# 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 or 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 2024–2025

**End-Semester Test**

19–November–2024 09:00am–12:00pm Maximum marks: 100

[*Write your answers in the question paper itself. Be brief and precise. Answer all questions.*]

Do not write anything on this page.

## 1. [Bottom-up parsing]

**(a)** Let $L$ be the language generated by the following grammar. The terminal symbols are $a$ and $b$, and the start symbol is $S$.

$$S \rightarrow aSb \mid aSbb \mid b$$

Demonstrate by an example that this grammar is ambiguous. **(3)**

*Solution* The string $aabbbb$ has two different derivations.

$$S \rightarrow aSb \rightarrow aaSbbb \rightarrow aabbbb$$

$$S \rightarrow aSbb \rightarrow aaSbbb \rightarrow aabbbb$$

The parse trees for these two derivations are certainly different. For example, the root node of the parse tree for the first derivation has three children, whereas that for the second derivation has four children.

**(b)** A (canonical) LR(1) parser for an ambiguous grammar must encounter conflicts. We propose the following alternative grammar for $L$ defined in Part (a), where $T$ is another nonterminal symbol ($S$ continues to remain the start symbol).

$$\begin{aligned} S &\rightarrow aSb \mid T \\ T &\rightarrow aTbb \mid b \end{aligned}$$

Formally justify that this grammar is unambiguous. Specific examples alone will not do. **(3)**

*Solution* We have $L = \{a^m b^n \mid 0 < m < n \leqslant 2m+1\}$. For $a^m b^n \in L$, write $m = k + l$ and $n = k + (2l+1)$ for some $k \geqslant 0$ and $l \geqslant 0$. Indeed, we have $l = n - m - 1 \geqslant 0$, and $k = m - l = 2m - n + 1 \geqslant 0$. In order to show that this decomposition is unique, let $m = k + l = k' + l'$ and $n = k + (2l+1) = k' + (2l'+1)$. But then $k - k' = l' - l = 2(l' - l)$, so $l' = l$ and consequently $k' = k$.

The start symbol $S$ in the new grammar first generates $k$ $a$'s and $k$ $b$'s. It then converts to $T$ which first generates $l$ $a$'s and $2l$ $b$'s. Eventually, $T$ vanishes by generating the last $b$. In no other way, sentences can be generated by this grammar. The uniqueness of $k$ and $l$ as established above guarantees that no other derivation process can generate this $a^m b^n$.

Note that the argument that $S$ working first followed by $T$ removes ambiguities is not complete. Consider a similar language $L'$ generated by the following grammar.

$$S \rightarrow aSb \mid aSbb \mid aSbbb \mid b$$

This grammar is clearly ambiguous. Let us do the following transformation to this grammar.

$$\begin{aligned} S &\rightarrow aSb \mid T \\ T &\rightarrow aTbb \mid U \\ U &\rightarrow aUbbb \mid b \end{aligned}$$

