1. Bottom Up LALR(1) Parsing **[24 Marks]**

Consider the CUP grammar in the **Appendix For Question 1**. Note that some rules are left recursive, while other rules are right recursive. Also note the rule for “ConcatExpr” that expands to empty.

(a) Using the information provided in the appendix, perform a shift-reduce LALR(1) parse of the input

```
a[bc]|d
```

Assume “`a` ”, “`b` ”, “`c` ”, “`d`”, match `CHAR`, and “`[`”, “`]`” and “`|`” match `LEFTSQ`, `RIGHTSQ` and `OR`, respectively.

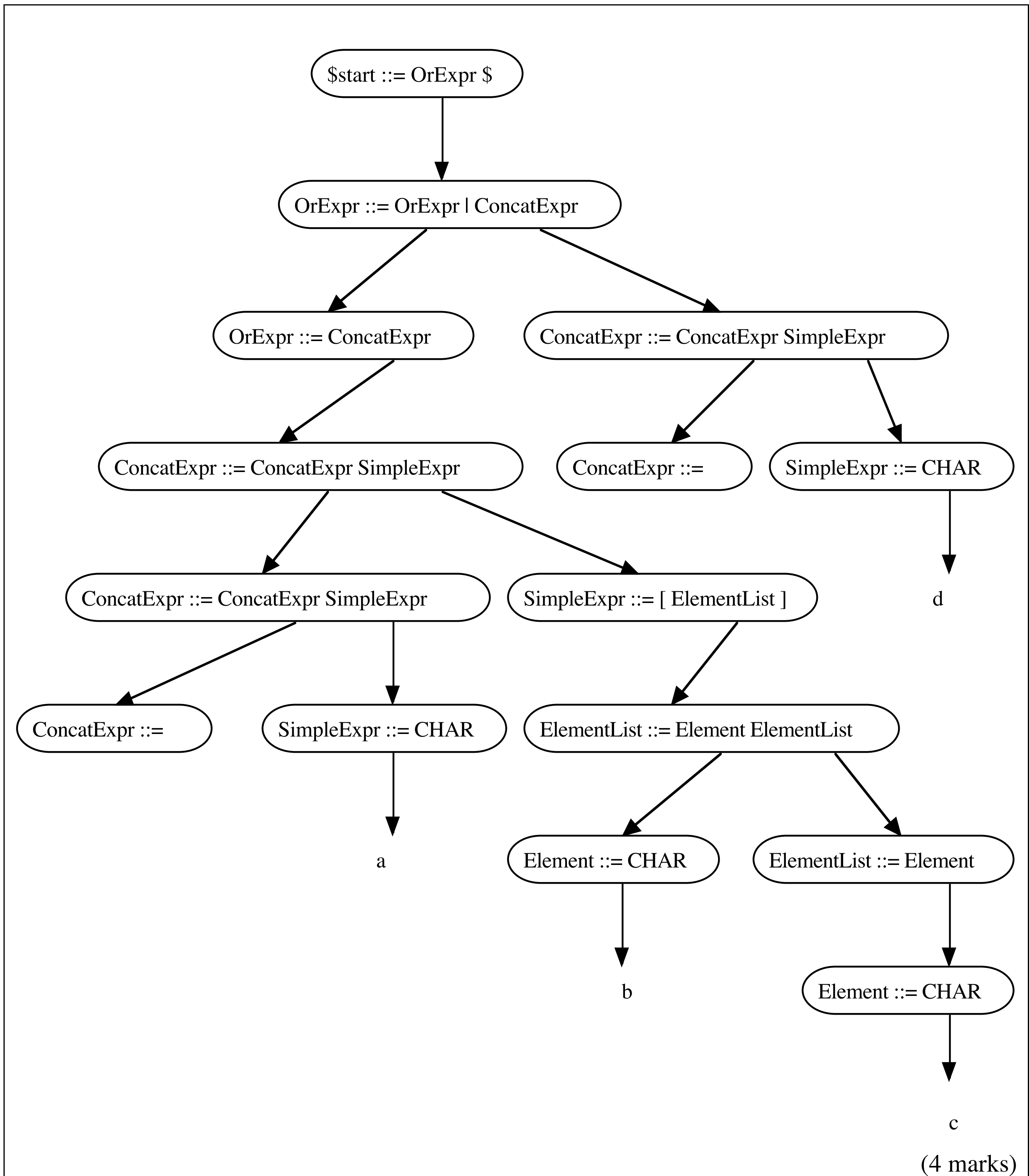
Stack

<table>
<tr><td>$0</td><td></td><td></td><td></td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td></td><td></td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td>CH 5</td><td></td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td>SE 4</td><td></td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td></td><td></td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td>[6</td><td></td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td>[6</td><td>CH 9</td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td>[6</td><td>E 7</td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td>[6</td><td>E 7</td><td>CH 9</td></tr>
<tr><td>$0</td><td>CE 2</td><td>[6</td><td>E 7</td><td>E 7</td></tr>
<tr><td>$0</td><td>CE 2</td><td>[6</td><td>E 7</td><td>EL 13</td></tr>
<tr><td>$0</td><td>CE 2</td><td>[6</td><td>EL 8</td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td>[6</td><td>EL 8</td><td>] 12</td></tr>
<tr><td>$0</td><td>CE 2</td><td>SE 4</td><td></td><td></td></tr>
<tr><td>$0</td><td>CE 2</td><td></td><td></td><td></td></tr>
<tr><td>$0</td><td>OE 1</td><td></td><td></td><td></td></tr>
<tr><td>$0</td><td>OE 1</td><td>| 16</td><td></td><td></td></tr>
<tr><td>$0</td><td>OE 1</td><td>| 16</td><td>CE 17</td><td></td></tr>
