1. The reference manual for a MiniJava-like programming language contains the following grammar rule for a do-while statement:

$$\textit{Statement} \;\rightarrow\; \textbf{do}\ \textit{Statement}\ \textbf{while}\ (\textit{Exp})\ ;$$

(a) Write down or draw a possible abstract syntax for the do-while statement. [10]

Answer:

```
Do Statement Exp
(be flexible about how this is expressed - it might be Java and have types;
it might be a picture)
```

(b) Show how semantic actions in a grammar for a parser-generator such as JavaCC can be used to produce abstract syntax trees for the do-while statement. [25]

Answer:

```
Statement DoWhileStatement() :
{ Exp e;
  Statement s;
}
{
  "do" s=Statement() "while" "(" e=Expression() ")"
  { return new DoWhile(s,e); }
}
```

5 marks for framework, 10 marks for syntax, 10 marks for correct actions.

(c) Informally describe an appropriate typecheck for the do-while statement. [10]

Answer: e must be a boolean expression.

(d) Suppose a compiler for a MiniJava-like language translates all statements and expressions into intermediate code (eg intermediate representation (IR) trees).

i. Draw or write down the intermediate representation required to access a local variable declared in a method. Explain your answer. [20]

Answer: MEM(BINOP(PLUS,TEMP fp, CONST k)) where k is offset of var in frame, fp the register holding the framepointer. Has to compute place in frame.

ii. Suppose the MiniJava-like language includes a do-while statement. Outline the intermediate code that might be generated in translation of the do-while statement. You may wish to use a simple example to explain your translation, eg:

```
do {
  x = i*i; System.out.println (x); i = i+1;
} while (i < j) ;
```

You can assume that the expression tree for any variable `v` is simply `TEMP v`. You need not show translations for the body of the example do-while statement (in braces in this example `{...}`). [35]

Answer:

```
LABEL(Lstart)
code for body (here square, print and add 1)
CJUMP(<,TEMP i,TEMP j,Lstart,Lend)
LABEL(Lend)
```

Might be expressed as trees.

2. (a) The following regular expression recognises certain strings over the alphabet $\{a, b, c\}$

$$(a|c)((bc)|c)*\,c*$$

Indicate which of these five strings are recognised by the above regular expression:

cbc, $abbc$, c, $abcbcbccc$, $ccbbcbcc$

Also, show three more strings that are recognised by the above expression. Finally, show two more strings consisting of the letters a, b and c that are *not* recognised by the above regular expression. [30]

Answer: Yes, No, Yes, Yes, No. 4 marks each. Five further strings, 2 marks each.

(b) Consider the following grammar for strings of balanced parentheses:

$$S \rightarrow S\,S$$
$$S \rightarrow (\ S\ )$$
$$S \rightarrow (\ )$$

Explain what it means for a context-free grammar to be ambiguous. Using your explanation, show that the balanced parentheses grammar is ambiguous using the shortest string that will illustrate the ambiguity. [30]

Answer: Two parse trees, two derivations, for same sentence. 10 marks. 20 for the two trees of () () ().

(c) Explain why left-recursion must be eliminated from grammar productions which are to be used in construction of a recursive-descent parser. Write down a general rule for rewriting left-recursive grammar productions to be right-recursive and use it to rewrite the following productions for an arithmetic expression grammar to be right-recursive:

$$E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow F$$
$$F \rightarrow (\ E\ )$$
$$F \rightarrow \texttt{integer}$$

[40]

Answer: Because the usual way of writing the procedures leads to immediate recursive call. 5 marks.

Rule (10 marks):

```
A  -> A a | b
(a and b strings of terms and non-terms)
rewrites to
A  -> b A'
A' -> a A' | empty
```

Answer (30 marks):

```
E  -> T E'
E' -> + T E' | empty
T  -> F T'
T' -> * F T' | empty
F  -> ( E )  | integer
```

3. (a) i. State two reasons why many programming language implementations require a memory model that implements a runtime stack? [10]

ii. Explain in detail how a stack frame is pushed to the stack, and removed from the stack, during program execution. [25]

Answer:

Procedure or method calls, recursion, need for separate storage space for parameters and locals. Code generated for a proc/func does the pushing/popping.

```
caller g(...) calls callee f(a1,...,an)
calling code in g puts arguments to f at end of g frame
stores return address, old FP in control link
referenced through SP, incrementing SP
on entry to f, SP points to first argument g passes to f
old SP becomes current frame pointer FP
f then allocates frame by setting SP=(SP - framesize)
old SP becomes current frame pointer FP
f then initialises locals
on exit from f : SP = FP, removing frame
jumps to return address, restores old FP
```

5 marks each reason for answer to first part and explanation. 25 marks for details.

(b) i. Some programming language implementations avoid in some circumstances the need to pass parameters via a stack frame. Outline what these circumstances might be and why passing via the stack frame might be avoided. [15]

Answer:

Appropriate when leaf procs, interproc reg alloc, dead variables, reg windows (but...).

ii. Explain three situations where the use of a stack frame to pass parameters cannot usually be avoided. [15]

Answer:

Reg saves: when address is taken, when call-by-ref, when accessed by inner nesting, value too big, an array, convention of save for partic reg prior to call, spilling in exp evaluation, saving a reg window.

(c) i. Explain the difference between *caller-save* and *callee-save* registers. [10]

ii. Study the following methods and suggest for each whether a caller-save or callee-save register is appropriate for variable `x`. Explain your answers.

```
int f (int a) { int x; x=a+1; g(x); return x+2; }

void p (int y) { int x; x=y+q(y); q(2); q(y+1)}
```

[25]

Answer:

Caller-save if code for caller of a func saves and restores the reg value around a func call. Callee save if code for a func does it. First method x in callee-save, since x live across the method calls. Second caller-save, since x not live after x=y+q(y) (so code generated shouldn't save it).
10 marks for explanation, 12.5 each for x answers and explanations.