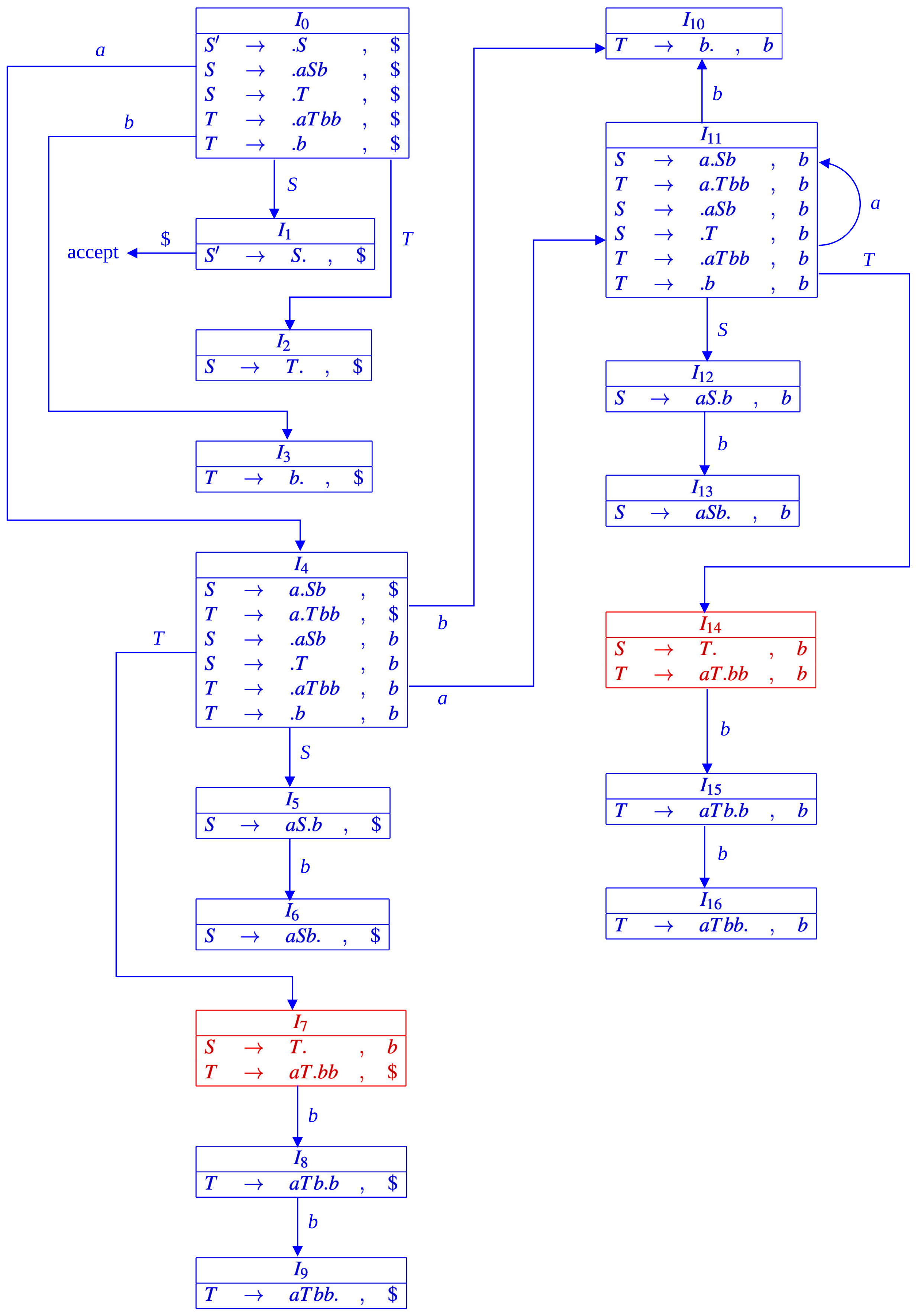
This new grammar for $L'$ fixes the order of the working of the nonterminals $S, T, U$. But we have:

$$S \rightarrow aSb \rightarrow aTb \rightarrow aUb \rightarrow aaUbbbb \rightarrow aabbbbb$$

$$S \rightarrow T \rightarrow aTbb \rightarrow aaTbbbb \rightarrow aaUbbbb \rightarrow aabbbbb$$

The problem here is that $2 + 2 = 1 + 3$. An argument that this type of situation cannot arise in the grammar of Part (b) is rather essential.

**(c)** We now plan to investigate whether the unambiguous grammar of Part (b) is LR(1). In order to do so, first draw the complete LR(1) automaton for the grammar of Part (b). **(10)**



**(d)** Conclude from the LR(1) automaton of Part (c) whether the grammar of Part (b) is LR(1). Supply proper justification. **(2)**

*Solution* The states $I_7$ and $I_{14}$ have shift-reduce conflicts on input symbol $b$, so the grammar of Part (b) is not LR(1).

**(e)** Justify whether the grammar of Part (b) is LALR(1). **(2)**

*Solution* The shift-reduce conflicts of the LR(1) automaton will continue to stay in the corresponding LALR(1) automaton. So the grammar is not LALR(1) either. (Whether merging of states introduces new conflicts does not matter, because the grammar is already non-LALR.)