<tr><td>$0</td><td>OE 1</td><td>| 16</td><td>CE 17</td><td>CH 5</td></tr>
<tr><td>$0</td><td>OE 1</td><td>| 16</td><td>CE 17</td><td>SE 4</td></tr>
<tr><td>$0</td><td>OE 1</td><td>| 16</td><td>CE 17</td><td></td></tr>
<tr><td>$0</td><td>OE 1</td><td></td><td></td><td></td></tr>
<tr><td>$0</td><td>OE 1</td><td>$18</td><td></td><td></td></tr>
<tr><td>$0</td><td>$start -1</td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><th>Token</th><th colspan="2">Action</th></tr>
<tr><td>CHAR a</td><td>Reduce</td><td>CE ::=</td></tr>
<tr><td></td><td>Shift</td><td>CHAR 5</td></tr>
<tr><td>[</td><td>Reduce</td><td>SE ::= CHAR</td></tr>
<tr><td></td><td>Reduce</td><td>CE ::= CE SE</td></tr>
<tr><td></td><td>Shift</td><td>[6</td></tr>
<tr><td>CHAR b</td><td>Shift</td><td>CHAR 9</td></tr>
<tr><td>CHAR c</td><td>Reduce</td><td>E ::= CHAR</td></tr>
<tr><td></td><td>Shift</td><td>CHAR 9</td></tr>
<tr><td>]</td><td>Reduce</td><td>E ::= CHAR</td></tr>
<tr><td></td><td>Reduce</td><td>EL ::= E</td></tr>
<tr><td></td><td>Reduce</td><td>EL ::= E EL</td></tr>
<tr><td></td><td>Shift</td><td>] 12</td></tr>
<tr><td>|</td><td>Reduce</td><td>SE ::= [EL]</td></tr>
<tr><td></td><td>Reduce</td><td>CE ::= CE SE</td></tr>
<tr><td></td><td>Reduce</td><td>OE ::= CE</td></tr>
<tr><td></td><td>Shift</td><td>OR 16</td></tr>
<tr><td>CHAR d</td><td>Reduce</td><td>CE ::=</td></tr>
<tr><td></td><td>Shift</td><td>CHAR 5</td></tr>
<tr><td>$</td><td>Reduce</td><td>SE ::= CHAR</td></tr>
<tr><td></td><td>Reduce</td><td>CE ::= CE SE</td></tr>
<tr><td></td><td>Reduce</td><td>OE ::= OE OR CE</td></tr>
<tr><td></td><td>Shift</td><td>$18</td></tr>
<tr><td></td><td>Reduce</td><td>$START ::= OE $</td></tr>
<tr><td></td><td>Accept</td><td></td></tr>
</table>

