# THE UNIVERSITY OF AUCKLAND

**First Semester, 2001**
**City Campus**
**Computer Science**
**Language Implementation**
**(Time allowed TWO hours)**

| |
|---|
| FAMILY NAME: |
| PERSONAL NAMES: |
| STUDENT ID NUMBER: |
| SIGNATURE: |
| LOGIN NAME: |

This Examination is out of 70 Marks. Attempt **ALL** questions. Write your answers in the spaces provided in this booklet.

| | | |
|---|---|---|
| 1 | | 10 |
| 2 | | 5 |
| 3 | | 15 |
| 4 | | 15 |
| 5 | | 15 |
| 6 | | 10 |
| Total | | 70 |

Continued ...

Print Name:_____

**1. Bottom Up LALR(1) Parsing** **[10 Marks]**

Consider the CUP grammar:

```
terminal
     LEFT, RIGHT,     COMMA,     ASSIGN;
/*   (          )          ,          =    */

terminal
     String IDENT;   /* Identifier */

non terminal
     Program, Header, IdentList, Expr, ExprList;

start with Program;

Program::=
          Header ASSIGN Expr
     ;

Header::=
          IDENT LEFT IdentList RIGHT
     ;

IdentList::=
          IDENT
     |
          IdentList COMMA IDENT
     ;

Expr::=
          IDENT
     |
          IDENT LEFT ExprList RIGHT
     ;

ExprList::=
          Expr
     |
          ExprList COMMA Expr
     ;
```

Continued ...

Print Name:_____

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

```
f ( a , b ) = g ( a )
```

| Stack | | | | | | | | Token | Action |
|---|---|---|---|---|---|---|---|---|---|
| $ 0 | | | | | | | | Id f | shift 2 |
| $ 0 | Id 2 | | | | | | | ( | shift 13 |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | | | | | | | | | |
| $ 0 | Pr 1 | | | | | | | | shift 19 |
| $ 0 | Pr 1 | $ 19 | | | | | | | reduce $S' -> Pr $ |
| $ 0 | $S' -1 | | | | | | | | accept |

Continued ...

Print Name:_____

**2. JLex** **[5 Marks]**

Specify JLex definitions to represent regular expressions for real numbers.

(a) The following should represent real numbers: `0.0`, `0.00`, `123.456`, `0123.4`, `12e6`, `12e+6`, `12e-6`, `123.456e-23`, `12E14`, etc.

(b) The following should **not** represent real numbers, and should **not** be matched: `0`, `123`, `123.`, `.123`, `-123.0`, `+123.0`, `1,024.5`, `e6`, `e-6`, `123f`, `123d`, `123.45f`, etc.

```
float =
```

Print Name:_____

**3. Write a grammar definition for sets of unsigned integers [15 Marks]**

Suppose we have a mathematical notation for sets of unsigned integers.

We can enumerate unsigned integers, as in `{}`, `{ 1 }`, `{ 1, 2, 4 }`, etc, to form basic sets.

We can also include ranges of unsigned integers within a basic set, by using a "`..`" notation, as in `{ 1 .. 10, 20 .. 30, 40 .. }`, etc.

The notation "`40 ..`" means the infinite range of integers from `40` up to (but excluding) infinity.

We can combine these notations, as in `{ 1, 2, 4, 10 .. 20, 31, 40 .. }`.

An infinite range may only appear as the last component in the list.

We can also combine sets using parentheses `( … )` and the left associative binary infix operators ∪ and ∩, representing union and intersection, as in

`{ 1, 2 }` ∪ `{ 3 }` ∪ `{ 20 .. 40 }` ∩ `( { 25 }` ∪ `{ 30 .. 50 } )`.

Union has a lower precedence than intersection.

Write a grammar definition for sets, to match the above notation. You do not have to write any actions.

Continued ...

Print Name:_____

Continued ...

Print Name:_____

**4. Show the run time stack. [15 Marks]**

Below is a program in the EXPREVAL6 "b" language, which allows nesting of function declarations. Show the run-time stack when the maximum level of nesting of function invocations occurs for the first time, and the process is almost ready to return. Remember the stack grows "up" towards low memory. Enter the appropriate values for each stack frame (activation record) you draw. Assume that all variables are implicitly initialised to zero or null. The numbers on the left-hand side of the program represent the "instruction addresses". Draw appropriate arrows for the var parameters, and pointers to structured objects. The stack information has already been filled in for the main program, and the first stack frame. You may label addresses, and refer to them, rather than drawing arrows, if you need to, to make your answer clearer.

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

```
 1   program
 2
 3   type Expr = struct(
 4        kind, priority : int;
 5        operator : string;
 6        left, right : Expr; );    // A structured object, like a class
 7
 8   var IDENT, BIN : int;
 9
10   funct toString( level : int; expr : Expr; var result : string; ) void
11
12        funct parenth(
13             level, priority: int; expr : Expr; var result : string; ) void
14             var result1 : string;
15             begin
16                  toString( level + 1, expr, result1 );
17                  if expr.priority < priority then
18                       result = "(" + result1 + ")";
19                  else
20                       result = result1;
21                  end
22             end
23
24        var left, right : string;
25        begin
26             if expr.kind == IDENT then
27                  result = expr.operator;
28             elif expr.kind == BIN then
29                  parenth( level + 1, expr.priority, expr.left, left );
30                  parenth( level + 1, expr.priority+1, expr.right, right );
31                  result = left + expr.operator + right;
32             end
33        end
34
35   var expr1, expr2, expr3, expr4, expr : Expr;
36   var result : string;
37   begin
38        IDENT = 1;
39        BIN = 2;
40        expr1 = new Expr{ IDENT, 3, "a", null, null };
41        expr2 = new Expr{ IDENT, 3, "b", null, null };
42        expr3 = new Expr{ IDENT, 3, "c", null, null };
43        expr4 = new Expr{ BIN, 1, "+", expr1, expr2 };
44        expr = new Expr{ BIN, 2, "*", expr4, expr3 };
```

Continued ...

