1. (a) The following regular expression recognises certain strings consisting of the letters a, b and c:

$$a(b|(bc))*c*$$

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

acc, $abac$, a, $abcbcbccc$, $abbbccbc$

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. [25]

Answer: Yes, No, Yes, Yes, Yes. 3 marks each. Five further strings, 2 marks each.

(b) Consider the following grammar, which we will call E:

$E \rightarrow E$ + E
$E \rightarrow E$ - E
$E \rightarrow$ (E)
$E \rightarrow$ num

i. Explain what it means for a context-free grammar to be ambiguous. Show that grammar E is ambiguous. [20]

Answer: Two parse trees, two derivations, for same sentence. 5 marks. 15 for the two trees/derivations.

ii. Explain why grammar E is not suitable to form the basis for a recursive descent parser. [10]

Answer: Because it's left-recursive and so the usual way of writing the procedures leads to immediate recursive call.

iii. Rewrite the rules to obtain an equivalent grammar which can be used as the basis for a recursive descent parser. [25]

Answer:

```
E -> num E'
E -> (E) E'
E' -> + E E'
E' -> - E E'
E' -> empty
```

(c) Consider the following Java method:

```
1  class A {
2   String a; int c;
3   public void f(int b, String c) {
4     System.out.println(c);
5     int d = 3;
6     int a = b;
7     System.out.println(a+d); System.out.println(b);
8     System.out.println(c); System.out.println(d);
9   }
10 }
```

Given an initial environment σ_0, derive the type binding environments for the method at each use of an identifier and indicate where type lookups will occur. [20]

Answer:

0 σ_0 is starting environment
2 $\sigma_1 = \sigma_0 + \{a \to string, c \to int\}$
3 $\sigma_2 = \sigma_1 + \{b \to int, c \to String\}$ (overrides instance c)
4 lookup id c in σ_2
5 $\sigma_3 = \sigma_2 + \{d \to int\}$
6 lookup id b, then $\sigma_4 = \sigma_3 + \{a \to int\}$ (overrides instance a)
7 lookup a, d, b in σ_4
8 lookup c, d in σ_4
9 discard σ_4 revert to σ_1
10 discard σ_1 revert to σ_0

2 marks for each line

2. The reference manual for a MiniJava-like programming language contains the following grammar for a for-statement:

$$\textit{Statement} \rightarrow \textbf{for}\ (\textit{id} = \textit{Exp}\ ;\ \textit{Exp}\ ;\ \textit{id} = \textit{Exp})\ \textit{Statement}$$

(a) Sketch a possible abstract syntax for the for-statement. [20]

Answer:

```
For Id Exp Exp Id Exp Statement
(be flexible about how this is expressed - it might be Java and have types)
```

(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 for-statement. [30]

Answer:

```
Statement ForStatement() :
{ Identifier i1, i2;
  Exp e1, e2, e3;
  Statement s;
}
{
  "for" "(" i1=Identifier() "=" e1=Expression() ";"
            e2=Expression() ";"
            i2=Identifier() "=" e3=Expression()
        ")" s=Statement()
  { return new For(i1,e1,e2,i2,e3,s); }
}
```

10 marks for syntactic structure, 15 for actions, 5 for a small explanation.

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

Answer:

i1 should be same type as e1, i2 same type as e3 (5 marks); e2 must be a boolean expression (5 marks).

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

Outline the intermediate code that might be generated in translation of the for-statement. You may wish to use a simple example to explain your translation, eg:

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

You can assume that the expression tree for any variable `v` is simply `TEMP v`. Do not show translations for the body of the example for-statement (in braces in this example `{...}`). [40]

Answer:

```
MOVE(TEMP i,TEMP j)
LABEL(Lstart)
CJUMP(<,TEMP i,TEMP k,Lbody,Lend)
LABEL(Lbody)
code for body (here square and print)
MOVE(TEMP i,BINOP(+,TEMP i,1))
JUMP(Lstart)
LABEL(Lend)
```

Might be expressed as trees.

3. (a) Why do many programming language implementations require a memory model that implements a runtime stack? Explain in detail how a stack frame is pushed to the stack, and removed from the stack, during program execution. [30]

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
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
```

10 marks for answer to first part and explanation. 20 marks for details.

(b) 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. Also, outline situations where the use of a stack frame to pass parameters cannot usually be avoided. [25]

Answer:

Appropriate when leaf procs, interproc reg alloc, dead variables, reg windows (but...).
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.
5 marks for explanation, 2 each for details.

(c) Suppose that a compiler translates a MiniJava-like language to an intermediate representation (for example IR trees) that will include the calculations required to address variables in stack frames. 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.

(d) Explain the difference between *caller-save* and *callee-save* registers. 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+1 (so code generated shouldn't save it).
5 marks for explanation, 10 each for x answers and explanations.