CS31003 Compilers
Autumn 2026, Class Test 2
29-Oct-2026, 5:45pm–7:00pm

Maximum Marks: 50

Roll No: _____

Name: _____

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

We are interested in translating the following code fragment into three-address code. This code fragment reverses the elements at even indices (between and including the specified positions `left` and `right`) of a one-dimensional array `arr[]`. Assume that all variables in the code snippet (that is, `start`, `end`, `left`, `right`, `temp`, `idx`, and the elements of the array `arr[]`) are declared as integers.

```
start = ( left % 2 == 0 ) ? left : left + 1 ;
end = ( right % 2 == 0 ) ? right : right - 1 ;
for ( idx = start ; idx < end ; idx += 2 ) {
  temp = arr[idx]; arr[idx] = arr[end]; arr[end] = temp ;  // swap arr[idx] and arr[end]
   end -= 2 ;
}
```

For the translation of the above code, write the semantic actions for a set of productions given below. Assume that `P` (standing for Program) is the start symbol of the grammar. Each of `P` and the other non-terminals `E` (derives arithmetic Expressions), `A` (derives Assignments to variables or array elements), `B` (derives Boolean expressions) and `S` (derives Statements) has a synthesized attribute `code` to store the three-address instructions as a string, obtained by translating the parse tree generated by their respective instances. Arithmetic expressions (`E`) have a synthesized attribute `addr` storing a reference to the address of the expression. The labels in `B.code` and `S.code` are managed using inherited attributes as follows. To a boolean expression `B`, associate two inherited labels `B.true` (the label to which control flows if `B` evaluates to true) and `B.false` (the label to which control flows if `B` evaluates to false). Furthermore, to each statement `S`, associate an inherited attribute `S.next` denoting a label for the instruction immediately after `S`. The arithmetic operators are denoted by `aop` = {`+,-,*,/,%`}, and the boolean operators by `bop` = {`<,<=,>,>=,==,!=`}. Use the following functions in your semantic rules.

- `newlabel()` to create a new label,
- `label(L)` to attach the label "`L:`" to the next three-address instruction to be generated,
- `newtemp()` to create a new temporary, and
- `gen()` to generate a three-address code string.

Here, each element of `aop` and `bop` as well as `for`, `[`, `]`, `{`, `}`, `(`, `)`, `?`, `:`, `=`, `+=`, `-=`, and `;` are lexical tokens (including keywords) carrying the conventional meanings as in the C language. Moreover, `id` and `num` respectively denote an identifier (a variable name) and a constant integer, whose `lexval` can be obtained from the lexical analyzer. While referencing an array variable using `id[]`, the corresponding `id.type` and `id.type.width` refer to the basic type and the width of each array element, whereas `id.base` indicates the base address of the array.

(1) Clearly write the semantic rules (in the right column) corresponding to each production (shown in the left column), in the table below. Note that this is only a subset of the entire grammar covered in the class, because your current task is to translate only the above code segment. Do not use any *fall-through* code optimization in this part (that is, use only `if`-conditioned jumps). Do not use any backpatching (because labels are passed via inherited attributes). For some of the productions, the semantic rules are already supplied. **[24]**

Productions	Semantic Rules

`P` ⟶ `S`	`S.next = P.next`
	`P.code = S.code + label ( S.next )`
	[+ denotes concatenation of strings]

`P` ⟶ P_1 `S`	P_1`.next = newlabel ()`
	`S.next = P.next`
	`P.code =` P_1`.code + label (` P_1`.next ) + S.code`

`S` ⟶ `A ;`	`S.code = A.code`

Productions	Semantic Rules
A ⟶ id = E **[1 mark]**	A.code = E.code + gen (id.lexval '=' E.addr)
A ⟶ id += E **[2 marks]**	T = newtemp () A.code = E.code + gen (T '=' id.lexval '+' E.addr) + gen(id.lexval '=' T)
A ⟶ id -= E **[2 marks]**	T = newtemp () A.code = E.code + gen (T '=' id.lexval '-' E.addr) + gen(id.lexval '=' T)
A ⟶ id [E_1] = E_2 **[3 marks]**	T = newtemp () A.code = E_1.code + E_2.code + gen (T '=' E_1.addr '*' id.type.width) + gen(id.base '[' T ']' '=' E_2.addr)
E ⟶ id	E.addr = id.lexval E.code = ''
E ⟶ num	E.addr = num.lexval E.code = ''
E ⟶ id [E_1] **[3 marks]**	T = newtemp () E.addr = newtemp () E.code = E_1.code + gen (T '=' E_1.addr '*' id.type.width) + gen (E.addr '=' id.base '[' T ']')