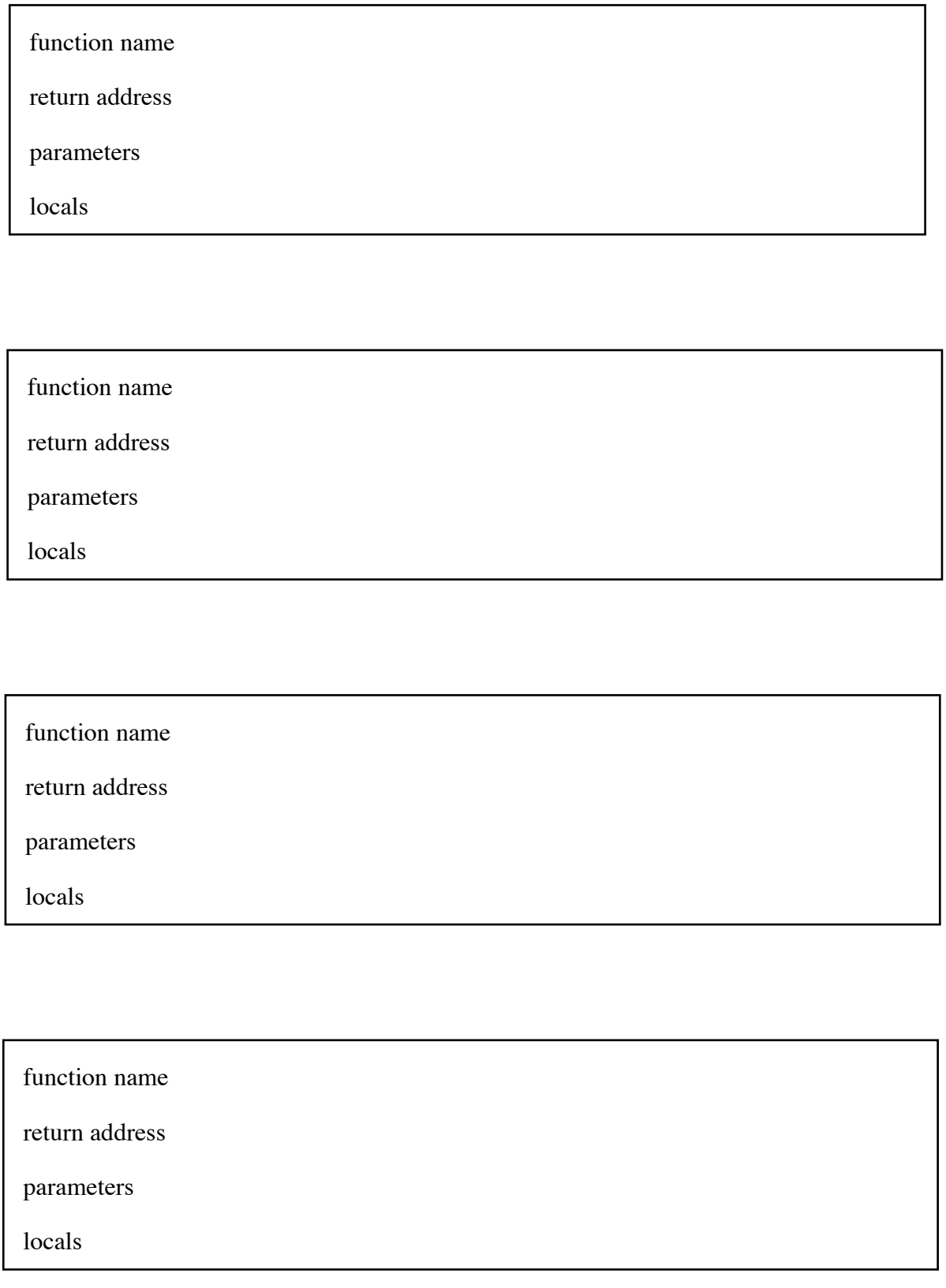
Print Name:_____

```
45          toString( 1, expr, result );
46          println( result );
47    end.
```

Continued ...

