1. (a) What is an ambiguous grammar? Show that the following grammar is ambiguous:

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{if } E \textbf{ then } S \\
 & | & \textbf{if } E \textbf{ then } S \textbf{ else } S \\
 & | & \textit{other}
\end{array}
$$

Here *other* stands for any other statement *S*. [15]

Answer:

Two parse trees for `if a then if b then s1 else s2`.

(b) The reference manual for a MiniJava-like programming language contains the following grammar rule for an if-statement:

Statement → **if** (*Exp*) *Statement* **else** *Statement*

i. Sketch a possible abstract syntax for the if-statement. [10]

Answer:

```
If Exp Statement Statement
(be flexible about how this is expressed - it might be Java and have type
```

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

Answer:

```
Statement IfStatement() :
{ Exp e;
  Statement s1,s2;
}
{
  "if" "(" e=Expression() ")" s1=Statement() "else" s2=Statement()
  { return new If(e,s1,s2); }
}
```

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

iii. Informally describe an appropriate typecheck for the if-statement. [10]

Answer:

e must be a boolean expression.

iv. Suppose a compiler for a MiniJava-like language that includes a if-statement translates all statements and expressions into intermediate code, for example intermediate representation (IR) trees. Outline the intermediate code that might be generated in translation of the if-statement. You may wish to use a simple example to explain your translation, eg:

```
if (a < b) c = a; else c = b;
```

You can assume that the expression tree for any variable v is simply `TEMP v`. [40]

Answer:

```
SEQ(SEQ(CJUMP(LT, TEMP a, TEMP b, L0, L1),
        SEQ(SEQ(SEQ(LABEL L0,
                    MOVE(TEMP c, TEMP a)),
                JUMP(NAME L2)),
            SEQ(SEQ(LABEL L1,
                    MOVE(TEMP c, TEMP b)),
                JUMP(NAME L2)))),
    LABEL L2)
```

(might be a longer translation where a ¡ b is evaluated to the 0 or 1 and then checked)
Might be expressed as trees.
15 marks for the comparison, 15 marks for appropriate jumping code, 10 marks for the assignments code.

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

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

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, No. 3 marks each. Five further strings, 2 marks each.

(b) Write a regular expression that recognises strings over the alphabet $\{a, b, c\}$ where there is an even number of a's. [20]

Answer:

```
(b|c)*(a(b|c)*a(b|c)*)*
or ((b|c)*a(b|c)*a)(b|c)*
```

(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 to be right-recursive:

$$\begin{array}{lcl} S & \rightarrow & (\,L\,) \mid a \\ L & \rightarrow & L\,,S \mid S \end{array}$$

[35]

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 (20 marks):

```
S  -> ( L ) | a
L  -> S L'
L' -> , S L' | empty
```

(d) Consider the following Java class:

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

3. (a) What programming language feature leads to the need for an implementation model that includes stack frames? Explain your answer. 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) 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.

(c) Explain why registers might be used for parameter passing and suggest situations where passing in registers is particularly appropriate. Outline situations where it is necessary for the code generated for a procedure or method to write registers to the stack. [25]

Answer:

Efficiency; particularly 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.

(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(); h(x); return x+2; }

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

[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.