(14 marks)

(b) Draw the parse tree corresponding to the grammar rules used to parse this input



(4 marks)

(c) State 2 is

```
lalr_state [2]: {
     [SimpleExpr ::= (*) LEFTSQ ElementList RIGHTSQ ,
          {EOF OR LEFT RIGHT LEFTSQ CHAR }]
     [OrExpr ::= ConcatExpr (*) , {EOF OR RIGHT }]
     [SimpleExpr ::= (*) LEFT OrExpr RIGHT ,
          {EOF OR LEFT RIGHT LEFTSQ CHAR }]
     [ConcatExpr ::= ConcatExpr (*) SimpleExpr ,
          {EOF OR LEFT RIGHT LEFTSQ CHAR }]
     [SimpleExpr ::= (*) CHAR ,
          {EOF OR LEFT RIGHT LEFTSQ CHAR }]
}
```

Derive the sets of items of State 6 = GoTo(State 2, LEFTSQ). Remember to take the closure to get the full set of items, and remember to compute the follow symbols.

lalr_state [6]: {

[SimpleExpr ::= LEFTSQ (*) ElementList RIGHTSQ , {EOF OR LEFT RIGHT LEFTSQ CHAR }]

[ElementList ::= (*) Element ElementList , {RIGHTSQ }]

[ElementList ::= (*) Element , {RIGHTSQ }]

[Element ::= (*) CHAR , {RIGHTSQ CHAR }]

[Element ::= (*) CHAR MINUS CHAR , {RIGHTSQ CHAR }]

}

(6 marks)

2. Write a grammar for variable declarations **[14 Marks]**

A (simplified) Java variable declaration is composed of a type, followed by a "," separated list of declarators, then a ";". A type is either a primitive type ("int", "char", etc), identifier (e.g., "String"), or an array type (a type followed by "[]"s, e.g., "int[]", "String[][]"). A declarator can be either uninitialised (just an identifier) or initialised ("identifier = initialiser"). An initialiser can be either an expression, or an array initialiser. An array initialiser is an optional "," separated list of initialisers enclosed in "{...}"s.

For example, the following represent variable declarations:

```
int a, b, d = 3, e = 4, f;
int[] g = { 1, 2, 3 }, h = new int[ 3 ], i;
int[][] j = { { 1, 2, 3 }, new int[ 3 ], { 4, 5, 6 } };
String[]  k = { "yes", "no", "maybe" };
```

Write a grammar for variable declarations **in general**. You do not have to define what a primitive type or expression is (and note that array constructors such as "new int[3]" are expressions). Unlike Java, you should not allow modifiers, "[]"s after the identifier being declared, or an additional "," after the last initialiser in an array initialiser. You do not have to write actions.

VariableDecl ::= Type DeclrList ";"

Type ::= PrimitiveType | IDENT | Type "[" "]"

DeclrList ::= Declr | DeclrList "," Declr

Declr ::= IDENT | IDENT "=" Initialiser

Initialiser ::= Expr | "{" InitialiserListOpt "}"

InitialiserListOpt ::= /* empty */ | InitialiserList

InitialiserList ::= Initialiser | InitialiserList "," Initialiser

(14 marks)

3. Interpretation **[15 Marks]**

Consider the INTERP9/BHP language of Assignment 2.

(a) Describe the general structure of a runtime environment.

A hash table containing a mapping from names to variables. A variable then maps to a value.

(1 mark)

(b) Explain how var parameters can be implemented using this representation.

two different hash tables can map a name to the same variable.

(2 marks)

(c) Explain how a runtime environment can be used to represent an array. Give an example of a program with

- A function that takes a list of values as parameters, and creates an array with those values as elements.
- A function that takes an array as a parameter, and loops printing the values of the elements.
- Global code to invoke the functions to create the array and print out its elements.

When invoke a function, creates a local environment that maps the parameter positions to actual parameters, and # to number of parameters. A function can have more actual parameters than formal parameters, so it is possible to pass an arbitrary number of parameters. Moreover , it is possible return the local environment, by returning $this, and later use this environment as an arrray.