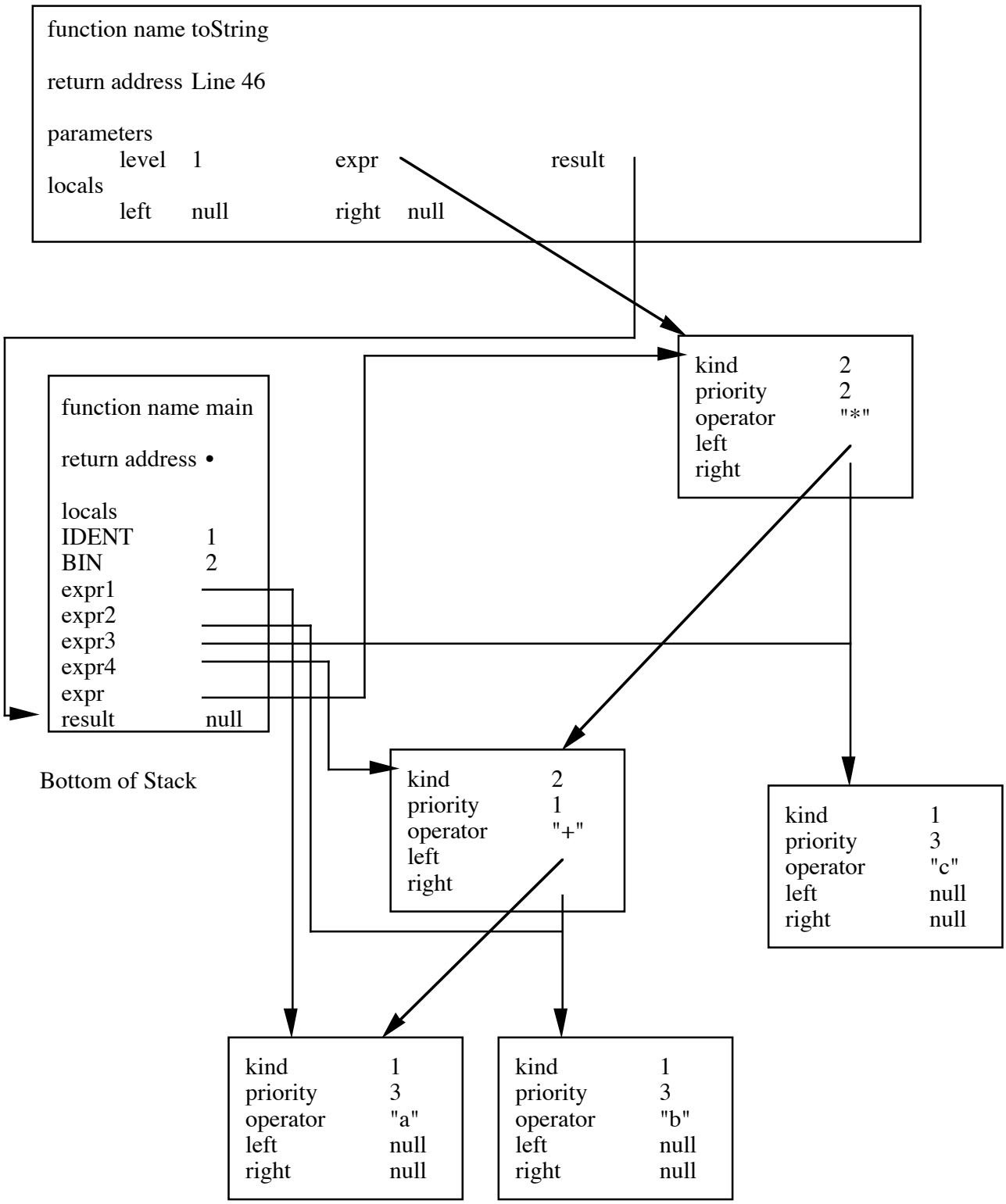
Print Name:_____

Top of Stack




Continued ...

Print Name:_____




Output generated:

Continued ...

Print Name:_____

**5. Compile time checking and code generation for switch statements [15**

Describe appropriate algorithms and data structures for performing compile time checking and code generation for switch statements.

Indicate the abstract syntax tree, data structures and assembly language likely to be generated by your algorithm, for the switch statement

```
switch x of
     case 1, -1:
          even = false;
     case 0, 2, -2:
          even = true;
     default:
          outOfRange = true;
     end
```

It is essential that you provide a precise and specific implementation, using a combination of pseudocode and English, rather than a vague discussion. You should specify the precise attributes passed through the tree, whether they are inherited, synthesized, or threaded, and their values in the above specific example.

Continued ...

Print Name:_____

Continued ...

Print Name:_____

Continued ...