Productions	Semantic Rules
$E \longrightarrow E_1$ aop E_2 **[2 marks]**	E.addr = newtemp () E.code = E_1.code + E_2.code + gen (E.addr '=' E_1.addr 'aop' E_2.addr)
$S \longrightarrow$ id = (B) ? E_1 : E_2 ; **[4 marks]**	B.true = newlabel () B.false = newlabel () S.code = B.code + label (B.true) + E_1.code + gen (id.lexval '=' E_1.addr) + gen ('goto' S.next) + label (B.false) + E_2.code + gen (id.lexval '=' E_2.addr)
$S \longrightarrow$ for (A_1 ; B ; A_2) { P } **[5 marks]**	CONDITION = newlabel () B.true = newlabel () B.false = S.next UPDATE = newlabel () P.next = UPDATE S.code = A_1.code + label (CONDITION) + B.code + label (UPDATE) + A_2.code + gen ('goto' CONDITION) + label (B.true) + P.code + gen ('goto' UPDATE)
$B \longrightarrow E_1$ bop E_2 **[2 marks]**	B.code = E_1.code + E_2.code + gen ('if' E_1.addr 'bop' E_2.addr 'goto' B.true) + gen (‘goto’ B.false)

(2) Now, translate the given code fragment (reproduced below for your convenience) to three-address code following the semantic rules that you presented in Part (1). You should not bother about the declarations of variables and of one-dimensional arrays. However, assume that the width of an integer is 4 bytes. **[14]**

```
start = ( left % 2 == 0 ) ? left : left + 1 ;
end = ( right % 2 == 0 ) ? right : right - 1 ;
for ( idx = start ; idx < end ; idx += 2 ) {
   temp = arr[idx]; arr[idx] = arr[end]; arr[end] = temp ;  // swap arr[idx] and arr[end]
   end -= 2 ;
}
```

```
    // Initialize 'start'
            t1 = left % 2
            if t1 == 0 goto LABEL1
            goto LABEL2

   LABEL1:  start = left
            goto LABEL3

   LABEL2:  t2 = left + 1
            start = t2


    // Initialize 'end'
   LABEL3:  t3 = right % 2
            if t3 == 0 goto LABEL4
            goto LABEL5

   LABEL4:  end = right
            goto LABEL6

   LABEL5:  t4 = right - 1
            end = t4


    // Loop Initialization
   LABEL6:  idx = start


    // Loop Condition Check
CONDITION:  if idx < end goto LABEL7
            goto LABEL8


    // Loop Update
   UPDATE:  t5 = idx + 2         // idx increment
            idx = t5

            goto CONDITION
```

```
 // Loop Body
LABEL7:  t6 = idx * 4          // Address offset for arr[idx]
         temp = arr[t6]        // temp = arr[idx]

         t7 = end * 4          // Address offset for arr[end]
         t8 = arr[t7]          // Load value of arr[end]
         arr[t6] = t8          // arr[idx] = arr[end]

         arr[t7] = temp        // arr[end] = temp

         t9 = end - 2          // end decrement
         end = t9

         goto UPDATE


LABEL8:
 // Code continues after the loop
```

(3) Eliminate redundant `goto` instructions that you generated in the three-address code of Part (2) using the semantic rules of Part (1). Only rewrite the semantic rules for the following <u>two</u> productions, that incorporate the "*fall-through*" optimization technique in order to eliminate redundant `goto` instructions. No need to write the semantic rules for the other productions where there is no change. **[6]**

Productions	Semantic Rules
$S \longrightarrow id = (B) ? E_1 : E_2 ;$ **[3 marks]**	B.true = fall B.false = newlabel () S.code = B.code + E_1.code + gen (id.lexval '=' E_1.addr) + gen ('goto' S.next) + label (B.false) + E_2.code + gen (id.lexval '=' E_2.addr)
$B \longrightarrow E_1$ bop E_2 **[3 marks]**	B.code = if B.true ≠ fall and B.false ≠ fall then gen ('if' E_1.addr 'bop' E_2.addr 'goto' B.true) + gen ('goto' B.false) else if B.true ≠ fall then gen ('if' E_1.addr 'bop' E_2.addr 'goto' B.true) else if B.false ≠ fall then gen ('ifFalse' E_1.addr 'bop' E_2.addr 'goto' B.false) else ''

(4) Again consider translating the given code fragment into three-address code with redundant `goto` instructions eliminated following the semantic rules that you formulated in Part (3). Only write the three-address code of the first two statements (as the changes would occur in that position only). Also write the three-address code for the initialization (`idx = start`) of the `for` loop (because this instruction will have a jump label). **[6]**

```
    // Initialize 'start'
            t1 = left % 2
            ifFalse t1 == 0 goto LABEL1

            start = left
            goto LABEL2

   LABEL1:  t2 = left + 1
            start = t2


    // Initialize 'end'
   LABEL2:  t3 = right % 2
            ifFalse t3 == 0 goto LABEL3

            end = right
            goto LABEL4

   LABEL3:  t4 = right - 1
            end = t4


    // Loop Initialization
   LABEL4:  idx = start


/* --- Rest of the code will remain similar to part (2) --- */
```