** **b. Construct the operator precedence relation table for the grammar.** **c. Find the right-most derivation in reverse for the string (a, (b, a)) and indicate the handles at each step.** **d. Draw the transition diagram of the grammar and simplify it (if possible). [*2004. Marks: 3***

	***+ 3 + 4 + 3 + 3*]** **e. Write down a recursive descent parser for the modified grammar. [*In-course 2005-2006. Marks: 5*]**
4.7	**Left factor the following grammar: [*2003. Marks: 3*]** `A → AabcA	Aad	AabA	Ad`
4.8	**Build an LL parsing table for the following grammar:** `S → (L)	a` `L → L,S	S` **Is this an LL(1) grammar? Justify your answer. [*2008. Marks: 5 + 2*]**
4.9	**Show that the following grammar is SLR but not LR(0). [*2008. Marks: 6*]** `S → A` `A → a A` `A → a`
4.10	**With LALR (lookahead LR) parsing, we can reduce the number of states in an LR(1) parser. Justify the above statement using the following grammar: [*2008. Marks: 8*]** `S → XX` `X → aX` `X → b`
4.11	**Consider the grammar: `S → aSbS	bSaS	∈`. Show that the grammar is ambiguous by constructing two different left most derivations for a string. [*2006. Marks: 3*]**
4.12	**Find the LL(1) parsing table for the following grammar. Show the first and follow sets for the non-terminals. [*In-course. Marks: 4 + 4*]** `S → aS	Ab` `A → XYZ	∈` `X → cS	∈` `Y → dS	∈` `Z → eS`
4.13	**Give a grammar with the following first and follow sets. Your grammar should have no epsilon productions. The non-terminals are X, Y, Z and the terminals are a, b, c, d, e. [*In-course. Marks: 6*]**

	X	Y	Z	a	b	c	d	e	F
First	**b, d, f**	**b, d**	**c, e**	**a**	**b**	**c**	**d**	**e**	**f**
Follow	**$**	**c, e**	**a**	**$**	**b, d**	**c, e**	**c, e**	**a**	**$**

4.14	**Consider the following grammar:** `X → YaYb	ZbZa` `Y → b` `Z → a	∈` **i. Give the parse tree for the string “babb”.** **ii. Show the sentential forms of each stage in a top-down leftmost derivation of “baa”.** **iii. Show the sentential forms of each stage in a bottom-up rightmost derivation of “babb”.** **iv. Work out the following sets: FIRST(X), FOLLOW(X), FOLLOW(Y). [*2006. Marks: 2 + 3 + 3 + 3*]**

4.15	`X → X + X	Y++` `Y → a` i. For the above grammar, construct the LR(1) parse table. ii. Point out where shift-reduce conflict(s) occurs. Give example(s) of a string(s) for which the parser will face the conflict(s)? iii. If the shift-reduce conflict(s) is resolved in favor of shift, what is the associativity of the '+' operator that corresponds to this choice? [*2006. Marks: 6 + 3 + 1*]
4.16	**For the following grammar, construct the DFA, recognizing viable prefixes, that includes just those states of an LR(1) parser that are pushed on the stack during a parse of the input: `(id||id&`. The set of terminals is { `|, &, (, ), id` } [*2005. Marks: 8*]** `Bexp → Bexp || Btrm	Bexp	Btrm |` `Bexp → Bexp	Btrm` `Bexp → Btrm` `Btrm → Btrm && Bfct	Btrm & Bfct	Bfct` `Bfct → (Bepr)	id`
4.17	**Consider the following grammar:** `S → aBX	Ay` `A → ab` `B → a	b` **Prove that this grammar is not LR(0), but is SLR(1).**
4.18	**Construct the LALR parse table for the following grammar that generates a subset of all possible regular expressions. Resolve shift/reduce or reduce/reduce conflicts (if occurs) using the usual precedence rules of the operators. The set of input symbols is {|, ., *, id} [*2004. Marks: 12*]** `R → R '	' R	R.R	R*	id`
4.19	**Construct the LALR parser table for the following grammar. Show all the necessary steps. [*2003. Marks: 12*]** `P → PaQ	Q` `Q → QR	R` `R → Rb	c	d`
4.20	**Construct the predictive parsing table for the following grammar: [*In-course. Marks: 6*]** `E → TA` `A → +TA	-TA	∈` `T → FB` `B → *FB	/FB	∈` `F → -S	S` `S → v	(E)`  **Show that the following grammar is not SLR(1): [*In-course. Marks: 4*]**  `S → Aa	bAc	dc	bda` `A → d`
4.21	**Construct SLR parsing table for the following grammar. [*In-course 1, 2008-2009. Marks: 10*]** `S → AS	b` `A → SA	a`
4.22	**The following grammar for if-then-else statements is proposed to remedy the dangling-else ambiguity:** **stmt → if expr then stmt | matched_stmt** **matched_stmt → if expr then matched_stmt else stmt | other** **Justify whether this grammar is ambiguous or not. If the grammar is still ambiguous,**

	rewrite the grammar by removing dangling-else ambiguity. [*2003. Marks: 6*]
4.23	**Construct the DFA, recognizing viable prefixes, that includes just those states of an LR(1) parser for the following grammar, that are pushed on the stack during a parse of the input y+++y. [*In-course. Marks: 6*]** `S → A` `A → A + A | B++` `B → y`
4.24	`E → E '|' T | T` `T → TF | F` `F → F* | F+ | (E) | a | b` **i. Modify the grammar for LL(1) parser.** **ii. Find First and Follow for each non-terminal of the modified grammar.** **iii. What are the items of the state associated with the viable prefix E | T (E in SLR parsing for the grammar in (i). Show why the items are valid for the state. [*In-course. Marks: 4 + 6 + 6 + 4*]**

CHAPTER 5
SYNTAX-DIRECTED TRANSLATIONS

Theories

5.1	**Define synthesized and inherited attributes. [*2003. Marks: 2*]** A *synthesized attribute* for a nonterminal A at a parse-tree node N is defined by a semantic rule associated with the production at N. Note that the production must have A as its head. A synthesized attribute at node N is defined only in terms of attribute values at the children of N and at N itself. An *inherited attribute* for a nonterminal B at a parse-tree node N is defined by a semantic rule associated with the production at the parent of N. Note that the production must have B as a symbol in its body. An inherited attribute at node N is defined only in terms of attribute values at N's parent, N itself, and N's siblings.
5.2	**What is L-attributed definition? "Every s-attributed definition is L-attributed" – Justify your answer. [*In-course 2, 2008-2009. Marks: 2 + 2*]** An SDD is called L-attributed definition if each attribute associated with the production bodies is either: 1. Synthesized, or 2. Inherited, but with the rules limited as follows. Suppose that there is a production $A \rightarrow X_1X_2 \cdots X_n$, and that there is an inherited attribute $X_i.a$ computed by a rule associated with this production. Then the rule may use only: (a) Inherited attributes associated with the head A. (b) Either inherited or synthesized attributes associated with the occurrences of symbols $X_1, X_2, \ldots, X_{i-1}$ located to the left of X_i. (c) Inherited or synthesized attributes associated with this occurrence of X_i itself, but only in such a way that there are no cycles in a dependency graph formed by the attributes of this X_i. An SDD is called s-attributed definition if every attribute is synthesized. On the other hand, L-attributed definition can have inherited attributes besides synthesized attributes. Hence, every s-attributed definition is also L-attributed.
5.3	**Explain how a translator for an s-attributed definition can be implemented as part of bottom-up parser. [*In-course 2, 2008-2009. Marks: 3.5*]** When an SDD is s-attributed, its attributes can be evaluated in any bottom-up order of the nodes of the parse tree. It is often simple to evaluate the attributes by performing a post-order traversal of the parse tree and evaluating the attributes at a node *N* when the traversal leaves *N* for the last time. That is, the function postoder() defined below is applied to the root of the parse tree. postorder(*N*) { for (each child *C* of *N*, from the left) postorder(*C*); evaluate the attributes associated with node *N*; } S-attributed definition can be implemented during bottom-up parsing, since a bottom-up parser corresponds to a post-order traversal. Post-order corresponds exactly to the order in which an LR-parser reduces a production body to its head.

