NAME: _____

STUDENT ID: _____

SIGNATURE: _____

The University of New South Wales

Final Examination

November/December 2001

COMP3131/COMP9102

Programming Languages and Compilers

Time allowed: **2 hours**

Total number of questions: **6**

Answer **all** questions

The questions are **not** of equal value

Each question must be answered in a separate book

This paper may be retained by the candidate

No examination materails

Answers must be written in ink.

Question 1. Context-Free Grammars (CFGs) *[10 marks]*

Consider the following CFG:

$$
\begin{array}{lcl}
S & \rightarrow & C\ eof \\
C & \rightarrow & C\ \textbf{repeatwhile}\ E \\
C & \rightarrow & \{B\} \\
C & \rightarrow & E = E\ ; \\
B & \rightarrow & B\ C \\
B & \rightarrow & C \\
E & \rightarrow & \textsf{id}
\end{array}
$$

where "C **repeatwhile** E" has the same meaning as "**do** C **while** E" in Java or C.

(a) Give a parse tree for { **id** = **id**; } **repeatwhile** **id** *eof*.

(b) Give a grammar that is free of the left recursion, ensuring that your revised grammar generates exactly the same set of strings.

Question 2. Regular Grammars, Regular Expressions and Finite Automata *[30 marks]*

A *regular grammar* is a grammar whose productions are in one of the following two forms (where A an B are nonterminals and a is a terminal):

$$\begin{aligned} A &\rightarrow a \\ A &\rightarrow aB \end{aligned}$$

(a) Give a regular grammar which generates the floating point numbers specified exactly by the following regular expression:

$$(0|1)^{+}.(0|1)^{*}[e(0|1)^{+}]$$

where "( )" indicates grouping, "[ ]" indicates optional item, "ρ^{+}" indicates one of more repetitions of ρ and "ρ^{*}" indicates zero or more repetitions of ρ.

(b) Give a non-regular CFG with fewer productions than your answer to (a) but which generates the same set of strings.

(c) Convert the regular expression given in (a) into a nondeterministic finite automaton (NFA). You may do so using Thompson's construction.

(d) Use the subset construction to convert the NFA from (c) into a deterministic finite automaton (DFA).

Question 3. Top-Down Parsing *[20 marks]*

Consider the following grammar:

$$\begin{array}{lcl} S & \to & UV \\ U & \to & XW \\ V & \to & +UV \mid \epsilon \\ W & \to & *XW \mid \epsilon \\ X & \to & (S) \mid n \end{array}$$

(a) Describe an algorithm to calculate the set $\mathsf{FOLLOW}(A)$ defined as:

$$\mathsf{FOLLOW}(A) \;=\; \{a \mid S \stackrel{*}{\Longrightarrow} \cdots Aa \cdots\}$$

That is, $\mathsf{FOLLOW}(A)$ is the set of terminals a that can follow a nonterminal A in a sentential form derived from the start symbol S.

(b) Construct FOLLOW sets for all nonterminals.

(c) Construct the LL(1) parsing table for the grammar.

Question 4. Attribute Grammars *[20 marks]*

Consider the following grammar for integer binary trees (in linearised form):

$$btree \quad \rightarrow \quad (\ \textbf{num}\ btree\ btree)\ \mid \textbf{nil}$$

(a) Write an attribute grammar to check that a binary tree is a binary search tree. A search tree is a *binary search tree* if the value of the number at each node is >= the values of the numbers of its left subtree and < the values of the numbers of its right subtree. For example,

(2 (1 nil nil) (3 nil nil))

is a binary search tree, but

(1 (2 nil nil) (3 nil nil))

is not.

(b) Is your attribute grammar L-attributed or S-attributed?

(c) Is each attribute in your attribute grammar a synthesised or inherited attribute?

Question 5. Type Checking *[5 marks]*

Consider the following VC program:

```
int f(int f) {
int f = 9102;
{
        float f;
        int putIntLn = 3131;
        putIntLn(PutIntLn);
}
putIntLn(f);
return f;
}

void main() { }
```

(a) Identify all semantic errors.

(b) Give the block level numbers for all declarations.

Question 6. Code Generation *[15 marks]*

Suppose we introduced a **switch-if** statement into VC:

```
switch expr if
        positive stmt1;
        zero stmt2;
        negative stmt3;
end;
```

The semantics of this statement are to evaluate *expr* and execute *stmt1*, *stmt2* or *stmt3* depending on whether *expr* is positive, zero or negative.

Suppose we used the following class to represent **switch-if** statements in the AST:

```
/* ============= SwitchIfStmt.java  ==============  */

package VC.ASTs;

import VC.Scanner.SourcePosition;

public class  SwitchIfStmt extends Stmt {

  public Expr E;
  public Stmt S1, S2, S3;

  public  SwitchIfStmt(Expr eAST, Stmt s1AST, Stmt s2AST, Stmt s3AST,
                                                SourcePosition Position) {
    super (Position);
    E = eAST;
    S1 = s1AST;
    S2 = s2AST;
    S3 = s3AST;
    E.parent = S1.parent = S2.parent = S3.parent = this;
  }
  public Object visit(Visitor v, Object o) {
    return v.visitSwitchIfStmt(this, o);
  }
}
```

(a) Show how to generate Jasmin code using Java, i.e., give Emitter.visitSwitchIfStmt.

(b) Show Jasmin code for the following **switch-if** statement:

```
switch (i + 1) if
        positive i = 1;
        zero i = 0;
        negative i = -1;
end;
```

The index for the variable i is assumed to be 1.