**2. [Syntax-directed translation]**

Consider the following grammar for the assignment (to a variable) of arithmetic expressions involving the operators $+$, $-$, $*$, and $/$. Assume that each operand in the arithmetic expression (the terminal NUM in the grammar) is a positive integer. ID is another terminal symbol standing for the name of a variable. The nonterminal symbols are $A$ (assignment statement, start symbol), $E$ (expression), and $T$ (term).

$$\begin{aligned} A &\rightarrow \text{ID} = E \\ E &\rightarrow T \mid T + E \mid T - E \\ T &\rightarrow \text{NUM} \mid \text{NUM} * T \mid \text{NUM} / T \end{aligned}$$

All of the four operators $+$, $-$, $*$, and $/$ are left-to-right associative. The precedence of $*$ and $/$ over $+$ and $-$ is already taken care of by the grammar.

The grammar symbols have two integer-valued attributes *inh* (inherited) and *val* (synthesized). NUM, $E$, and $A$ do not use the attribute *inh*. The nonterminals $T$ and $E$ possess another inherited attribute *op* (the **preceding** operator; its value can be ADD or SUB for $E$, and MUL or DIV for $T$). Fill in the blanks in the following actions of the SDD (syntax-directed definition) to compute $A.val$. **(12)**

$A \rightarrow \text{ID} = E$ { $A.val = E.val$
  $E.op = \text{ADD}$
}

$E \rightarrow T$ { $T.inh = (E.op == \text{ADD})\ ?$ ____1____ : ____$-1$____

  $T.op =$ ____MUL____

  $E.val =$ ____$T.val$____
}

$E \rightarrow T + E_1$ { $T.inh = (E.op == \text{ADD})\ ?$ ____1____ : ____$-1$____

  $T.op =$ ____MUL____

  $E_1.op =$ ____ADD____

  $E.val =$ ____$T.val + E_1.val$____
}

$E \rightarrow T - E_1$ { $T.inh = (E.op == \text{ADD})\ ?$ ____1____ : ____$-1$____

  $T.op =$ ____MUL____

  $E_1.op =$ ____SUB____

  $E.val =$ ____$T.val + E_1.val$____
}

```
$T \to \text{NUM}$          { if ($T.op == \text{MUL}$)
                                  $T.val =$ $T.inh \times \text{NUM}.val$
                              else
                                  $T.val =$ $T.inh \ / \ \text{NUM}.val$
                              }
```

```
$T \to \text{NUM} * T_1$    { if ($T.op == \text{MUL}$)
                                  $T_1.inh =$ $T.inh \times \text{NUM}.val$
                              else
                                  $T_1.inh =$ $T.inh \ / \ \text{NUM}.val$
                              $T_1.op =$ MUL
                              $T.val =$ $T_1.val$
                              }
```

```
$T \to \text{NUM} \ / \ T_1$    { if ($T.op == \text{MUL}$)
                                  $T_1.inh =$ $T.inh \times \text{NUM}.val$
                              else
                                  $T_1.inh =$ $T.inh \ / \ \text{NUM}.val$
                              $T_1.op =$ DIV
                              $T.val =$ $T_1.val$
                              }
```

**3. [Intermediate-code generation]**

In this exercise, we continue with the grammar of Exercise 2, but handle multiple assignment statements. We plan for incremental code generation in conjunction with bottom-up parsing (like LALR(1) parsing). The inherited attributes used in Exercise 2 are problematic for this purpose. We use some global data structures (explained later in detail) to store operands and operators. We also use four marker nonterminals $M, N, P, Q$. The revised grammar is given below. Here, $L$ (list of assignments) is the new start symbol. The earlier nonterminals have the same meanings: $A$ generates an assignment statement, $E$ an expression, and $T$ a term. Each individual factor of a term is a positive integer (the terminal NUM).

$$\begin{array}{rcl}
L & \rightarrow & A\,M \mid A\,M\,L \\
A & \rightarrow & \text{ID} = E \\
E & \rightarrow & T\,N \mid T\,N + P\,E \mid T\,N - Q\,E \\
T & \rightarrow & \text{NUM} \mid \text{NUM} * T \mid \text{NUM}\,/\,T \\
M & \rightarrow & \varepsilon \\
N & \rightarrow & \varepsilon \\
P & \rightarrow & \varepsilon \\
Q & \rightarrow & \varepsilon
\end{array}$$