5.4	**Write an algorithm for constructing a dependency graph from a given parse tree. [*In-course 2, 2008-2009. Marks: 3*]** Put each semantic rule in to the form b := f(c_1, c_2, …, c_k). **for** each node ***n*** in the parse tree **do** **for** each attribute ***a*** of the grammar symbol at ***n*** **do** construct a node in the dependency graph for ***a***; **for** each node ***n*** in the parse tree **do** **for** each semantic rule ***b* := *f* (c_1, c_2, … ,c_k)** associated with the production used at ***n*** do **for** *i* := 1 to *k* **do** construct an edge from the node for c_i to the node for ***b***;

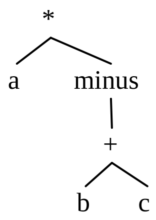
Exercises

5.1 **Translate the arithmetic expression a*-(b+c) into**

i. A syntax tree

ii. Postfix notation

iii. Three-address code [*2003. Marks: 4*]



Syntax Tree

abc+-*

Postfix Notation

```
t1 = b + c
t2 = minus t1
t3 = a * t2
```

3-Address Code

5.2 **Consider the following grammar for declaration of identifiers. [*2003. Marks: 4*]**

```
D → TL
T → integer | real
L → L, id | id
```

i. **Write down a syntax directed definition to propagate the type information using inherited and synthesized attributes.**

ii. **Rewrite the grammar so that the type information can be propagated using synthesized attributes only.**

	PRODUCTION	SEMANTIC RULES
1)	$D \to T\ L$	$L.inh = T.type$
2)	$T \to$ **int**	$T.type$ = integer
3)	$T \to$ **float**	$T.type$ = float
4)	$L \to L_1$, **id**	$L_1.inh = L.inh$ $addType(\mathbf{id}.entry, L.inh)$
5)	$L \to$ **id**	$addType(\mathbf{id}.entry, L.inh)$

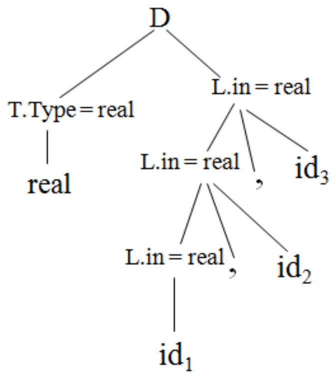
i.

5.3 **A syntax directed definition for declaring identifiers is**

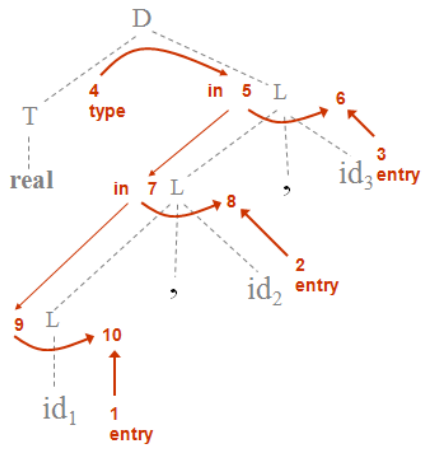
Productions	**Semantic Actions**
D → TL	**L.in := T.type**
T → int	**T.type := integer**
T → real	**T.type := real**
L → L1, id	**L1.in := L.in, addtype(id.entry, L.in)**
L → id	**addtype(id.entry, L.in)**

i. **Draw an annotated parse tree for the sentence real id_1, id_2, id_3**

ii. **Draw the dependency graph for the parse tree in (i). [*In-course 2, 2008-2009. Marks: 3 + 3*]**



Parse Tree



Dependency Graph

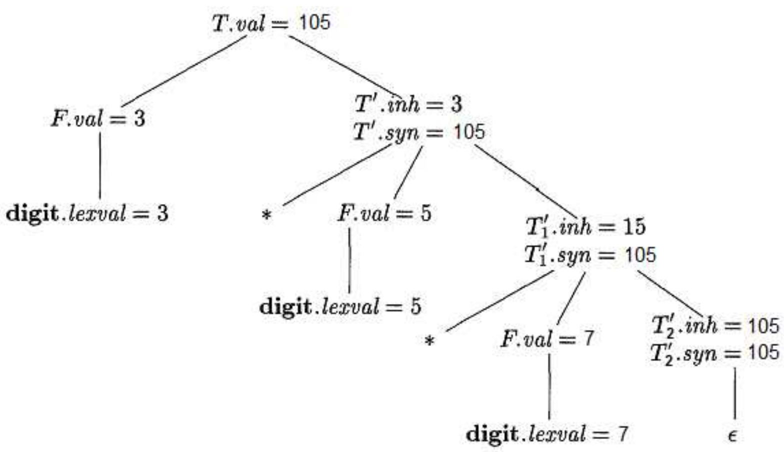
5.4 **Consider the following SDD:**

Productions	Rules
T* → *FT′	***T′.inh = F.val*** ***T.val = T′.syn***
***T′* → **FT*₁′**	***T₁′.inh = T′.inh × F.val*** ***T′.syn = T₁′.syn***
***T′* → ∈**	***T′.syn = T′.inh***
F* → *digit	***F.val = digit.lexval***

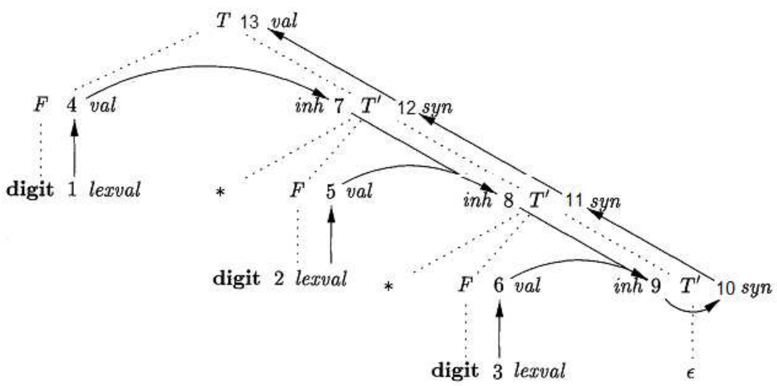
i. **Draw the annotated parse tree for the expression 3*5*7 using the semantic rules given above.**

ii. **Draw the dependency graph. [*2008. Marks: 2 + 1*]**

i.



ii.



CHAPTER 6
INTERMEDIATE-CODE GENERATION

Theories

6.1 **What are the advantages of using intermediate code generation? [*In-course 2, 2008-2009. Marks: 2.5*]**

OR, Why would we be interested to generate intermediate code as the final product of the front end of a compiler? [*2005. Marks: 2*]

With a suitably defined intermediate representation, a compiler for language *i* and machine *j* can then be built by combining the front end for language *i* with the back end for machine *j*. This approach to creating suite of compilers can save a considerable amount of effort: $m \times n$ compilers can be built by writing just *m* front ends and *n* back ends.

6.2 **Translate the expression `(a+b)*(c+d)+(b+c)` into**

- **i. Quadruples**
- **ii. Triples**
- **iii. Indirect triples [*In-course 2, 2008-2009. Marks: 3*]**

Discuss the comparative advantages and disadvantages of the above representations. [*2007. Maks: 2*]

ALSO, What are the advantages and disadvantages of using indirect triple representation of 3-address codes? [*2004. Marks: 2*]

```
t1 = a + b
t2 = c + d
t3 = t1 * t2
t4 = b + c
t5 = t3 + t4
```

3-Address Code

	op	*arg₁*	*arg₂*	*result*
0	+	a	b	t_1
1	+	c	d	t_2
2	*	t_1	t_2	t_3
3	+	b	c	t_4
4	+	t_3	t_4	t_5

Quadruples

	op	*arg₁*	*arg₂*
0	+	a	b
1	+	c	d
2	*	(0)	(1)
3	+	b	c
4	+	(2)	(3)

Triples

	instruction
35	(0)
36	(1)
37	(2)
38	(3)
39	(4)
	…

	op	*arg₁*	*arg₂*
0	+	a	b
1	+	c	d
2	*	(0)	(1)
3	+	b	c
4	+	(2)	(3)

Indirect Triples

Comparative Advantages and Disadvantages:

	Advantages	**Disadvantages**
Quadruples	Instructions can be moved around without requiring changing all references to it.	Takes more memory space than Triples.
Triples	Takes less memory space.	Moving an instruction may require changing all references to it.
Indirect Triples	Instructions can be moved by reordering the *instruction* list, without affecting the triples themselves. Thus, changes to all references to the moved instruction are not needed.	Takes more memory space than Triples.