Print Name:_____

**6. Implementation of object oriented languages [10 Marks]**

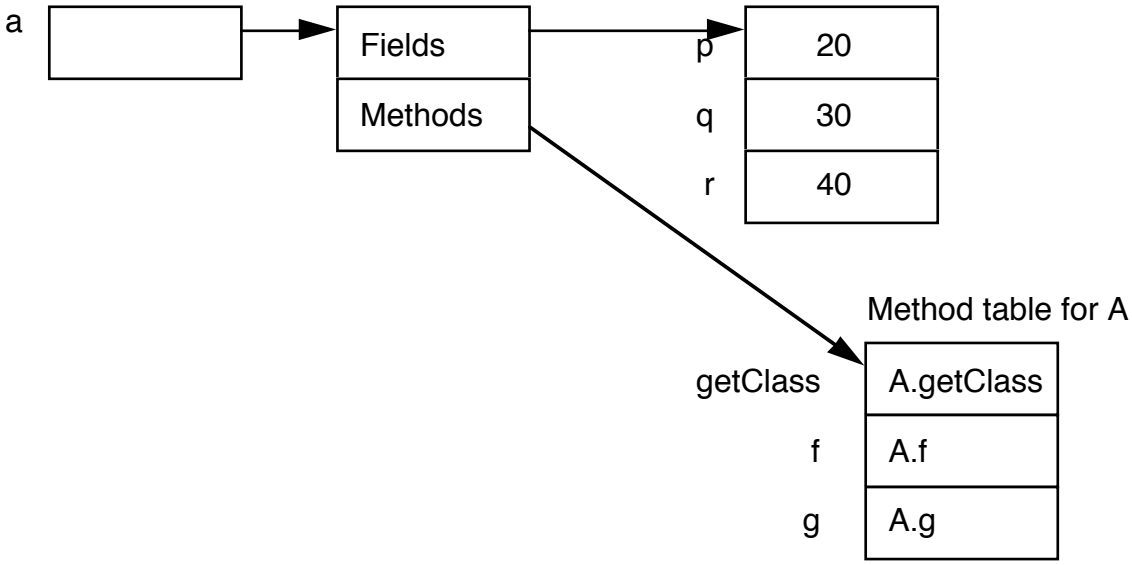
Suppose we have the Java program

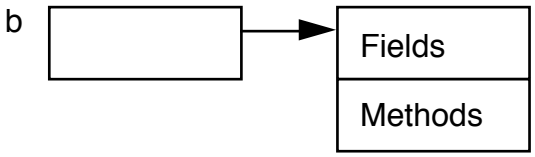
```
class A {
     int p, q, r;
     A( int p, int q, int r ) { this.p = p; this.q = q; this.r = r; }
     String f() { return "A.f"; }
     String g() { return "A.g"; }
     static String h() { return "A.h"; }
     }

class B extends A {
     int q, r, s;
     B( int q, int r, int s, int t, int u, int v ) {
          super( t, u, v );
          this.q = q; this.r = r; this.s = s;
          }
     String e() { return "B.e"; }
     String f() { return "B.f"; }
     static String h() { return "B.h"; }
     }

class Main {
     public static void main( String[] args ) {
          A a = new A( 20, 30, 40 );
          B b = new B( 50, 60, 70, 80, 90, 100 );
          A c = new B( 110, 120, 130, 140, 150, 160 );
          System.out.println( "a.p = " + a.p );
          System.out.println( "a.q = " + a.q );
          System.out.println( "a.r = " + a.r );
          System.out.println( "b.p = " + b.p );
          System.out.println( "b.q = " + b.q );
          System.out.println( "b.r = " + b.r );
          System.out.println( "b.s = " + b.s );
          System.out.println( "c.p = " + c.p );
          System.out.println( "c.q = " + c.q );
          System.out.println( "c.r = " + c.r );
          System.out.println( "c.f() = " + c.f() );
          System.out.println( "c.g() = " + c.g() );
          System.out.println( "c.h() = " + c.h() );
          }
     }
```

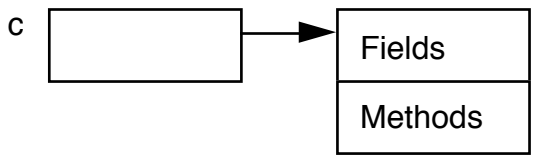
Continued ...

Print Name:_____

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




Continued ...

Print Name:_____

Indicate the output generated by the method Main.main.

```
a.p = 20
a.q = 30
a.r = 40
b.p =

b.q =

b.r =

b.s =

c.p =

c.q =

c.r =

c.f() =

c.g() =

c.h() =
```

_____End of Questions_____

Print Name:_____

This page left blank for formatting purposes, and any questions that overflow

End of Questions, Appendix follows

# Appendix

## Grammar Rules

```
8: ExprList ::= ExprList COMMA Expr
7: ExprList ::= Expr
6: Expr ::= IDENT LEFT ExprList RIGHT
5: Expr ::= IDENT
4: IdentList ::= IdentList COMMA IDENT
3: IdentList ::= IDENT
2: Header ::= IDENT LEFT IdentList RIGHT
1: Program ::= Header ASSIGN Expr
0: $START ::= Program EOF
```

## Action Table

```
From state #0
     IDENT:SHIFT(2)
From state #1
     EOF:SHIFT(19)
From state #2
     LEFT:SHIFT(13)
From state #3
     ASSIGN:SHIFT(4)
From state #4
     IDENT:SHIFT(6)
From state #5
     EOF:REDUCE(1)
From state #6
     EOF:REDUCE(5) LEFT:SHIFT(7) RIGHT:REDUCE(5)
     COMMA:REDUCE(5)
From state #7
     IDENT:SHIFT(6)
From state #8
     RIGHT:SHIFT(11) COMMA:SHIFT(10)
From state #9
     RIGHT:REDUCE(7) COMMA:REDUCE(7)
From state #10
     IDENT:SHIFT(6)
From state #11
     EOF:REDUCE(6) RIGHT:REDUCE(6) COMMA:REDUCE(6)
From state #12
     RIGHT:REDUCE(8) COMMA:REDUCE(8)
From state #13
     IDENT:SHIFT(14)
From state #14
     RIGHT:REDUCE(3) COMMA:REDUCE(3)
From state #15
     RIGHT:SHIFT(17) COMMA:SHIFT(16)
From state #16
     IDENT:SHIFT(18)
From state #17
     ASSIGN:REDUCE(2)
From state #18
     RIGHT:REDUCE(4) COMMA:REDUCE(4)
From state #19
     EOF:REDUCE(0)
```