We plan to store the intermediate code in a table $Q$ of quads. An example is given below. The different assignments are separated by lines of dashes. The temporaries are numbered as \$1, \$2, \$3, and so on.

| **Input** | **Table $Q$ of quads** |
|---|---|
| `a = 1`<br>`b = 1 + 2 + 3`<br>`c = 4 * 3 / 2`<br>`d = 1 - 2 * 3 * 4 / 5 * 6 / 7 - 8 + 9 * 10` | `OP      ARG1     ARG2     RES`<br>`--------------------------------`<br>`=       1                 a`<br>`--------------------------------`<br>`=       1                 b`<br>`+       b        2        b`<br>`+       b        3        b`<br>`--------------------------------`<br>`*       4        3        $1`<br>`/       $1       2        $2`<br>`=       $2                c`<br>`--------------------------------`<br>`*       2        3        $3`<br>`*       $3       4        $4`<br>`/       $4       5        $5`<br>`*       $5       6        $6`<br>`/       $6       7        $7`<br>`*       9        10       $8`<br>`=       1                 d`<br>`-       d        $7       d`<br>`-       d        8        d`<br>`+       d        $8       d`<br>`--------------------------------` |

Let us look at a term $T$ first. Every production for $T$ reveals a new NUM operand. We store these numbers in a global integer array `factorlist` and the corresponding operators (* and /) in a global character array `factorops`. Moreover, a global integer variable `factorno` stores the number of factors in a term. Since we use bottom-up parsing and the productions of $T$ do not involve marker nonterminals, the operands are stored in the reverse sequence as they appear in the input. For example, for the term `2 * 3 * 4 / 5 * 6 / 7`, we generate `factorlist = ( 7, 6, 5, 4, 3, 2 )` and `factorops = ( /, *, /, *, * )`. We also have `factorno = 6`.

After an instance of $T$ is reduced, the marker $N$ springs into action. It prepares a term for an expression. A term is a pair $(type, value)$. If `factorlist` contains a single operand, then *type* is NUM, and *value* is the lexical value of that NUM. If `factorlist` contains multiple factors, then temporaries are used to generate the value of the term, so *type* is TMP, and *value* is the number of the temporary storing the final value of the term. The action for the production of $N$ calls a function `evalterm()` for that purpose. This function returns the calculated pair $(type, value)$. The action then resets `factorno` to 0 (for the next term). The action also stores the returned pair in a global array `termlist` of pairs. The markers $P$ and $Q$ are used to store the signs ($+$ or $-$) of the terms, in a global character array `termsigns`. The number of terms in an expression is stored in the global integer variable `termno`. For the assignment of `d` in the above example, we eventually have `termlist = { (NUM,1), (TMP,7), (NUM,8), (TMP,8) }`, `termsigns = { -, -, + }`, and `termno = 4`. The terms and signs are stored in the same order as they appear in the input.

The action against the production of $A$ generates the code for setting ID to the signed addition of terms in `termlist`. The function `evalexpr(ID)` is called to do that. Finally, the action against the production of the marker $M$ resets `termno` to 0 as a preparation for the next assignment instruction.

Fill in the pseudocode below in order to store the intermediate code for $L$ in $Q$. Assume that a global variable `quadno` stores the number of quads stored in $Q$. We also have a global variable `tmpno` to store the number of the temporary last generated.

Only the following productions have non-empty actions. **(4)**

$M$ is used only to reset `termno` for the next assignment.

```
M → ε      { termno = 0; }
```

$N$ is used to store the reference to a term in `termlist`, and to prepare for the next term.

```
N → ε      { ________termlist[termno]________ = evalterm(); ++termno;
             factorno = 0;
           }
```

$P$ and $Q$ are used to store the signs of the terms.

```
P → ε      { ________termsigns[termno]________ = '+' }

Q → ε      { ________termsigns[termno]________ = '-' }
```