6.3 **Find the expression that computes the relative address of the array element $A[i_1, i_2, \ldots, i_n]$, where A is a $10 \times 10 \times \ldots \times 10$ array stored in row-major form. [*2006. Marks: 2*]**

6.4 **What is backpatching? What is the advantage of backpatching? Explain with an example. [*In-course 2, 2008-2009. Marks: 2 + 4*]**

OR, Write short notes on backpatching. [*2007. Marks: 3*]

Backpatching is the activity of filling up unspecified information of labels using appropriate semantic actions during the code generation process.

In the code generation process, a separate pass is needed to bind jump labels to addresses.

	Backpatching merges the intermediate and target code generation into one pass.

Theories & Exercises (Type Checking)

6.1	**Define static and dynamic type checking. [*2006. Marks: 3*]**
6.2	**What are type constructors? Name and explain the basic type constructors. [*2003. Marks: 2 + 3*]**
6.3	**What is name and structural equivalence of type expressions? [*In-course 2, 2007. Marks: 1*]**
6.4	**Write type expression for the following type: An array of pointers to reals, where array index ranges from 1 to 100. [*2005. Marks: 2*]**
6.5	**Show whether the following recursive type expressions are equivalent or not: [*2005. Marks: 2*]**
6.6	**How do we determine the structural equivalence to types? [*In-course 2, 2007. Marks: 2*]** **ALSO, Give an algorithm for testing structural equivalence of type expressions and explain how it works? [*2003. Marks: 3*]**
6.7	**How is encoding of type expressions used for checking structural equivalence of type expressions and what are its advantages (if any)? [*2005, 2003; In-course 2, 2007. Marks: 3*]**
6.8	**Consider the following grammar in a programming language. Here, P, D, T, S and E represent the program, the declarations, types, statements and expressions respectively.** `P → D; S` `D → D; D │ id : T` `T → char │ int │ array[num] of T` `S → id = E	if E then S	S; S` `E → literal │ num │ id │ E[E]` **Write down the syntax-directed translation for type checking and determine each declaration's offset in the activation record. Define the basic types, type constructs and functions you use. [*2004. Marks: 7*]**
6.9	**Let the following attribute grammar is used for type checking. [*2008. Marks: 5*]** `E → num        { E.type := integer; }` `E → id         { E.type := lookup(id.name); }` `E → E1 + E2    { if (E1.type = integer & E2.type = integer)` `                 then E.type := integer;` `                 else type_error();   }` `E → E1 [E2]    { if (E2.type = integer & E1.type = array of T)` `                 then E.type := T;` `                 else type_error(); }` `E → E1 (E2)    { if (E1.type = T1 → T2 & E2.type = T1)` `                 then E.type := T2;` `                 else type_error(); }` `E → E1^         { if (E1.type = ^T) then E.type := T` `                 else type_error(); }` **Now consider the following declarations:** `I: integer;` `A: array[20] of integer;` `B: array[20] of ^integer;`

```
F: ^integer → integer;
```

Show how type checking is performed for the following expressions using the grammar

(i) `I := B[F(A[3])]`
(ii) `I := A[F(B[3])]`

Exercises (Others)

6.1 **Write an SDD to generate 3-address code for the following grammar:**

```
P → S
S → assign | if (B) S1 | S1 S2
B → B1 || B2 | B1 && B2 | id1 rel id2 | true | false
```

According to your SDD what code will be generated for the following expression?

`if (x < 100 || x > 200 && x != y) x = 0` **[*2008. Marks: 6 + 2*]**

PRODUCTION	SEMANTIC RULES
$P \rightarrow S$	$S.next = newlabel()$ $P.code = S.code \;\|\|\; label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** (B) S_1	$B.true = newlabel()$ $B.false = S_1.next = S.next$ $S.code = B.code \;\|\|\; label(B.true) \;\|\|\; S_1.code$
$S \rightarrow$ **if** (B) S_1 **else** S_2	$B.true = newlabel()$ $B.false = newlabel()$ $S_1.next = S_2.next = S.next$ $S.code = B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen('\texttt{goto}' \; S.next)$ $\|\| \; label(B.false) \;\|\|\; S_2.code$
$S \rightarrow$ **while** (B) S_1	$begin = newlabel()$ $B.true = newlabel()$ $B.false = S.next$ $S_1.next = begin$ $S.code = label(begin) \;\|\|\; B.code$ $\|\| \; label(B.true) \;\|\|\; S_1.code$ $\|\| \; gen('\texttt{goto}' \; begin)$
$S \rightarrow S_1 \; S_2$	$S_1.next = newlabel()$ $S_2.next = S.next$ $S.code = S_1.code \;\|\|\; label(S_1.next) \;\|\|\; S_2.code$

Production	Semantic Rules
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true$ $B_1.false = newlabel()$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \text{ \&\& } B_2$	$B_1.true = newlabel()$ $B_1.false = B.false$ $B_2.true = B.true$ $B_2.false = B.false$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow\ !\ B_1$	$B_1.true = B.false$ $B_1.false = B.true$ $B.code = B_1.code$
$B \rightarrow (B_1)$	$B_1.true = B.true$ $B_1.false = B.false$ $B.code = B_1.code$
$B \rightarrow E_1 \textbf{ rel } E_2$	$B.code = E_1.code \mid\mid E_2.code$ $\mid\mid gen('\texttt{if}'\ E_1.addr\ \textbf{rel}.op\ E_2.addr\ '\texttt{goto}'\ B.true)$ $\mid\mid gen('\texttt{goto}'\ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen('\texttt{goto}'\ B.true)$
$B \rightarrow \textbf{false}$	$B.code = gen('\texttt{goto}'\ B.false)$

Code for the expression **`if (x < 100 || x > 200 && x != y) x = 0`**:

```
       if x < 100 goto L2
       goto L3
L3:    if x > 200 goto L4
       goto L1
L4:    if x != y goto L2
       goto L1
L2:    x = 0
L1:
```

6.2 **Write a syntax-directed definition that generates three-address code for Boolean expression of the following grammar:**

E → E1 or E2 | E1 and E2 | not E1 | (E1) | id1 relop id2 | true | false

Using your definition, generate code for the expression

`m < n or p < q and s < t` **[*In-course 2, 2008-2009. Marks: 6 + 3.5*]**

[*This grammar is a subset of the grammar from the above question (6.1). Just replace B with E, || with or, && with and, ! with not, rel with relop in the following SDD.*]

Code for the expression **`m < n or p < q and s < t`**:

```
        if m < n goto L1
        goto L2
L2:     if p < q goto L3
        goto L4
L3:     if s < t goto L5
        goto L6
```

6.3 **Write down a SDT scheme with backpatching to generate 3-address code for the following grammar:**

S → if E then S | if E then S else S | while E do S | do S while E | {L} | A
E → E and E | E or E | not E | (E) | id relop id | true | false

```
L → L; S | S
```

Construct the annotated parse tree for the code segment below:

```
while (i > j) and (k < m ) do
  if a > b then s = s + 1 else s = s – 1
```

Assume the arithmetic statements “s = s + 1” and “s = s – 1” above are generated from the non-terminal A in the above grammar. [*2003. Marks: 8 + 4*]

[*The grammars are similar to the following grammars:*]

1) $S \rightarrow \textbf{if}\,(\,B\,)\;M\;S_1$ $\{\; backpatch(B.truelist,\;M.instr);$
$S.nextlist \;=\; merge(B.falselist,\;S_1.nextlist);\;\}$

2) $S \rightarrow\;\; \textbf{if}\,(\,B\,)\;M_1\;S_1\;N\;\textbf{else}\;M_2\;S_2$
$\{\; backpatch(B.truelist,\;M_1.instr);$
$backpatch(B.falselist,\;M_2.instr);$
$temp \;=\; merge(S_1.nextlist,\;N.nextlist);$
$S.nextlist \;=\; merge(temp,\;S_2.nextlist);\;\}$

3) $S \rightarrow\;\; \textbf{while}\;M_1\;(\,B\,)\;M_2\;S_1$
$\{\; backpatch(S_1.nextlist,\;M_1.instr);$
$backpatch(B.truelist,\;M_2.instr);$
$S.nextlist \;=\; B.falselist;$
$emit(\text{'goto'}\;M_1.instr);\;\}$