Appendix Continued

## Reduce (Go To) Table

```
From state #0:
     Program:GOTO(1)
     Header:GOTO(3)
From state #1:
From state #2:
From state #3:
From state #4:
     Expr:GOTO(5)
From state #5:
From state #6:
From state #7:
     Expr:GOTO(9)
     ExprList:GOTO(8)
From state #8:
From state #9:
From state #10:
     Expr:GOTO(12)
From state #11:
From state #12:
From state #13:
     IdentList:GOTO(15)
From state #14:
From state #15:
From state #16:
From state #17:
From state #18:
From state #19:
                    _____________________End of Appendix_____________________
```

## Solution

**1. Bottom Up LALR(1) Parsing [10 Marks]**

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

```
f ( a , b ) = g ( a )
```

(10 marks)

| Stack | | | | | | | | Token | Action |
|---|---|---|---|---|---|---|---|---|---|
| $ 0 | | | | | | | | Id f | shift 2 |
| $ 0 | Id 2 | | | | | | | ( | shift 13 |
| $ 0 | Id 2 | ( 13 | | | | | | Id a | shift 14 |
| $ 0 | Id 2 | ( 13 | Id 14 | | | | | , | reduce IL -> Id |
| $ 0 | Id 2 | ( 13 | IL 15 | | | | | | shift 16 |
| $ 0 | Id 2 | ( 13 | IL 15 | , 16 | | | | Id b | shift 18 |
| $ 0 | Id 2 | ( 13 | IL 15 | , 16 | Id 18 | | | ) | reduce IL -> IL , Id |
| $ 0 | Id 2 | ( 13 | IL 15 | | | | | | shift 17 |
| $ 0 | Id 2 | ( 13 | IL 15 | ) 17 | | | | = | reduce Hd -> Id(IL) |
| $ 0 | Hd 3 | | | | | | | | shift 4 |
| $ 0 | Hd 3 | = 4 | | | | | | Id g | shift 6 |
| $ 0 | Hd 3 | = 4 | Id 6 | | | | | ( | shift 7 |
| $ 0 | Hd 3 | = 4 | Id 6 | ( 7 | | | | Id a | shift 6 |
| $ 0 | Hd 3 | = 4 | Id 6 | ( 7 | Id 6 | | | ) | reduce E->Id |
| $ 0 | Hd 3 | = 4 | Id 6 | ( 7 | E 9 | | | | reduce EL -> E |
| $ 0 | Hd 3 | = 4 | Id 6 | ( 7 | EL 8 | | | | shift 11 |
| $ 0 | Hd 3 | = 4 | Id 6 | ( 7 | EL 8 | ) 11 | | $ | reduce E->Id(EL) |
| $ 0 | Hd 3 | = 4 | E 5 | | | | | | reduce Pr->Hd=E |
| $ 0 | Pr 1 | | | | | | | | shift 19 |
| $ 0 | Pr 1 | $ 19 | | | | | | | reduce $S' -> Pr $ |
| $ 0 | $S' -1 | | | | | | | | accept |

**2. JLex** **[5 Marks]**

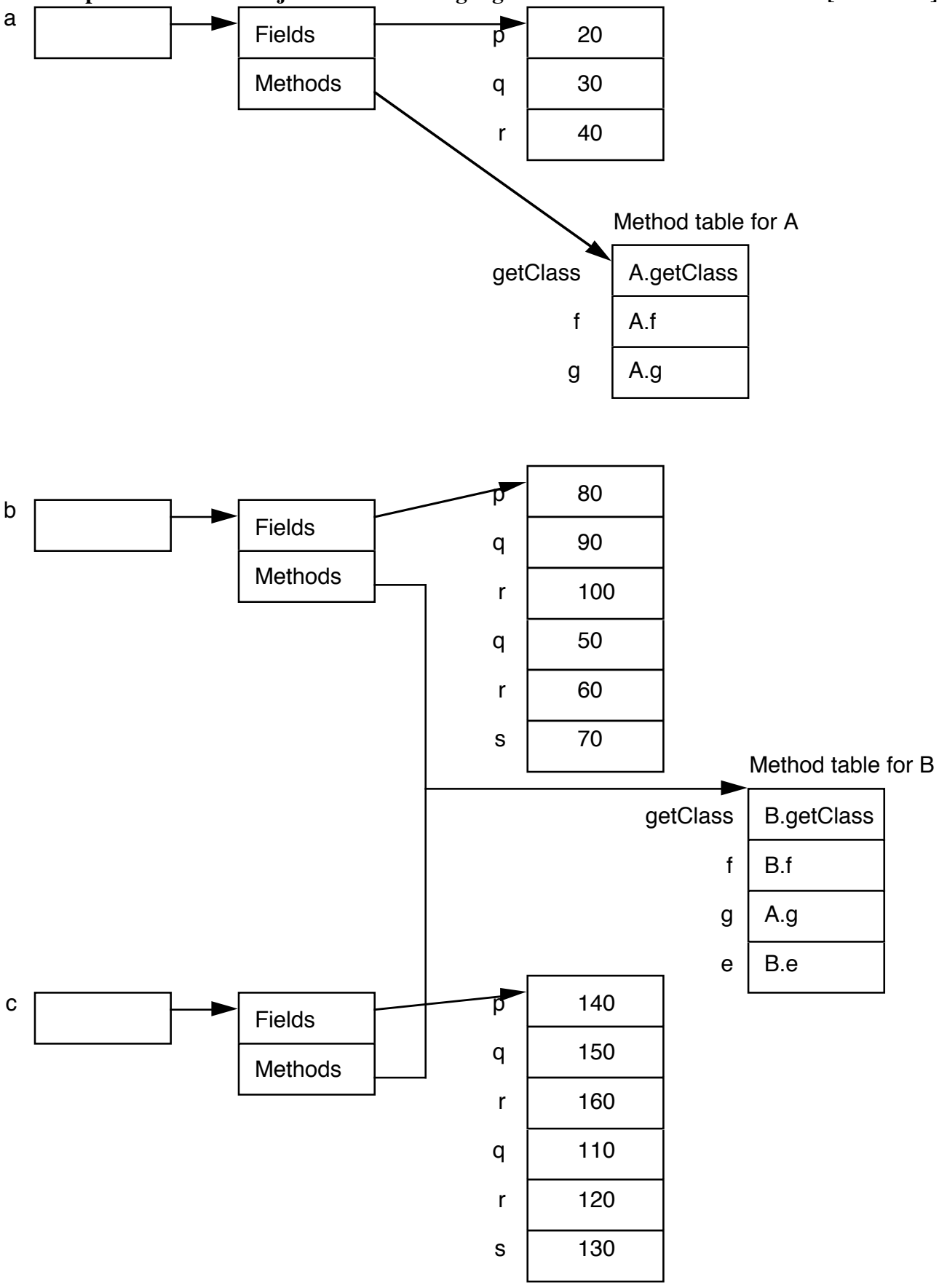
Specify JLex definitions to represent regular expressions for real numbers.

```
exponent        =    ([Ee][\+\-]?[0-9]+)
float1          =    ([0-9]+\.[0-9]+)
float2          =    ([0-9]+{exponent})
float3          =    ([0-9]+\.[0-9]+{exponent})
float     =    ({float1}|{float2}|{float3})
```