$T$ is used to add a factor and an operator (except for the last factor) to `factorlist` and `factorops`.

```
T → NUM:      {
                 ________factorlist[factorno]________ = NUM.val;
                 ++factorno;
              }
T → NUM * T1:    {
                    ________factorlist[factorno]________ = NUM.val;
                    ________factorops[factorno]________ = '*';
                    ++factorno;
                 }
T → NUM / T1:    {
                    ________factorlist[factorno]________ = NUM.val;
                    ________factorops[factorno]________ = '/';
                    ++factorno;
                 }
A → ID = E:    { evalexpr(ID); }
```

Now, write the pseudocode for the function **evalterm**. This function returns a $(type, value)$ pair. Each quad in the quad table $Q$ is a 4-tuple $(op, arg1, arg2, res)$.

```
pair evalterm ( )
{
   /* If there is only one factor */                                              (1)

      if (factorno == 1) return (NUM,factorlist[0])

   /* For multiple factors, temporaries have to be created */
   /* The first operation is that of only numeric operands */                      (3)

      ++tmpno;
      Q[quadno] = (factorops[factorno-1],factorlist[factorno-1],factorlist[factorno-2],$tmpno);
      ++quadno;

   /* Other operations involve the previous temporary and the next operand */      (3)

      for i = (factorno - 3) downto 0 {
         ++tmpno;
         Q[i] = (factorops[i+1],$(tmpno-1),factorlist[i],$tmpno);
         ++quadno;
      }

   /* Return a temporary */                                                        (1)

   return (TMP,tmpno)

}
```

Finally, complete the pseudocode for **evalexpr**.

```
evalexpr ( var )
{
/* Set var to the first term (a NUM or a TMP) */                                   (3)

   if (termlist[0].type == NUM) Q[quadno] = ('=',termlist[0].value,0,var);
   else Q[quadno] = ('=',$termlist[0].value,0,var);
   ++quadno;

/* For the remaining terms, add or subtract the next term to var */                (3)

   for i = 1 to termno-1 {
      if (termlist[i].type == NUM) Q[quadno] = (termsign[i],var,termlist[i].value,var);
      else Q[quadno] = (termsign[i],var,$termlist[i].value,var);
      ++quadno;
   }

}
```

**4. [Backpatching]**

Consider the following grammar with start symbol $S$.

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{while } B \textbf{ do } S \mid \textbf{begin } L \textbf{ end} \mid A \\
L & \rightarrow & L\,S \mid S \\
A & \rightarrow & \textbf{id} = E \\
E & \rightarrow & E + E \mid E - E \\
B & \rightarrow & E \textbf{ relop } E \\
B & \rightarrow & (\,B\,) \\
B & \rightarrow & B \;\&\&\; B \\
B & \rightarrow & B \;||\; B \\
B & \rightarrow & \textbf{true} \\
B & \rightarrow & \textbf{false} \\
E & \rightarrow & \textbf{id}
\end{array}
$$

Here, **relop** indicates the relational operators, such as $<$, $>$, and so on.

**(a)** Augment the above grammar with a marker nonterminal $M$ at suitable places of the productions such that backpatching can be handled. **(3)**

*Solution*

**(b)** Write the semantic actions to design a suitable syntax-directed translator (SDT) which generates three-address codes for the above grammar. Note that the semantic actions will not generate parse trees, but should handle backpatching to generate three-address codes. **(5)**

*Solution*

**(c)** Apply your SDT to translate the code snippet to the right, to the three-address code. Assume that address generation starts from the address 100, and all labels are instruction numbers. Draw the suitably annotated parse tree, and clearly show the backpatching steps (the goto Label statements before and after the backpatching). Clearly show the annotations of the parse tree, and the consequent generation of three-address code, and the backpatching steps. **(7)**

```
begin
    while $a > b$ || false do
    begin
        $x = y + z$
        $a = a - b$
    end
    $x = y - z$
end
```

*Solution*

**5. [Target-code generation]**

**(a)** Suppose that you have a RISC-like processor with four registers R1, R2, R3, and R4, and machine instructions (ADD, MUL, SUB) are available for each basic operation OP, in the following form.

```
OP reg1, reg2, reg3,
```

Here, `reg1` is the register which stores the result after operating on `reg2` and `reg3`. Two dedicated instructions

`LD reg, mem` and `ST mem, reg`

are available for the load and store operations.