4) $S \rightarrow \{\;L\;\}$ $\{\; S.nextlist \;=\; L.nextlist;\;\}$

5) $S \rightarrow A\;;$ $\{\; S.nextlist \;=\; \textbf{null};\;\}$

6) $M \rightarrow \epsilon$ $\{\; M.instr \;=\; nextinstr;\;\}$

7) $N \rightarrow \epsilon$ $\{\; N.nextlist \;=\; makelist(nextinstr);$
$emit(\text{'goto \_'});\;\}$

8) $L \rightarrow L_1\;M\;S$ $\{\; backpatch(L_1.nextlist,\;M.instr);$
$L.nextlist \;=\; S.nextlist;\;\}$

9) $L \rightarrow S$ $\{\; L.nextlist \;=\; S.nextlist;\;\}$

1) $B \rightarrow B_1\;||\;M\;B_2$ $\{\; backpatch(B_1.falselist, M.instr);$
$B.truelist = merge(B_1.truelist, B_2.truelist);$
$B.falselist = B_2.falselist;\;\}$

2) $B \rightarrow B_1\;\&\&\;M\;B_2$ $\{\; backpatch(B_1.truelist, M.instr);$
$B.truelist = B_2.truelist;$
$B.falselist = merge(B_1.falselist, B_2.falselist);\;\}$

3) $B \rightarrow\;!\;B_1$ $\{\; B.truelist = B_1.falselist;$
$B.falselist = B_1.truelist;\;\}$

4) $B \rightarrow (\;B_1\;)$ $\{\; B.truelist = B_1.truelist;$
$B.falselist \;=\; B_1.falselist;\;\}$

5) $B \rightarrow E_1\;\textbf{rel}\;E_2$ $\{\; B.truelist = makelist(nextinstr);$
$B.falselist = makelist(nextinstr + 1);$
$emit(\text{'if'}\;E_1.addr\;\textbf{rel}.op\;E_2.addr\;\text{'goto \_'});$
$emit(\text{'goto \_'});\;\}$

6) $B \rightarrow \textbf{true}$ $\{\; B.truelist = makelist(nextinstr);$
$emit(\text{'goto \_'});\;\}$

7) $B \rightarrow \textbf{false}$ $\{\; B.falselist = makelist(nextinstr);$
$emit(\text{'goto \_'});\;\}$

8) $M \rightarrow \epsilon$ $\{\; M.instr = nextinstr;\;\}$

6.4 **The following grammar generates the assignment statements of single dimensional array and identifiers**

	`S → E = E` `E → E + E | id | id[E]` **i. Write a syntax-directed translation scheme for the grammar to produce 3-address codes.** **ii. Draw the annotated parse tree for the statement: `x[i] = x[x[i]] + i`, according to your translation scheme. Assume the array `x[]` is a 10 element array where each element occupies 2 bytes. [*2007. Marks: 5 + 5*]**
6.5	**Consider the following grammar: [*2007, 2005. Marks: 8 + 6*]** `S → if E then S | for id = NUM to NUM Step by NUM S| {L} | A` `E → E or E | id relop id | id relop NUM` `L → L; S | S` **i. Write down a syntax-directed translation scheme with backpatching to generate 3-address codes for the grammar. Assume *A* generates assignment statements (e.g. a = a + 1) that are represented by a single quad. The semantics of "for": id will be initialized to the first NUM, incremented in each iteration by the amount specified by the third NUM up to the second NUM. The token NUM is a signed integer.** **ii. According to your translation scheme for the grammar in question (i), construct the annotated parse tree for the following code fragment. Assume the 3-address codes start from quad 100.** `for a = 1 to 10 step by 2 {` `  if b > 10 or a < c then b = b – 1;` `  c = c – b` `}` `a = a + 1`
6.6	**Consider the following grammar: [*2004. Marks: 8 + 6*]** `S → if E then S else S | for (A; E; A) S | {L} | A` `E → E and E | E or E | id relop id | id relop NUM` `L → L; S | S` **i. Write down a syntax-directed translation scheme with backpatching to generate 3-address codes for the grammar. Assume *A* generates assignment statements (e.g. a = a + 1) that are represented by a single quad and "for" has the same semantics of C's *for* loop.** **ii. According to your translation scheme for the grammar in question (i), construct the annotated parse tree for the following code fragment. Assume the priority of 'and' is greater than that of 'or' and the 3-address codes start from quad 100.** `if (b > 10 or a < d and a > c) then b = b – 1 else d = d + 1` `a = a + 1` **iii. According to your translation scheme for the grammar in question (i), construct the annotated parse tree for the following code fragment. Assume the priority of 'and' is greater than that of 'or' and the 3-address codes start from quad 20. [*In-course 2, 2007. Marks: 5*]** `a = a + 1;` `for (b = 0; b > a; b = b + 1) {` `  if b < c and b > 10 then b = c – 1` `  else b = c - 2` `}` `c = 2;`
6.7	**Consider the following grammar: [*2006. Marks: 8 + 6*]** `S → if E then S | repeat S until E| {L} | A` `E → E or E | id relop id`

	`L → L; S | S` i. **Write down a syntax-directed translation scheme with backpatching to generate 3-address codes for the grammar. Assume *A* generates assignment statements (e.g. a = a + 1) that are represented by a single quad and relop represents any relational operator.** ii. **According to your translation scheme for the grammar in question (i), construct the annotated parse tree for the following code fragment. Assume the 3-address codes start from quad 10.** `repeat {` `  a = a + 1;` `  if b < 10 or a > b then b = b + 1` `} until (a > c)`
6.8	**Consider the following grammar: [*2006. Marks: 8 + 6*]** `S → if E then S | repeat S until E| {L} | A` `E → E and E | id relop id | (E)` `L → L; S | S` i. **Add production to generate "break" and "continue" statements. Write down a syntax-directed translation scheme with backpatching to generate 3-address codes for the modified grammar. Assume:** a. ***A* generates assignment statements (e.g. a = a + 1) that are represented by a single quad.** b. **"break" and "continue" statements have the semantics that those statements have in C language.** ii. **According to your translation scheme for the grammar in question (i), construct the annotated parse tree for the following code fragment. Assume the 3-address codes start from quad 100.** `repeat {` `  if (a > b and a < c) then break;` `  a = a – b` `} until (a > d)` `a = a + 1`
6.9	**The following grammar generates expressions formed by applying an arithmetic operator + to integer and real constants. When two integers are added, the resulting type is integer, otherwise, it is real:** `E → E + T | T` `T → num . num | num` **Give an SDD to determine the type of each sub-expression. [*2006. Marks: 4*]**

CHAPTER 8
CODE GENERATION

Theories

8.1 **Briefly describe the issues in the design of a code generator. [*In-course 3, 2008-2009. Marks: 4*]**

Issues in the design of a code generator:

1. Input to the Code Generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as bytecodes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's.

2. The Target Program

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

3. Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement-by-statement code generation, however, often produces poor code that needs further optimization. If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.

4. Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

	5. Evaluation Order The order in which computations are performed can affect the efficiency of the target code. some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem.
8.2	**Write short notes on the following: [*2003. Marks: 2 × 4*]** **i. Basic blocks** **ii. Peephole optimization** i. Basic blocks are maximal sequences of consecutive three-address instructions with the properties that a. The flow of control can only enter the basic block through the first instruction in the block. That is, there are no jumps into the middle of the block. b. Control will leave the block without halting or branching, except possibly at the last instruction in the block. ii. While most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program. A simple but effective technique for locally improving the target code is *peephole optimization*, which is done by examining a sliding window of target instructions (called the *peephole*) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible. Following are examples of program transformations that are characteristic of peephole optimizations: • Redundant-instruction elimination • Flow-of-control optimizations • Algebraic simplifications • Use of machine idioms
8.3	**With the help of an example describe the "next-use" algorithm. [*In-course 3, 2008-2009. Marks: 4.5*]** • Input: A basic block B of three-address statement. Initially the symbol table shows all nontemporary variables in B as being live on exit. • Output: At teach statement i: $x = y + z$ in B, attach to I the liveness and next-use information of x, y, z. • Method: Start at the last statement in B and scan backwards. At each statement i: $x = y + z$ in B, do – Attach to statement i the information currently found in the symbol table regarding the next use and liveness of x, y, z. – In the symbol table, set x to "not live" and "no next use" (i.e., "dead") – In the symbol table, set y and z to "live" and the next uses of y and z to i. **Example:**

```
 1:  t1 := 4 * i          <---- t1:L(2)    i:L(3)
 2:  t2 := a[t1]          <---- t2:L(5)    a:L(0)     t1:D
 3:  t3 := 4 * i          <---- t3:L(4)    i:L(8)
 4:  t4 := b[t3]          <---- t4:L(5)    b:L(0)     t3:D
 5:  t5 := t2 * t4        <---- t5:L(6)    t2:D       t4:D
 6:  t6 := prod + t5      <---- t6:L(7)    prod:D     t5:D
 7:  prod := t6           <---- prod:L(0)  t6:D
 8:  t7 := i + 1          <---- t7:L(9)    i:D
 9:  i := t7              <---- i:L(10)    t7:D
10:  if i <= 20 goto ...  <---- i:L(0)
```