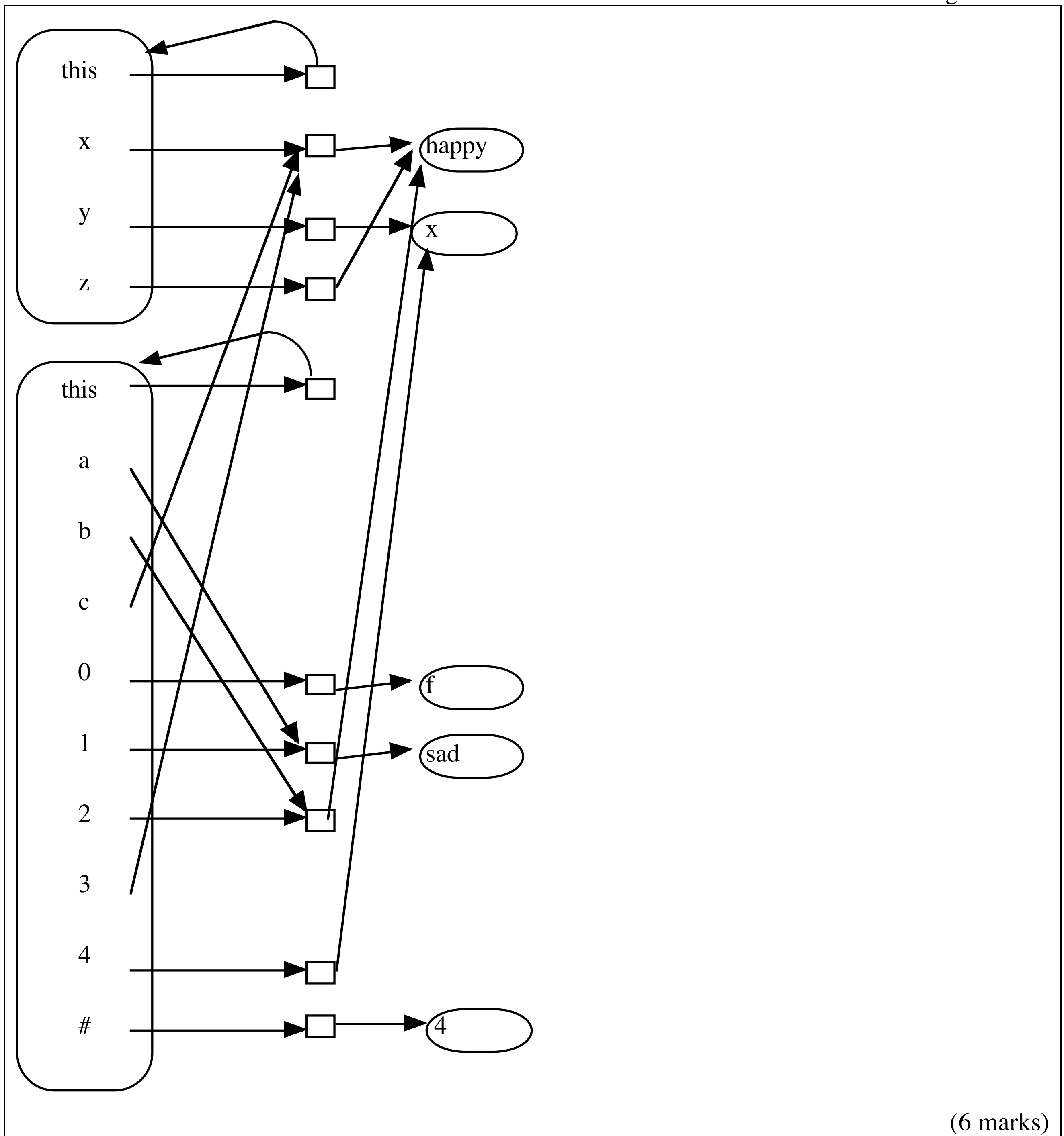
```
function array()
     begin
          return $this;
     end

function printArray( $a )
     begin
          for
               $i = 1;
          do
               break $i > $a[ # ];
               print( "$a[ $i ] " );
          step
               $i++;
          end
          print( "\n" );
     end
$a = array( 1, 2, 4, 8, 16, 32 );
printArray( $a );
```

(6 marks)

(d) Draw a diagram showing the runtime environments that would be generated by the following program, at the time when the function f is being invoked.

```
function f( $a, $b, &$c )
     begin
          //   show at this point
     end
$x = happy;
$y = x;
$z = $x;
f( sad, $x, &$x, $y ); // Note extra actual parameter
```



(6 marks)

4. Show the run time stack for an INTERP7 program [15 Marks]

Use the program written in the Chapter 8 INTERP7 language in the **Appendix For Question 4**.