(i) Now, consider the assignment $d = (a-b) + b*c + (b+c)$. Directly write the corresponding three-address intermediate code (no need for showing and annotating the parse tree). Assume that these three-address instructions constitute a single basic block. **(1)**

*Solution*

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

(ii) From the above intermediate code, generate the target code using the simple target-code generation algorithm. Assume that all registers are free at the beginning and at the end of the block. Clearly show:

- Machine instructions generated **(3)**
- States of the Register-Descriptor and the Address-Descriptor tables before and after each machine instruction generation. **(4)**
- At each step of the translation, apply the GetReg(I) algorithm to obtain the required registers, for the generation of the machine code. Suitably show if any spill operation (that is, write back to memory) is required. Note that GetReg() can access the Register- and Address-Descriptor tables. Attach your comments on the register assignment after each step. **(4)**

*Solution*

**(b)** A basic block consists of the following four three-address instructions.

```
I1:  x = y + z
I2:  p = x + q
I3:  m = p + x
I4:  y = x + z
```

Here, `I1`, `I2`, `I3`, `I4` are the indices of the three-address instructions. Assume that all the variables are live on exit. For each instruction `Ix` in this block, assign the liveness for all the variables, in the table on the next page. If a variable is live, specify its next use as an instruction `Iy` (in this block) or as *other block* (whichever is applicable). Otherwise, specify *not live*. **(4)**

| | $x$ | $y$ | $z$ | $p$ | $q$ | $m$ |
|---|---|---|---|---|---|---|
| I1 | *not live* | I1 | I1 | *not live* | I2 | *not live* |
| I2 | I2 | *not live* | I4 | *not live* | I2 | *not live* |
| I3 | I3 | *not live* | I4 | I3 | *other block* | *not live* |
| I4 | I4 | *not live* | I4 | *other block* | *other block* | *other block* |

**(c)** A basic block consists of the following three-address code.

```
p = a[i]
a[j] = y
z = a[i]
m = a[i]
x = y + z
t = z + q
r = t + y
```

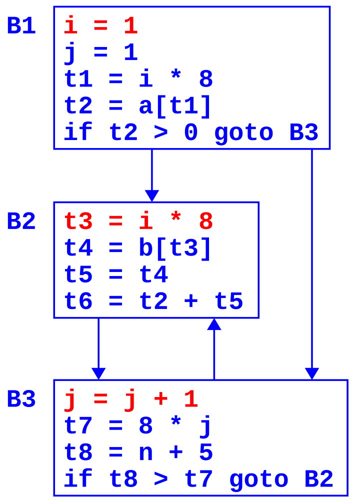
First, construct a DAG for this basic block. Next, optimize the DAG. Assume that `t` is a temporary not used in any other block, whereas all other variables are live on exit. Show each step of your optimization process. Finally, rewrite the basic block from the optimized DAG. Show each step in this process. **(4)**

**6. [Global optimization]**

**(a)** Consider the following three-address code.

```
          i = 1
          j = 1
          t1 = i * 8
          t2 = a[t1]
          if t2 > 0 goto L2
     L1:  t3 = i * 8
          t4 = b[t3]
          t5 = t4
          t6 = t2 + t5
     L2:  j = j + 1
          t7 = 8 * j
          t8 = n + 5
          if t8 > t7 goto L1
```

(i) Identify the leaders, construct the basic blocks, and construct the flow graph. **(2)**



(ii) Assume that the temporaries `t1` through `t8` are used nowhere else outside the block. Step by step apply various intermediate-code optimization techniques to optimize the basic blocks globally. Show each optimization step clearly, specify the name of the technique, and show the final optimized basic blocks. (You do not have to optimize the goto's.) **(3)**

*Solution* Global common subexpression: `i * 8` is used twice.

Constant folding: `i` always stays 1, so both the occurrences of `i * 8` can be replaced by 8, and `t1` and `t3` do not need to be computed at all (dead-code elimination).