Table:

	Initial	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10
t1	D	D	D	D	D	D	D	D	D	L(2)	D
t2	D	D	D	D	D	D	L(5)	L(5)	L(5)	D	D
t3	D	D	D	D	D	D	D	L(4)	D	D	D
t4	D	D	D	D	D	D	L(5)	D	D	D	D
t5	D	D	D	D	D	L(6)	D	D	D	D	D
t6	D	D	D	D	L(7)	D	D	D	D	D	D
t7	D	D	L(9)	D	D	D	D	D	D	D	D
a	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(2)	L(2)
b	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(0)	L(4)	L(4)	L(4)	L(4)
prod	L(0)	L(0)	L(0)	L(0)	D	L(6)	L(6)	L(6)	L(6)	L(6)	L(6)
i	L(0)	L(10)	D	L(8)	L(8)	L(8)	L(8)	L(8)	L(3)	L(3)	L(1)

8.4 **Write down the heuristic for graph coloring. What options do we have when an interference graph is found not to be k-colorable when there are k registers in the target machine? How do we determine the cost of fixing that problem? [*In-course. Marks: 3 + 3*]**

ALSO, Describe the heuristic used to color an interference graph. [*In-course. Marks: 2.5*]

Although the problem of determining whether a graph is k-colorable is NP-complete in general, the following heuristic technique can usually be used to do the coloring quickly in practice. Suppose a node n in a graph G has fewer than k neighbors (nodes connected to n by an edge). Remove n and its edges from G to obtain a graph G'. A k-coloring of G' can be extended to a k-coloring of G by assigning n a color not assigned to any of its neighbors.

By repeatedly eliminating nodes having fewer than k edges from the register-interference graph, either we obtain the empty graph, in which case we can produce a k-coloring for the original graph by coloring the nodes in the reverse order in which they were removed, or we obtain a graph in which each node has k or more adjacent nodes. In the latter case a k-coloring is no longer possible. At this point a node is spilled by introducing code to store and reload the register. Chaitin has devised several heuristics for choosing the node to spill. A general rule is to avoid introducing spill code into inner loops.

Exercises

8.1 **Describe the code generation algorithm for a 2-address machine. Generate the code for the following code segment according to the algorithm. [*In-course. Marks: 6 + 4*]**

```
T1 := a + c
T2 := b * T1
T3 := T1 - T2
T4 := T3
a  := T3 * T4
```

8.2	Consider a hypothetical machine with four registers R1, R2, R3, R4 and six addressing modes with the following costs.

Addressing Mode	Cost
Absolute Memory Address	1
Register	0
Literal	0
Indirect Register	1
Indirect Plus Address	1
Double Indirect	2

Now use an efficient algorithm to generate code for the target machine from the following block of 3-address codes:

```
t1 := a + b
t2 := t1 * c
t3 := t2 – t1
b  := t3
```

Calculate the cost of generated code and compare with cost of code generated with naïve approach to code generation. [*2008. Marks: 8*]

8.3 Draw the flow graph for the following program: [*In-course 3, 2008-2009. Marks: 4*]

```
begin
  prod := 0;
  i := 1;
  do begin
       prod := prod + a[i] * b[i];
       i := i + 1;
  end
  while (i <= 20);
end
```

8.4 Draw the flow graph for the following sequence of 3-address codes. [*2008. Marks: 3*]

```
(1)   i = 0
(2)   t1 = 10
(3)   t2 = i < t1
(4)   if False t2 goto (15)
(5)   t3 = 4
(6)   t4 = t3 * i
(7)   t5 = a + t4
(8)   if t5 >= 100 goto (15)
(9)   if t4 <= 20 goto (5)
(10)  t6 = z[t5]
(11)  *(t7) = t6
(12)  t8 = 1
(13)  i = i + t9
(14)  goto (2)
(15)  return
```

8.5 For the following code fragment, determine the next-use information (assume that all the variables are live and all temporaries are dead at the end of the block). [*2008. Marks: 5*]

```
(1)   t6 := 4 * i
(2)   x := a[t6]
(3)   t7 := 4 * i
(4)   t8 := 4 * j
(5)   t9 := a[t8]
(6)   a[t7] := t9
(7)   t10 := 4 * j
(8)   b[t10] := x
(9)   goto ....
```

8.6 Consider the following block of 3-address code:

```
t1 := z * x
t2 := z + t1
y  := t2 * z
z  := x + y
t1 := z * x
y  := x / t1
```

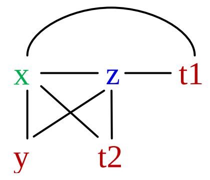
Use the graph coloring algorithm for register allocation for the block of code given above.

Assume the number of registers is R = 3 and all temporaries are dead at the end of the block. [*2008. Marks: 6*]

Liveness Information:

```
t1 := z * x   {x, z}
t2 := z + t1  {x, z, t1}
y  := t2 * z  {x, z, t2}
z  := x + y   {x, y}
t1 := z * x   {x, z}
y  := x / t1  {x, z, t1}
              {x, y, z}
```

Interference Graph with Coloring:



Stack: t1, z, t2, x, y

CHAPTER 9
MACHINE-INDEPENDENT OPTIMIZATIONS

Theories

9.1 **What do you understand by peephole optimizations? [*In-course 3, 2008-2009. Marks: 4*]**

While most production compilers produce good code through careful instruction selection and register allocation, a few use an alternative strategy: they generate naive code and then improve the quality of the target code by applying "optimizing" transformations to the target program.

A simple but effective technique for locally improving the target code is *peephole optimization*, which is done by examining a sliding window of target instructions (called the *peephole*) and replacing instruction sequences within the peephole by a shorter or faster sequence, whenever possible.

Following are examples of program transformations that are characteristic of peephole optimizations:

- Redundant-instruction elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms

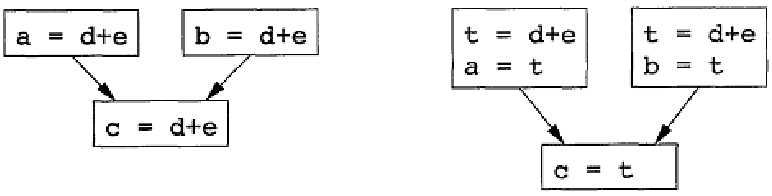
9.2 **What is a copy statement? When can we eliminate copy statements? Give an example. [*2007, 2004. Marks: 4*]**

Assignments of the form u = v are called *copy statements*.

We can eliminate copy statements when there are common sub-expressions in statements.

Example:

In order to eliminate the common subexpression from the statement `c = d+e` in Fig. 9.6(a), we must use a new variable t to hold the value of $d+e$. The value of variable t, instead of that of the expression $d+e$, is assigned to c in Fig. 9.6(b). Since control may reach `c = d+e` either after the assignment to a or after the assignment to b, it would be incorrect to replace `c = d+e` by either `c = a` or by `c = b`.



(a) (b)

Figure 9.6: Copies introduced during common subexpression elimination

9.3 **What is meant by dead code? Give examples. [*2007. Marks: 2*]**

Dead code are statements that compute values that never get used.

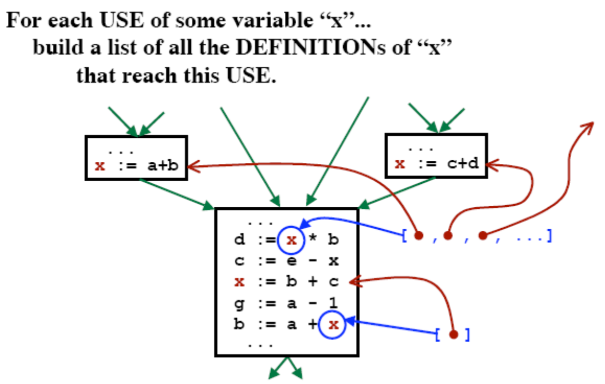
For example, suppose a variable *debug* is set to FALSE at various points in the program, and used in statements like *if (debug) print…* If copy propagation replaces *debug* by FALSE, then the print statement is dead because it cannot be reached. Hence, both the test and the print operation can be eliminated from the object code.