**3. Write a grammar definition for sets of integers [15 Marks]**

```
terminal
     LEFT, RIGHT, LEFTBRACE, RIGHTBRACE, COMMA, DOTDOT, UNION, INTERSECT;

terminal
     String INTCONST;

non terminal
     Set1, Set2, Set3, BasicSet, ComponentList, Component;

start with Set1;

Set1::=
          Set1 UNION Set2
     |
          Set2
     ;

Set2::=
          Set2 INTERSECT Set3
     |
          Set3
     ;

Set3::=
          LEFT Set1 RIGHT
     |
          BasicSet
     ;

BasicSet::=
          LEFTBRACE RIGHTBRACE
     |
          LEFTBRACE ComponentList RIGHTBRACE
     |
          LEFTBRACE ComponentList COMMA INTCONST DOTDOT RIGHTBRACE
     ;

ComponentList::=
          Component
     |
          ComponentList COMMA Component
     ;

Component::=
          INTCONST
     |
          INTCONST DOTDOT INTCONST
     ;
```

**6. Implementation of Object Oriented Languages** **[10 Marks]**



```
a.p = 20
a.q = 30
a.r = 40
b.p = 80
b.q = 50
b.r = 60
b.s = 70
c.p = 140
c.q = 150
c.r = 160
c.f() = B.f
c.g() = A.g
c.h() = A.h
```

# Computer Science 330 Language Implementation

# Examination Information 2001 - Confirmed

This Examination is out of 70 Marks. Attempt **ALL** questions. Write your answers in the spaces provided in the booklet.

| | | |
|---|---|---|
| 1 | | 10 |
| 2 | | 5 |
| 3 | | 15 |
| 4 | | 15 |
| 5 | | 15 |
| 6 | | 10 |
| Total | | 70 |

**1. Bottom Up LALR(1) Parsing [10 Marks]**

Consider the CUP grammar:

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

**2. JLex [5 Marks]**

Specify JLex regular expressions for …

**3. Write a grammar definition for ... [15 Marks]**

You only have to write the grammar. You do not have to write any actions.

**4. Show the run time stack. [15 Marks]**

Below is a program in the EXPREVAL6 "b" language, which allows nesting of function declarations. Show the run-time stack when the maximum level of nesting of function invocations occurs for the first time, and the process is almost ready to return. Remember the stack grows "up" towards low memory. Enter the appropriate values for each stack frame (activation record) you draw. Assume that all variables are implicitly initialised to zero or null. The numbers on the left-hand side of the program represent the "instruction addresses". Draw appropriate arrows for the var parameters, and pointers to structured objects. The stack information has already been filled in for the main program, and the first stack frame. You may label addresses, and refer to them, rather than drawing arrows, if you need to, to make your answer clearer.

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

Note that this questions involves nested function declarations, var parameters, structured types, and use of “new” to create a new initialised structured object. It is best to be familiar with the syntax of the EXPREVAL6 "b" language (almost the same as the EXPREVAL7 "b" language, but with nested function declarations, and a slightly different syntax for the main program).