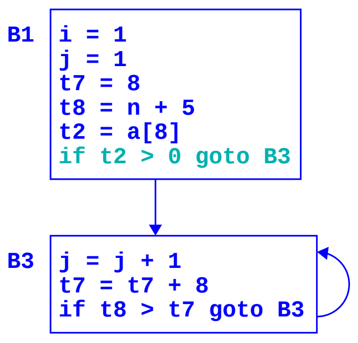
Dead-code elimination: The entire block `B2` can be eliminated, because it computes some temporaries that are used nowhere else.

Code motion: `t8 = n + 5` remains constants in all the loops, and can be taken out of the loop and moved to block `B1`.

Strength reduction: `t7` can be computed as `t7 = t7 + 8`. The initialization `t7 = 8` is introduced in block `B1`.

Note that the initialization and increments of `j` cannot be eliminated because `j` may be live on exit. However, the `goto B3` statement can also be eliminated, because irrespective of the condition `t2 > 0`, control will go to block `B3`.

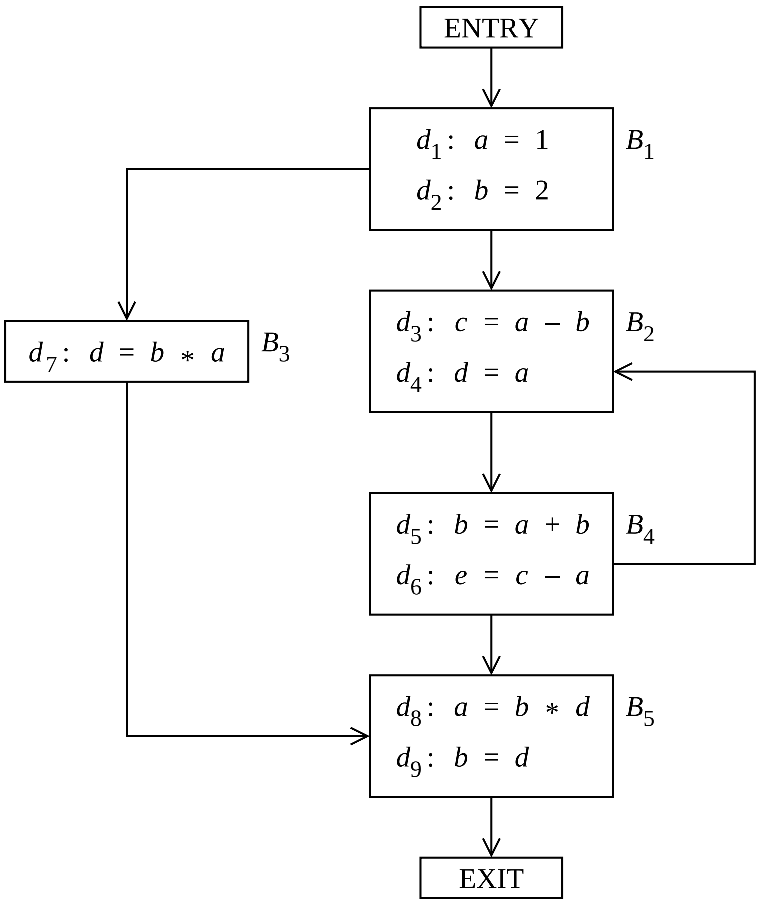
The optimized flow graph is given below.



B1
```
i = 1
j = 1
t7 = 8
t8 = n + 5
t2 = a[8]
if t2 > 0 goto B3
```

B3
```
j = j + 1
t7 = t7 + 8
if t8 > t7 goto B3
```

**(b)** Consider the flow graph given below.



Here, we focus on the reaching-definition problem for data-flow analysis. The definitions are shown as $d_i$.