9.4 **In order to generate optimized code, you have the options to do dead-code elimination, copy propagation, CSE (Common Sub-expression Elimination), global register allocation and instruction scheduling. In which order would you perform these operations? Justify your choice with proper reasoning. [*2006. Marks: 5*]**

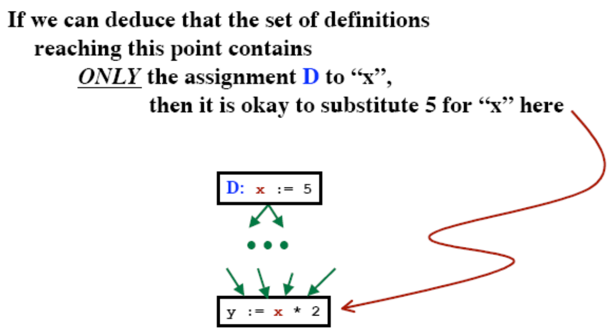
1. First of all, copy propagation should be performed. It would increase the possibility of finding common sub-expressions.
2. Then, CSE should be performed.
3. After these two steps, there would be a good possibility of existence of dead-code. So, at this stage, dead-code elimination should be performed.
4. Then, instruction scheduling might increase the chance of a more efficient register allocation. So, it should be performed.
5. Finally, Register allocation should be applied.

9.5 **Define UD- and DU-chains. What purposes do they serve? [*2007, 2004. Marks: 4*]**

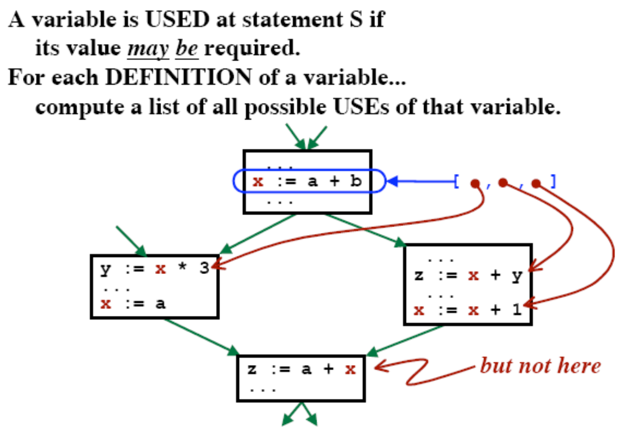
UD-Chain:



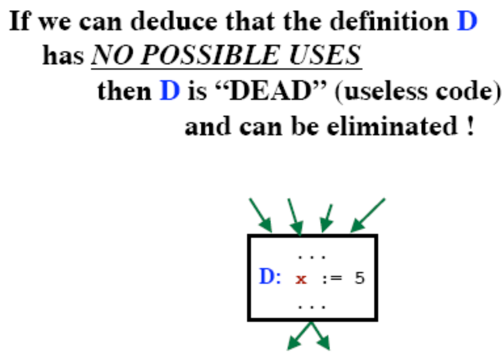
Purpose of UD-Chain:



DU-Chain:



Purpose of DU-Chain:



9.6 **What are the techniques to optimize a loop? Describe any one of them. [*In-course 3, 2008-2009. Marks: 3.5*]**

ALSO, What is the basic loop optimization technique? [*2004. Marks: 1*]

The techniques to optimize a loop are as follows:

1. Code Motion
2. Induction-Variable Elimination
3. Reduction in Strength

The basic loop optimization technique is *code motion*. If an expression is computed within a loop and it does not depend on variables that change in the loop, then it can be moved to just before the loop.

Example:

```
while i <= MAX-1 do
        ...
        j := i * (MIN+1);
        ...
end;
```

These computations are "loop-invariant"

(a) Before Code Motion

```
t1 := MAX-1;
t2 := MIN-1;
while i <= t1 do
        ...
        j := i * t2;
        ...
end;
```

(b) After Code Motion

9.7 **Why would you be concerned to find whether a flow graph is reducible or not? [*2007. Marks: 2*]**

OR, For loop optimization, why are we interested in determining whether a flow graph is reducible or not? [*2004. Marks: 2*]

A *back edge* is an edge $a \to b$ whose head b dominates its tail a. For any flow graph, every back edge is retreating, but not every retreating edge is a back edge. A flow graph is said to be *reducible* if all its retreating edges in any depth-first spanning tree are also back edges. In other words, if a graph is reducible, then all the DFST's have the same set of retreating edges, and those are exactly the back edges in the graph. If the graph is *nonreducible* (not reducible), however, all the back edges are retreating edges in any DFST, but each DFST may have additional retreating edges that are not back edges. These retreating edges may be different from one DFST to another. Thus, if we remove all the back edges of a flow graph and the remaining graph is cyclic, then the graph is nonreducible, and conversely.

9.8 **Write down an algorithm for detecting loop invariant computations. [*2007, 2005. Marks: 6*]**

Exercises

9.1 **Consider the following fragment of intermediate code:**

```
y = w
z = 4
v = y * y
u = z + 2
r = w ** 2 //this is exponentiation
t = r * v
s = u * t
```

Assume the only variable live at the exit is *s*. Show the result of applying constant propagation, algebraic simplification, common sub-expression elimination, constant folding, copy propagation and dead code elimination as much as possible to this code. You should explain the changes in each step. [*2007, 2005. Marks: 6*]

```
y = w
z = 4
v = y * y
  = w * w     [Copy Propagation]
u = z + 2
  = 4 + 2     [Constant Propagation]
  = 6         [Constant Folding]
r = w ** 2
  = w * w     [Algebraic Simplification]
  = v         [Common Sub-expression Elimination]
```

```
t = r * v
  = v * v    [Copy Propagation]
s = u * t
  = 6 * t    [Constant Propagation]
```

After dead-code elimination:

```
v = w * w
t = v * v
s = 6 * t
```

9.2 **Consider the following fragment of intermediate code:**

```
w = 2
u = z
y = w + 1
v = y * y
r = v ** 2 //this is exponentiation
t = u * u
s = u * t
x = y * y
```

Assume the only variables live at the exit are *s*, *x*. Show the result of applying constant propagation, algebraic simplification, common sub-expression elimination, constant folding, copy propagation and dead code elimination as much as possible to this code. You should explain the changes in each step. [*In-course. Marks: 4*]

```
w = 2
u = z
y = w + 1
  = 2 + 1    [Constant Propagation]
  = 3        [Constant Folding]
v = y * y
 = 3 * 3     [Constant Propagation]
 = 9         [Constant Folding]
r = v ** 2
  = v * v    [Algebraic Simplification]
  = 9 * 9    [Constant Propagation]
  = 81       [Constant Folding]
t = u * u
  = z * z    [Copy Propagation]
s = u * t
  = z * t    [Copy Propagation]
x = y * y
 = 3 * 3     [Constant Propagation]
 = 9         [Constant Folding]
```

After dead-code elimination:

```
t = z * z
s = z * t
x = 9
```

9.3 **For the following code fragment, list all the dependencies between statements and draw the dependency graph. [*2006. Marks: 3*]**

```
(1)   j = 4            (5)   m = m + 2
(2)   k = j + 1        (6)   k = j + l
(3)   j = 6            (7)   j = k + j
(4)   m = k * j
```

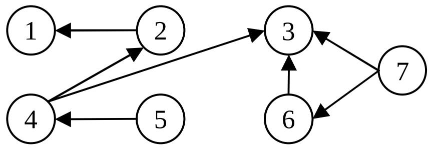
List of dependencies between statements:

1. (2) depends on (1) for value of *j*.
2. (4) depends on both (2) and (3) for values of *k* and *j* respectively.
3. (5) depends on (4) for value of *m*.

4. (6) depends on (3) for value of *j*.
5. (7) depends on both (3) and (6) for values of *j* and *k* respectively.

Dependency Graph:

Let, for statements *A* and *B*, (A) → (B) denotes that *A* is dependent on *B*.



