Instructions: This is a closed book, closed note, 120 minute exam.

There are 3 questions worth a total of 100 points. There is a 5 point extra credit (EC) question. Read all of the questions before starting to answer so you can allocate your time appropriately.

You must answer all of the questions. Put your name on all pieces of paper that you turn in. Show all of your work, write legibly and good luck.

1) (70 points) You are to add a new language feature to ESJ and enhance your compiler appropriately. The language currently supports a number of control structures and you are to add support for `switch` statements. Your switch will improve on Java's switch by eliminating certain features that are error prone, e.g., `break`, and adding others to maintain expressivenes, e.g., multiple case values.

Here is an example of an ESJ `switch`:

```
switch (x) {
  case 1: y = x;
  case 2,4,6,8 : y = x/2;
  case 9-13 : y = x-7;
  default : y = -1;
}
```

As you can see, there are some similarities and differences with Java's switch. We outline those issues here:

1. there is no `break` statement, all case bodies tranfer control to the statement following the switch;
2. the optional `default` case is supported;
3. case *values* must be constants;
4. cases may specify a comma-separated list of *values*;
5. cases may specify a dash-separated inclusive range of *values*;
6. *values* in all cases must match the type of the `switch` parameter; and
7. the *values* for each case must be distinct from all other cases.

This page contains some useful information about JDT and JVM support for switch statements.

The JDT has support for representing switch statements. The `SwitchStatement` node is a `Statement` node that has two children : `Expression` node and a list of `Statements`. The statement list is an alternating sequence of `SwitchCase` statements and instances of `Statement`, e.g., return, if, block, or any other statement, that constitute the case body. The convention is that an adjacent pair of `SwitchCase` and subsequent `Statement` form a case-body pair of the switch statement. The `SwitchCase` has a single child expression that is the case value. The default case is identified with a `null` expression value.

The JVM has specialized support for code generation related to switch statements. The `lookupswitch` bytecode works as follows:

```
lookupswitch N:
    v1 : L1
    v2 : L2
    ...
    vN : LN
    default : L0
```

introduces a `N`-way branching structure where each of the `vi` correspond to a value compared to the value on the top of the stack and where the matching of a value `vi` causes a `goto Li`. The default case, `goto L0`, is executed if no `vi` matches the top of the stack.

1 continued) In answering the following questions, I expect you to describe in detail the changes you would make to the ESJ compiler. You may use a mixture of descriptions including English text, compiler notations (e.g., regular expressions), diagrams, and pseudo-code sketches. I do not expect you to write down correct Java code, but writing code to help illustrate your approach is fine.

a) **(10 pts)** Enhance the scanner to handle the new lexical elements.

b) **(10 pts)** Enhance the parser to handle the new syntactic elements.

c) **(10 pts)** Enhance the AST construction actions in the parser to handle the new language elements using the JDT structures described above. Illustrate the AST for the example given above.

d) **(10 pts)** Enhance the weeding pass to enforce any specific constraints that need to be applied for switch statements. Show a switch statement for which this pass would catch an error.

e) **(10 pts)** Sketch out type rules for switch; focus on expressing the essential ideas of the type rules. Illustrate the type derivation tree for the example given above and indicate whether it is type correct.

f) **(20 pts)** Enhance the code generator to handle `switch`. Be clear in explaining your high-level code generation strategy, what resources if any you need to calculate and allocate, and show your code generation template. Illustrate the generated JVM bytecodes for the given example.

EC) **(5 pts)** Briefly explain how you would change your solution above to support the `break` statement.

2) (10 points) This question is about how to use the results of static analysis in optimizing code generation. You have studied *live variables* (LV) analysis. In this problem, you are to describe an approach that use the results of LV analysis to improve the code generated by the compiler. You may refer to bytecode generation, native code generation, peephole optimization, or any other approach to generating or improving code.

For full points, you must

- explain how the information in LV is used;
- explain how the code is improved;
- explain why the improvement is correct; and
- illustrate the improvement on a small example.

3) (20 points) Consider the arithmetic expression grammar defined over the identifiers `a`, `b`, and `c` and the integer literal constants with operators `+`, `-`, `*`, and `( )` for grouping.

a) **(10 pts)** Write a context-free grammar that recognizes this language that enforces operator precedence.

b) **(5 pts)** Show a leftmost derivation for the grammar applied to the input :
`a * b + 1 - (a + c) * 2`

c) **(5 pts)** Draw the parse tree for that derivation.

Do not use any special ANTLR or other parser generator features to solve this problem.