(i) Write down the GEN[$B$] and the KILL[$B$] sets for each basic block $B$, in the following table. **(2)**

| | |
|---|---|
| GEN[$B_1$] = $\{d_1, d_2\}$ | KILL[$B_1$] = $\{d_5, d_8, d_9\}$ |
| GEN[$B_2$] = $\{d_3, d_4\}$ | KILL[$B_2$] = $\{d_7\}$ |
| GEN[$B_3$] = $\{d_7\}$ | KILL[$B_3$] = $\{d_4\}$ |
| GEN[$B_4$] = $\{d_5, d_6\}$ | KILL[$B_4$] = $\{d_2, d_9\}$ |
| GEN[$B_5$] = $\{d_8, d_9\}$ | KILL[$B_5$] = $\{d_1, d_2, d_5\}$ |

(ii) Write the two functions for each block $B$, to compute the data-flow values IN[$B$] and OUT[$B$]. **(2)**

IN[$B_1$] = OUT(ENTRY)

OUT[$B_1$] = $\{d_1, d_2\} \cup \left(\text{IN}(B_1) - \{d_5, d_8, d_9\}\right)$

IN[$B_2$] = OUT($B_1$) ∪ OUT($B_4$)

OUT[$B_2$] = $\{d_3, d_4\} \cup \left(\text{IN}(B_2) - \{d_7\}\right)$

$$\mathrm{IN}[B_3] = \mathrm{OUT}(B_1)$$

$$\mathrm{OUT}[B_3] = \{d_7\} \cup \Big(\mathrm{IN}(B_3) - \{d_4\}\Big)$$

$$\mathrm{IN}[B_4] = \mathrm{OUT}(B_2)$$

$$\mathrm{OUT}[B_4] = \{d_5, d_6\} \cup \Big(\mathrm{IN}(B_4) - \{d_2, d_9\}\Big)$$

$$\mathrm{IN}[B_5] = \mathrm{OUT}(B_3) \cup \mathrm{OUT}(B_4)$$

$$\mathrm{OUT}[B_5] = \{d_8, d_9\} \cup \Big(\mathrm{IN}(B_5) - \{d_1, d_2, d_5\}\Big)$$

(iii) Use these functions to compute the IN[$B$] and OUT[$B$] for each block $B$. In this computation, strictly follow the order $B_1, B_2, B_3, B_4, B_5$ to compute the data-flow values. Show the outcome of the first two iterations. Use the bitmap representation to specify and update the values of IN and OUT.

**Initialization:**

OUT[ENTRY] = OUT[$B_1$] = OUT[$B_2$] = OUT[$B_3$] = OUT[$B_4$] = OUT[$B_5$] = 000000000

**Iteration 1:** **(3)**

IN[$B_1$] = OUT[ENTRY] = 000000000

OUT[$B_1$] = $110000000 + (000000000 - 000010011) = 110000000$

IN[$B_2$] = $\mathrm{OUT}[B_1] \cup \mathrm{OUT}[B_4] = 110000000 + 000000000 = 110000000$

OUT[$B_2$] = $001100000 + (110000000 - 000000100) = 111100000$

IN[$B_3$] = OUT[$B_1$] = 110000000

OUT[$B_3$] = $000000100 + (110000000 - 00010000) = 110000100$

$\text{IN}[B_4] = \text{OUT}[B_2] = 111100000$

$\text{OUT}[B_4] = 000011000 + (111100000 - 010000001) = 101111000$

$\text{IN}[B_5] = \text{OUT}[B_3] \cup \text{OUT}[B_4] = 110000100\text{+}101111000\text{=}111111100$

$\text{OUT}[B_5] = 000000011 + (111111100 - 110010000) = 001101111$

**Iteration 2:** **(3)**

$\text{IN}[B_1] = \text{OUT[ENTRY]} = 000000000$

$\text{OUT}[B_1] = 110000000 + (000000000 - 000010011) = 110000000$

$\text{IN}[B_2] = \text{OUT}[B_1] \cup \text{OUT}[B_4] = 110000000 + 101111000 = 111111000$

$\text{OUT}[B_2] = 001100000 + (111111000 - 000000100) = 111111000$

$\text{IN}[B_3] = \text{OUT}[B_1] = 110000000$

$\text{OUT}[B_3] = 000000100 + (110000000 - 00010000) = 110000100$

$\text{IN}[B_4] = \text{OUT}[B_2] = 111111000$

$\text{OUT}[B_4] = 000011000 + (111111000 - 010000001) = 101111000$

$\text{IN}[B_5] = \text{OUT}[B_3] \cup \text{OUT}[B_4] = 110000100\text{+}101111000\text{=}111111100$

$\text{OUT}[B_5] = 000000011 + (111111100 - 110010000) = 001101111$