**5. Compile time checking and code generation [15**

Indicate the algorithms and information (attributes or data structures) needed to perform compile time checking and code generation for some kind of construct. Perform the algorithm and indicate the code generated for a specific example. For example, function declaration/invocation, arithmetic expressions, control statements, break and continue statements, Boolean expressions, simple and field variables, switch statements.

**6. Implementation of Object Oriented Languages [10 Marks]**

Suppose we have the Java program …

Draw a diagram showing the data structures (object, field table and method table) created for the variables …, within the method Main.main.

Indicate the output generated by the method Main.main.

## Bruce Hutton and Mike Barley

## Exercises

### Exercise 1(a)

Try drawing the stack and appropriate data structures at the maximal level of recursion

```
type List = struct( value : int;  next : List; );

funct printList( a : List; ) void
        begin
                print( "{ " );
                while a <> null do
                        print( a.value );
                        a = a.next;
                        if a <> null then
                                print( ", " );
                        end
                end
                print( " }" );
        end

funct appendHead( var a : List; data : int; ) void
        begin
                a = new List { data, a };
        end

funct insertList( var a : List; data : int; ) void
        begin
                if a == null or data <= a.value then
                        appendHead( a, data );
                else
                        insertList( a.next, data );
                end
        end

funct main() void
        var a, a1, a3, a5 : List;
        begin
                a5 = new List { 5, null };
                a3 = new List { 3, a5 };
                a1 = new List { 1, a3 };
                a = a1;
                insertList( a, 4 );
                printList( a );
                println();
        end
```

### Exercise 1(b)

Try drawing the stack and appropriate data structures at the maximal level of recursion

```
type List = struct( value : int;  next : List; );

funct appendText( var result : string; text : string; ) void
        begin
```

```
                result = result + text;
        end

funct buildListText( a : List; var result : string; ) void
        begin
                if a <> null then
                        appendText( result, a.value );
                        if a.next <> null then
                                appendText( result, ", " );
                                buildListText( a.next, result );
                        end
                end
        end
funct main() void
        var a, a1, a3, a5 : List;
        var result : string;
        begin
                a5 = new List { 5, null };
                a3 = new List { 3, a5 };
                a1 = new List { 1, a3 };
                a = a1;
                result = "{ ";
                buildListText( a, result );
                appendText( result, " }" );
                println( result );
        end
```

**Exercise 1(c)**

Try drawing the stack and appropriate data structures at the maximal level of recursion

```
type List = struct( value : int;  next : List; );

funct buildListText( a : List; var result : string; ) void
        var result1 : string;
        begin
                if a <> null then
                        if a.next == null then
                                result = "" + a.value;
                        else
                                buildListText( a.next, result1 );
                                result = a.value + ", " + result1;
                        end
                end
        end
funct main() void
        var a, a1, a3, a5 : List;
        var result : string;
        begin
                a5 = new List { 5, null };
                a3 = new List { 3, a5 };
                a1 = new List { 1, a3 };
                a = a1;
                buildListText( a, result );
                println( "{" + result + "}" );
        end
```

**Exercise 2**

Which determines the meaning for the following - the declared type, or the actual type?

- static variables.
- static methods.

- instance variables.
- instance methods.

If you overload a method

```
void t( int x, int y )
void t( double x, double y )
```

in a superclass
then override the method

```
void t( double x, double y )
```

in a subclass, what happens if you try to invoke t(1,2) in the subclass?
How is an object stored at run time?
What can you say about the layout of an object of a subclass, compared with the layout of an object of the superclass?
What can you say about the allocation of space, when

- an instance variable is declared with the same name as one in a superclass?
- an instance method is declared with the same name as one in a superclass?
- what if the name is the same, but the signature is different?