9.4 **Consider the following code fragment. [*2003. Marks: 3 + 2 + 7*]**

```
begin
  for i := 1 to n do
       for j := 1 to n do
            begin
                 c[i,j] := 0;
                 for k := 1 to n do
                      c[i,j] = c[i,j] + a[i,k] * b[k,j];
            end
end
```

i. **Assume a, b and c are allocated static storage and there are 2 bytes per word in a byte addressable memory. Produce three-address code for the code fragment.**
ii. **Construct the flow graph from the three address statement.**
iii. **Optimize the code by eliminating common sub-expressions, eliminating dead-code and different loop optimization techniques.**

9.5 **Consider the following code segment. [*2006. Marks: 3 + 5 + 4*]**

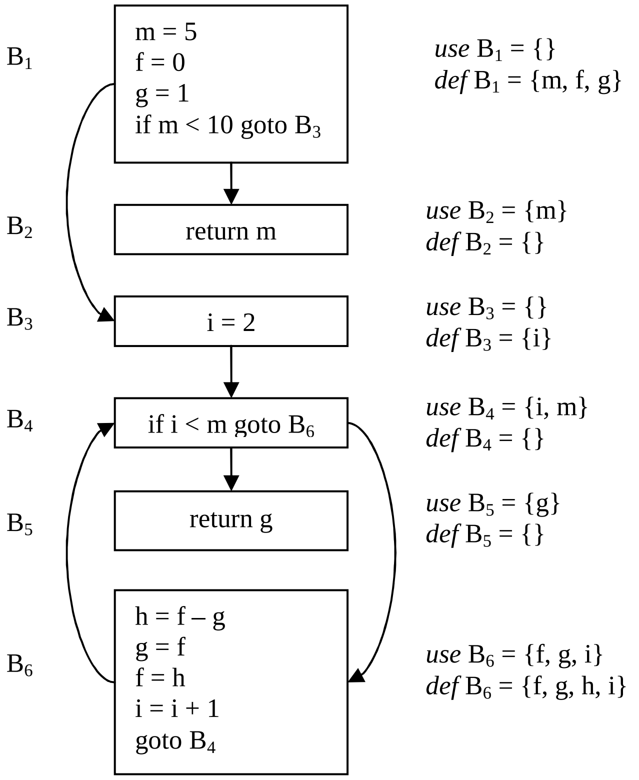
```
(1)   i = b + c                 (10)   goto (13)
(2)   b = 10                    (11)   g = -k
(3)   k = 9                     (12)   f = d + 4
(4)   a = b + c                 (13)   c = 25
(5)   d = e + f                 (14)   h = e + 10
(6)   k = j + 1                 (15)   j = b + j
(7)   if p > 10 goto (9)        (16)   if q > 10 goto (4)
(8)   goto (11)                 (17)   exit
(9)   e = 5
```

i. **Draw the control flow graph (CFG).**
ii. **Perform global CSE and draw the CFG for the code that results from CSE. You need not to show the computation for finding available expressions. Only show the available expressions at the input of each basic block.**
iii. **Find all natural loops and identify the loop invariant statements. Which statements are safe to be moved to the loop's pre-header and why?**

9.6 **Consider the code segment below: [*2004. Marks: 3 + 6*]**

```
(1)   m = 5                     (8)    return g
(2)   f = 0                     (9)    h = f – g
(3)   g = 1                     (10)   g = f
(4)   if m < 10 goto (6)        (11)   f = h
(5)   return m                  (12)   i = i + 1
(6)   i = 2                     (13)   goto (7)
(7)   if i < m goto (9)
```

i. **Construct a flow graph.**
ii. **Find the live variables at the end of each block.**



	$IN[B]^0$	$OUT[B]^1$	$IN[B]^1$	$OUT[B]^2$	$IN[B]^2$	$OUT[B]^3$	$IN[B]^3$
B_1	{ }	{ f, g, m }	{ }	{ f, g, m }	{ }	{ f, g, m }	{ }
B_2	{ }	{ }	{ m }	{ }	{ m }	{ }	{ m }
B_3	{ }	{ f, g, i, m }	{ f, g, m }	{ f, g, i, m }	{ f, g, m }	{ f, g, i, m }	{ f, g, m }
B_4	{ }	{ f, g, i }	{ f, g, i, m }	{ f, g, i, m }	{ f, g, i, m }	{ f, g, i, m }	{ f, g, i, m }
B_5	{ }	{ }	{ g }	{ }	{ g }	{ }	{ g }
B_6	{ }	{ }	{ f, g, i }	{ f, g, i, m }	{ f, g, i, m }	{ f, g, i, m }	{ f, g, i, m }

∴ Live variables after each block:

B_1: { f, g, m }
B_2: { }
B_3: { f, g, i, m }
B_4: { f, g, i, m }
B_5: { }
B_6: { f, g, i, m }

[*Points to be noted from this answer:*

1. There is no edge from the block containing ***return m*** to the following block, because after returning from a block, the program never flows below in the block. The same case holds for ***return g***, too. This is according to the rule of putting an edge in flow graph (from page 529, bullet point no. 2).
2. As there is no edge outgoing from ***return m***, hence its OUT is empty. However, its IN is not empty as it *use*s ***m***.
3. Block B_4 should be counted – even though there is no variable assignment statement. That's because the compiler must know to before comparing ***i < m*** whether both are live or not. And after the end of the block, should ***i*** and ***m*** be dead and discarded or not. Similarly, ***m*** in B_1 should also be counted.
4. ***m*** is not *use*d in B_1 as can be mistakenly assumed from the ***if*** instruction. That's because *m* is

defined *before* it is used in this block. From the definition of use_B (from page 609, point 2), ***m*** is not *use*d. Similarly, ***h*** is not *use*d in B_6. (Also see page 609, the paragraph after *Example 9.13* for further clarification.)

5. Although usually live variable analysis includes a block containing **EXIT**, in this particular case, there should be no exit block. That's because the last block B_6 includes as its last statement an ***un****conditional* jump – *goto* B_3, after executing which the program control will always flow to B_4 and never to any other block.]

9.7 **Consider the following sequence of 3-address codes: [*In-course. Marks: 4 + 6*]**

```
(1)   e = e – b
(2)   d = a * c
(3)   if e < d goto (1)
(4)   i = e + f
(5)   j = a + b
(6)   c = c * 2
(7)   if c > d goto (9)
(8)   goto (11)
(9)   g = a * c
(10)  goto (13)
(11)  i = d * d
(12)  j = c + 1
(13)  if i > j goto (5)
(14)  exit
```

i. **Draw the flow graph.**
ii. **Compute live variables at the end of each block using the iterative solution to dataflow equations for live variable analysis.**

9.8 **Consider the following sequence of 3-address codes: [*In-course. Marks: 2 + 6 + 2*]**

```
(1)   c = a + b
(2)   d = a * c
(3)   e = d * d
(4)   i = 1
(5)   f[i] = a + b
(6)   c = c * 2
(7)   if c > d goto (9)
(8)   goto (11)
(9)   g[i] = a * c
(10)  goto (13)
(11)  g[i] = d * d
(12)  i = i + 1
(13)  if i > 10 goto (5)
(14)  exit
```

i. **Draw the flow graph.**
ii. **Compute the available expressions at the beginning of each block using the iterative solution to dataflow equations for available expressions.**
iii. **Draw the flow diagram after global CSE.**

9.9 **Consider the following sequence of 3-address codes: [*2007. Marks: 4 + 6*]**

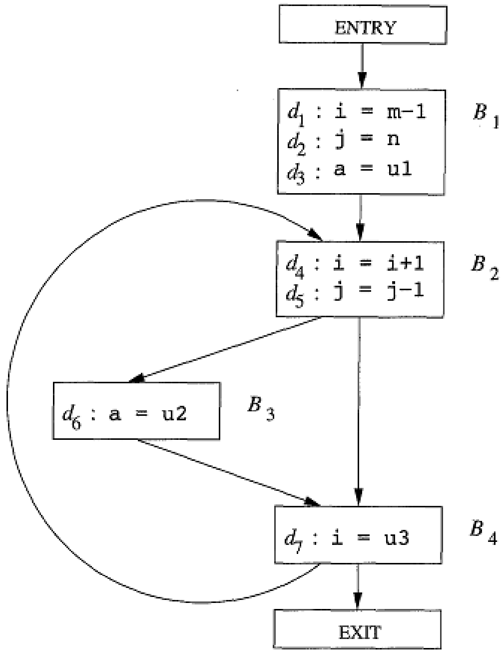
```
(1)   a = a – d
(2)   f = b * d
(3)   c = a + b
(4)   d = c – a
(5)   if d > x goto (7)
(6)   d = b * d
(7)   b = a + b
(8)   e = c – a
(9)   if e > 10 goto (3)
(10)  goto (13)
(11)  a = b * d
(12)  b = a – d
(13)  if b > 10 goto (1)
(14)  exit
```

i. **Draw the flow graph.**
ii. **Compute live variables at the end of each block using the iterative solution to dataflow equations for live variable analysis.**
iii. **Show the execution of the algorithm solving the data flow equations set up for available expressions. [*In-course. Marks: 6*]**