Complete the drawing of the data structure built for the global variables "`source1`", "`source2`" and "`dest`".

Display the stack frames (activation records) for all methods in the process of being invoked when the maximum level of nesting of method invocations occurs when the statement

```
merge( 0, source1, source2, dest );
```

on line 47 is invoked, and the process is almost ready to return. At this stage the process should be executing the method "`merge`" at line 33.

Indicate the appropriate values for each stack frame (activation record) you draw. The line numbers on the left-hand side of the program should be used to represent the return address. Draw appropriate arrows for the var parameters, and pointers to objects. For var parameters, make sure you indicate the exact field pointed to in an object very clearly. Represent `List` nodes as shown in the sample entries.

Also indicate the output generated by the complete execution of the program.

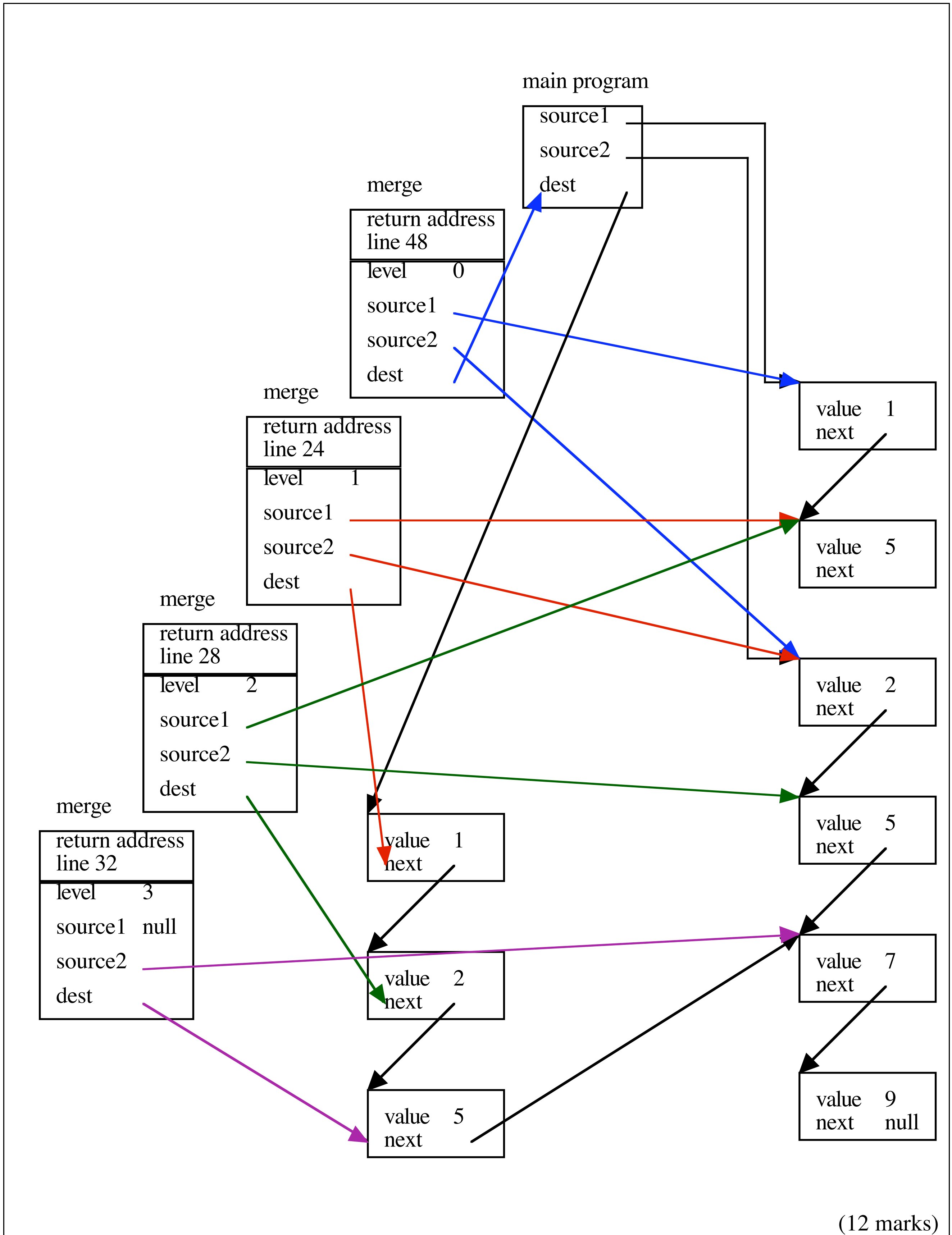
Output generated:

{ 1, 5 }

{ 2, 5, 7, 9 }

{ 1, 2, 5, 7, 9 }

(3 marks)

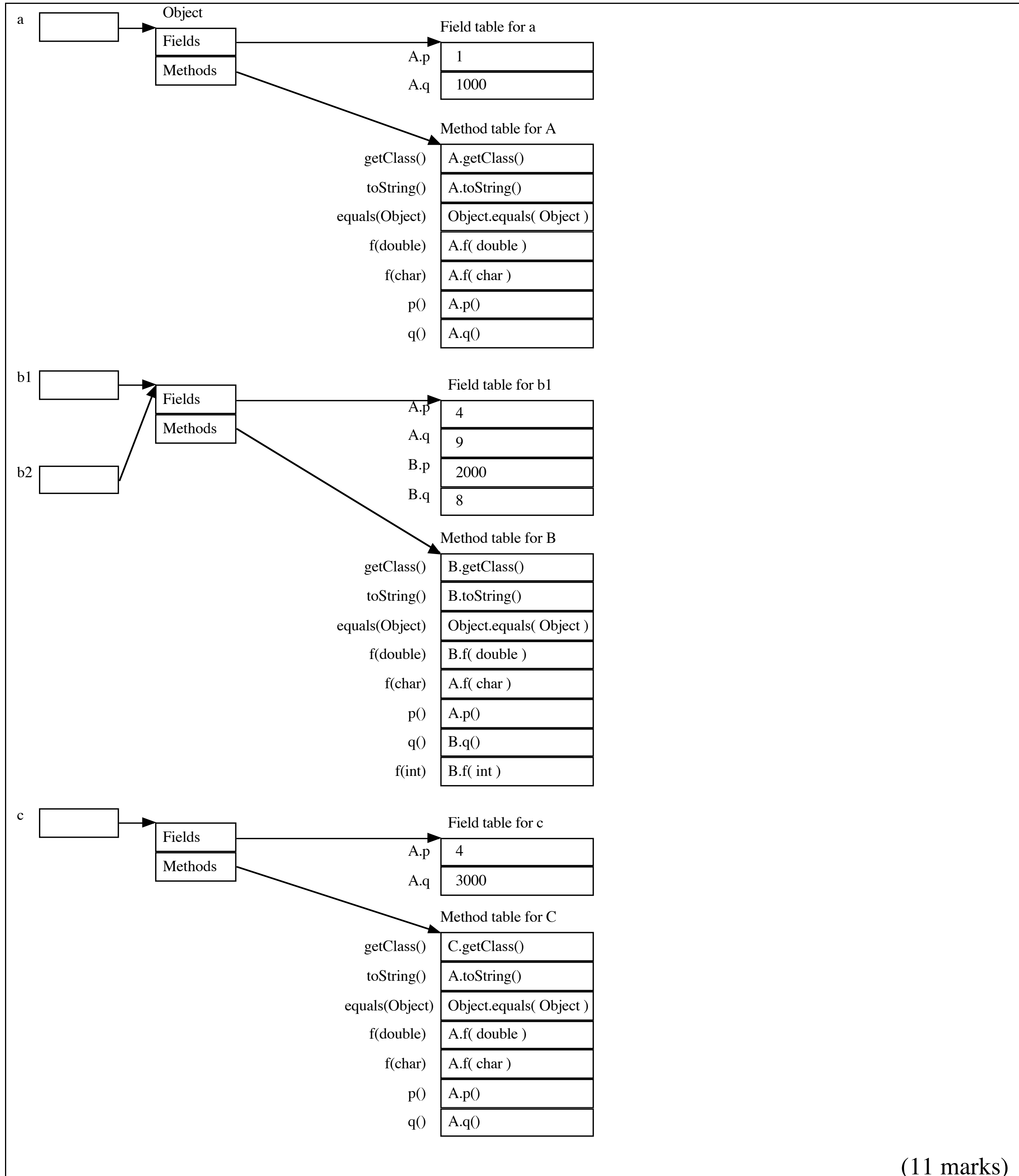


(12 marks)

5. Implementation of object oriented languages **[16 Marks]**

Use the Java program in the **Appendix For Question 5**.

(a) Draw a diagram showing the data structures (object, field table, method table, etc) created for the variables `a`, `b1`, `b2`, `c`, within the method `Main.main`. Shared data structures should be drawn only once.



(11 marks)

(b) Indicate the output generated by the method Main.main.

```
A            a.p  = 1       a.q  = 1000
B            b1.p = 2000    b1.q = 8
B            b2.p = 4       b2.q = 9
A            c.p  = 4       c.q  = 3000


             b1.p() = 4     b1.q() = 8
             b2.p() = 4     b2.q() = 8


b1.f( 'A' ) = A.f( 'A' )
b2.f( 'A' ) = A.f( 'A' )


b1.f( 65 ) = B.f( 65 )
b2.f( 65 ) = B.f( 65.0 )
```

(5 marks)

6. Code generation **[16 Marks]**

Consider the program written in the assignment 3 OBJECT7 language in the **Appendix For Question 6**.

(a) Explain why the lines

```
y = a;
z = list[ i ];
```

are legal in the OBJECT7 language.

references to []type are converted to ^type, and references to a class are converted to ^class.

so the 3 assignments correspond to assignment of values of type ^int, ^List.

(2 marks)