What can you say about storage for two variables a, b of reference type, after an assignment "a = b;"?
What can you say about storage for two variables a, b of reference type, when the actual values are of the same type?
What is the output from the following program?
Draw diagrams showing the layout for the objects.

```
class A {
     static int a = 1, b = 2, c = 3;
     int g = 11, h = 12, i = 13;
     static void l() { System.out.println( "A.l()" ); }
     static void m() { System.out.println( "A.m()" ); }
     void o() { System.out.println( "A.o()" ); }
     void p() { System.out.println( "A.p()" ); }
     void t( int x, int y ) { System.out.println( "A.t(int, int)" ); }
     void t( double x, double y ) { System.out.println( "A.t(double, double)" ); }
     A( int g, int h, int i ) { this.g = g; this.h = h; this.i = i; }
     }

class B extends A {
     static int b = 4, c = 5, d = 6;
     int h = 14, i = 15, j = 16;
     static void m() { System.out.println( "B.m()" ); }
     static void n() { System.out.println( "B.n()" ); }
     void o() { System.out.println( "B.o()" ); }
     void r() { System.out.println( "B.r()" ); }
     void t( int x, int y ) { System.out.println( "B.t(int, int)" ); }
     B( int Ag, int Ah, int Ai, int g, int h, int i, int j ) {
          super( Ag, Ah, Ai );
          this.g = g; this.h = h; this.i = i; this.j = j;
          }
     }

class C extends B {
     static int c = 7, e = 8;
     int h = 17, k = 18;
     static void l() { System.out.println( "C.l()" ); }
     static void m( int x ) { System.out.println( "C.m(int)" ); }
     void r() { System.out.println( "C.r()" ); }
     void s() { System.out.println( "C.s()" ); }
     C( int Ag, int Ah, int Ai, int Bg, int Bh, int Bi, int Bj, int h, int k ) {
```

```
            super( Ag, Ah, Ai, Bg, Bh, Bi, Bj );
            this.h = h; this.k = k;
            }
      }

class D extends B {
      static int e = 9, f = 10;
      int a = 19, i = 20, k = 21;
      static void l() { System.out.println( "D.l()" ); }
      void o() { System.out.println( "D.o()" ); }
      void r( int x ) { System.out.println( "D.r(int)" ); }
      D( int Ag, int Ah, int Ai, int Bg, int Bh, int Bi, int Bj, int a, int i, int k ) {
            super( Ag, Ah, Ai, Bg, Bh, Bi, Bj );
            this.a = a; this.i = i; this.k = k;
            }
      }

public class JavaTest {

      static void printAll( String name, A a ) {
            System.out.println( name + ".a = " + a.a );
            System.out.println( name + ".b = " + a.b );
            System.out.println( name + ".c = " + a.c );
            System.out.println( name + ".g = " + a.g );
            System.out.println( name + ".h = " + a.h );
            System.out.println( name + ".i = " + a.i );
            a.l();
            a.m();
            a.o();
            a.p();
            a.t(1,2);
            }

      static void printAll( String name, B b ) {
            System.out.println( name + ".a = " + b.a );
            System.out.println( name + ".b = " + b.b );
            System.out.println( name + ".c = " + b.c );
            System.out.println( name + ".d = " + b.d );
            System.out.println( name + ".g = " + b.g );
            System.out.println( name + ".h = " + b.h );
            System.out.println( name + ".i = " + b.i );
            System.out.println( name + ".j = " + b.j );
            b.l();
            b.m();
            b.n();
            b.o();
            b.p();
            b.r();
            b.t(1,2);
            b.t(1.0,2);
            }

      static void printAll( String name, C c ) {
            System.out.println( name + ".a = " + c.a );
            System.out.println( name + ".b = " + c.b );
            System.out.println( name + ".c = " + c.c );
            System.out.println( name + ".d = " + c.d );
            System.out.println( name + ".e = " + c.e );
            System.out.println( name + ".g = " + c.g );
            System.out.println( name + ".h = " + c.h );
```

```
        System.out.println( name + ".i = " + c.i );
        System.out.println( name + ".j = " + c.j );
        System.out.println( name + ".k = " + c.k );
        c.l();
        c.m();
        c.m(3);
        c.n();
        c.o();
        c.p();
        c.r();
        c.s();
        c.t(1,2);
        c.t(1.0,2);
        }

    static void printAll( String name, D d ) {
        System.out.println( name + ".a = " + d.a );
        System.out.println( name + ".b = " + d.b );
        System.out.println( name + ".c = " + d.c );
        System.out.println( name + ".d = " + d.d );
        System.out.println( name + ".e = " + d.e );
        System.out.println( name + ".f = " + d.f );
        System.out.println( name + ".g = " + d.g );
        System.out.println( name + ".h = " + d.h );
        System.out.println( name + ".i = " + d.i );
        System.out.println( name + ".j = " + d.j );
        System.out.println( name + ".k = " + d.k );
        d.l();
        d.m();
        d.n();
        d.o();
        d.p();
        d.r();
        d.r(3);
        d.t(1,2);
        d.t(1.0,2);
        }

    public static void main( String[] arg ) {
        A a = new A( 22, 23, 24 );
        B b1 = new B( 25, 26, 27, 28, 29, 30, 31 );
        B b2 = new B( 32, 33, 34, 35, 36, 37, 38 );
        C c = new C( 41, 42, 43, 44, 45, 46, 47, 48, 49 );
        D d = new D( 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 );
        A a1 = b1;
        A a2 = b2;
        A a3 = c;
        A a4 = d;
        B c1 = c;
        printAll( "a", a );
        printAll( "b1", b1 );
        printAll( "b2", b2 );
        printAll( "c", c );
        printAll( "d", d );
        printAll( "a1", a1 );
        printAll( "a2", a2 );
        printAll( "a3", a3 );
        printAll( "a4", a4 );
        printAll( "c1", c1 );
        }
```

```
    }
```

**Exercise 3**

What data structures and code would you expect to generate for the switch statement, in your own implementation of switch statements?

```
int x, y;
...
switch x of
     case -1, 1:
          y = 1;
     case -2, 0, 2:
          y = -1;
     default:
          y = 0;
     end
```

**Exercise 4**

describe the code generation algorithm for binary arithmetic operators.

Explain why it is necessary to have a tree weighting pass for the compiler?

What difference would it make if we omitted this phase, and dealt with running out of registers when it occurred?

Consider the assignment statement

```
a = b - c + ( d / e ) * ( f / g );
```

Perform tree weighting, and code generation, on the assumption that there are only 2, 3, or 4 available registers, and assuming that the compiler has been improved to generate good code for simple variables.

what is the layout for a stack frame in the EXPREVAL7 compiler?

Suppose a function had 3 formal parameters and 2 local variables, and needed two temporary memories (due to running out of temporary registers). what would the stack frame look like?

Describe the code generation algorithm for binary logical operators.

Consider the if statement

```
if a < b and c < d or e < f and g < h then
     x = 1;
else
     x = 2;
```

What code would be generated?

Explain it in terms of your recursive code generation algorithm.