9.10 **Consider the following sequence of 3-address codes: [*2005. Marks: 2 + 5*]**

```
(1)   i = m – 1
(2)   j = n
(3)   a = u1
(4)   i = i + 1
(5)   j = j - 1
(6)   if i > j goto (8)
(7)   goto (9)
(8)   a = u2
(9)   i = u3
(10)  if u3 > 0 goto (4)
(11)  exit
```

	i. Draw the flow graph. **ii. Find the definitions reaching the end of each block by iteratively solving the dataflow equations for reaching definitions.**



$gen_{B_1} = \{ d_1, d_2, d_3 \}$

$kill_{B_1} = \{ d_4, d_5, d_6, d_7 \}$

$gen_{B_2} = \{ d_4, d_5 \}$

$kill_{B_2} = \{ d_1, d_2, d_7 \}$

$gen_{B_3} = \{ d_6 \}$

$kill_{B_3} = \{ d_3 \}$

$gen_{B_4} = \{ d_7 \}$

$kill_{B_4} = \{ d_1, d_4 \}$

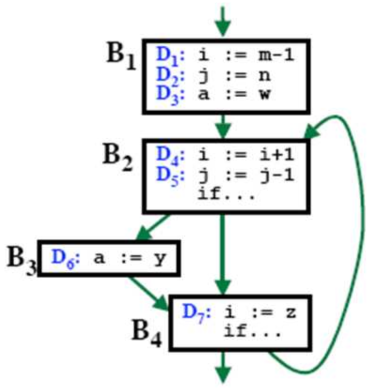
Block B	OUT$[B]^0$	IN$[B]^1$	OUT$[B]^1$	IN$[B]^2$	OUT$[B]^2$
B_1	000 0000	000 0000	111 0000	000 0000	111 0000
B_2	000 0000	111 0000	001 1100	111 0111	001 1110
B_3	000 0000	001 1100	000 1110	001 1110	000 1110
B_4	000 0000	001 1110	001 0111	001 1110	001 0111
EXIT	000 0000	001 0111	001 0111	001 0111	001 0111

∴ Definitions reaching at the end of each block:

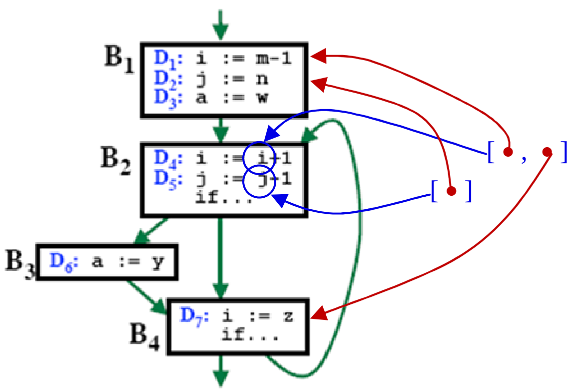
B_1: { d_1, d_2, d_3 }
B_2: { d_3, d_4, d_5, d_6 }
B_3: { d_4, d_5, d_6 }
B_4: { d_3, d_5, d_6, d_7 }

9.11 **Consider the following flow graph: [*In-course 3, 2008-2009. Marks: 4 + 2*]**

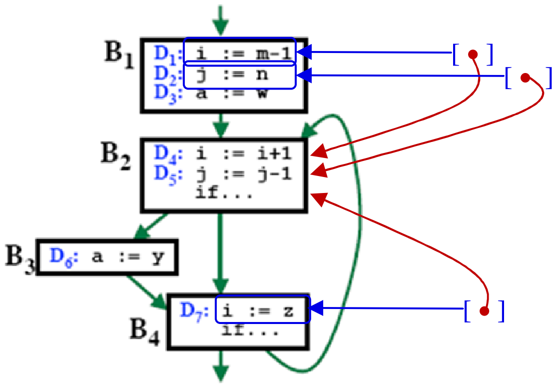
i. Compute UD and DU-chain for the flow graph.
ii. Compute live variables at the end of each block of the flow graph.



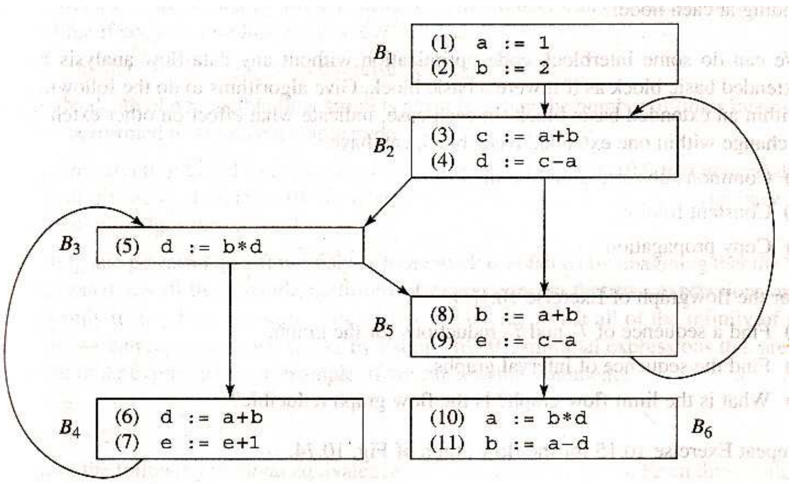
i. UD-Chain:



DU-Chain:



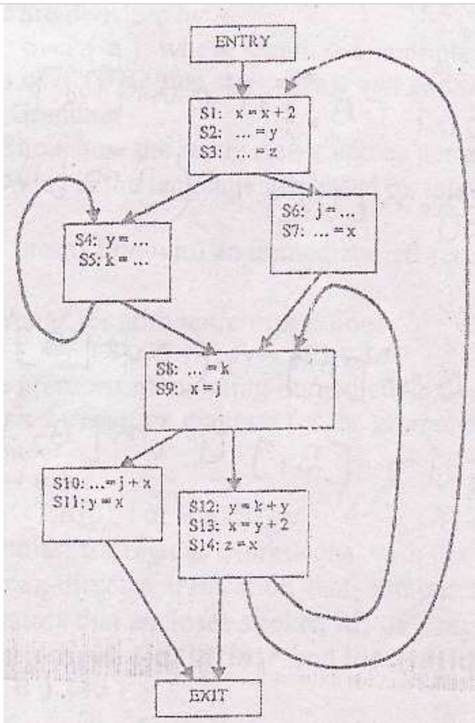
9.12 Consider the following flow graph: [*2008. Marks: 4 + 2 + 2 + 4 + 4*]



- **i. Compute UD and DU-chain for the above flow graph.**
- **ii. Compute live variables at the end of each block of the flow graph.**
- **iii. Compute available expressions for the flow graph.**
- **iv. Is any constant folding possible in the flow graph? If so, do it.**
- **v. Are there any common sub-expressions in the flow graph? If so, do it.**

9.13 Consider the CFG below where only definition (v = …) and use (… = v) of the interesting variables are shown.

- **i. List the webs as web_i = {list of statements S_ks in web_i}.**
- **ii. Draw the interference graph.**
- **iii. If coloring is possible and if you have three registers R_0, R_1, R_2, identify for each node the allocated register allocation. Assume the variable k cannot be allocated to register R_2. Otherwise, which node would you choose to spill and why? [*2006. Marks: 2 + 3 + 3*]**



9.14	**Consider the CFG below where only definition (v = …) and use (… = v) of the interesting variables are shown.** **i. List the webs as web_i = {list of statements S_ks in web_i}.** **ii. Draw the interference graph.** **iii. If you have three registers, show the code after register allocation (if coloring is possible). [*2005. Marks: 2 + 3 + 3*]**