(b) Indicate the code likely to be generated to implement the following statements

Notes:

An appendix is provided with common Alpha instructions.

Addresses are represented using 8 bytes on the Alpha.

Each statement should be translated independently, and not make use of values left in registers from previous declarations or statements.

```
y = a;
```

```
ldiq $t0, a;
ldiq $t1, y;
stq $t0, 0($t1);
```

(1 mark)

```
y = p;
```

```
ldiq $t0, p;
ldq $t0, 0($t0);
ldiq $t1, y;
stq $t0, 0($t1);
```

(1 mark)

```
x = p^;
```

```
ldiq $t0, p;
ldq $t0, 0($t0);
ldq $t0, 0($t0);
ldiq $t1, x;
stq $t0, 0($t1);
```

(2 marks)

```
x = a[ i ];
```

```
ldiq $t0, a;
ldiq $t1, i;
ldq $t1, 0($t1);
s8addq $t1, $t0, $t0;
ldq $t0, 0($t0);
ldiq $t1, x;
stq $t0, 0($t1);
```

(2 marks)

```
y = &a[ i ];
```

```
ldiq $t1, a;
ldiq $t2, i;
ldq $t2, 0($t2);
s8addq $t2, $t1, $t1;
ldiq $t0, y;
stq $t0, 0($t1);
```

(2 marks)

```
y = b[ i ];
```

```
ldiq $t0, b;
ldiq $t1, i;
ldq $t1, 0($t1);
mulq $t1, 40, $t1;
addq $t0, $t1, $t0;
ldiq $t1, y;
stq $t0, 0($t1);
```

(2 marks)

```
z = list[ i ];
```

```
ldiq $t0, list;
ldiq $t1, i;
ldq $t1, 0($t1);
mulq $t1, 16, $t1;
addq $t0, $t1, $t0;
ldiq $t1, z;
stq $t0, 0($t1);
```

(2 marks)

```
z++;
```

```
ldiq $t1, z;
ldq $t0, 0($t1);
addq $t0, 16, $t2;
stq $t2, 0($t1);
```

(2 marks)

__________________End of Questions________________