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

No examination materials

Answers must be written in ink.

Question 1. Consider the following CFG:

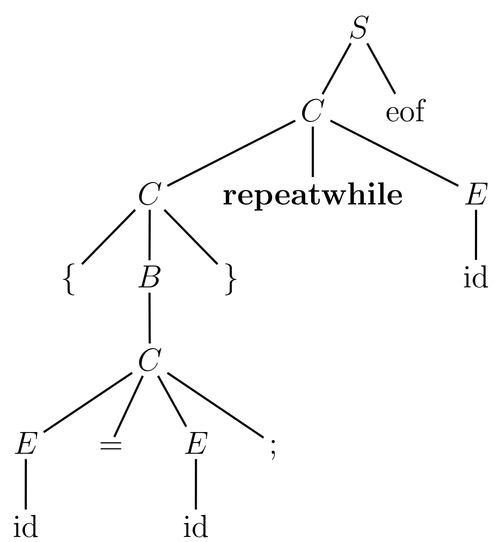
$$
\begin{array}{lcl}
S & \rightarrow & C\ eof \\
C & \rightarrow & C\ \textbf{repeatwhile}\ E \\
C & \rightarrow & \{B\} \\
C & \rightarrow & E = E\ ; \\
B & \rightarrow & B\ C \\
B & \rightarrow & C \\
E & \rightarrow & \textsf{id}
\end{array}
$$

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

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

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

(a)



(b)

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

Question 2. A *regular grammar* is a grammar whose productions are in one of the following two forms (where A an B are nonterminals and w represents a sequence of zero or more terminals):

$$\begin{array}{lcl} A & \rightarrow & w \\ A & \rightarrow & wB \end{array}$$

(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 and "ρ^{+}" indicates one 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 NFA that is not also a DFA.

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

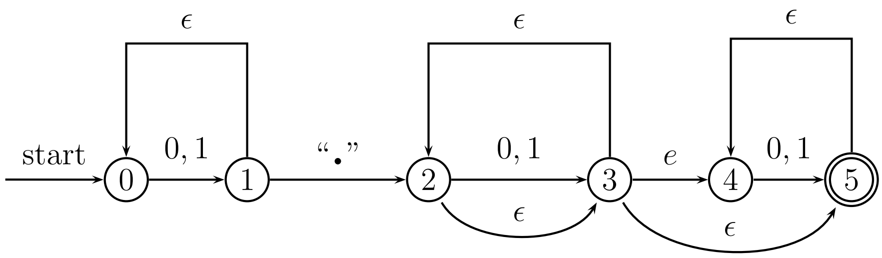
(a)

$$\begin{array}{lcl} S & \rightarrow & 0I \mid 1I \\ I & \rightarrow & 0I \mid 1I \mid .F \\ F & \rightarrow & \epsilon \mid 0F \mid 1F \mid eE \\ E & \rightarrow & 0E \mid 1E \mid 0 \mid 1 \end{array}$$

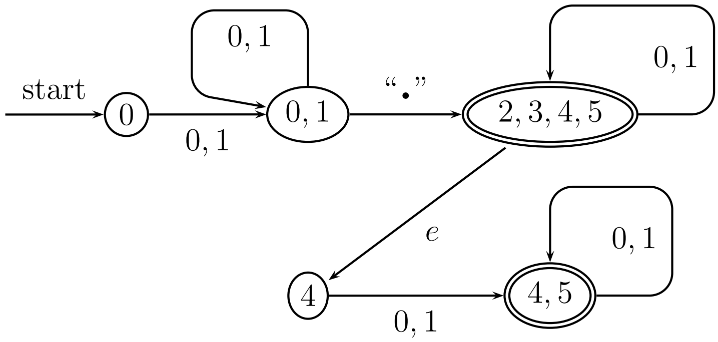
(b)

$$\begin{array}{lcl} S & \rightarrow & I.FE \\ I & \rightarrow & 0I \mid 1I \mid 0 \mid 1 \\ F & \rightarrow & I \mid \epsilon \\ E & \rightarrow & eI \mid \epsilon \end{array}$$

(c)



(d)



Question 3. 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:

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

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.

(a) an algorithm for computing Follow sets is given in lectures

(b)

$$
\begin{array}{lcl}
\mathsf{FOLLOW}(S) & = & \{\$,)\} \\
\mathsf{FOLLOW}(U) & = & \{\$,), +\} \\
\mathsf{FOLLOW}(V) & = & \{\$,)\} \\
\mathsf{FOLLOW}(W) & = & \{\$,), +\} \\
\mathsf{FOLLOW}(X) & = & \{\$,), +, *\}
\end{array}
$$

(c)

	n	+	*	(	)	$
S	UV			UV		
U	XW			XW		
V		+UV			ϵ	ϵ
W		ϵ	*XW		ϵ	ϵ
X	n			(S)		

Question 4. 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,

$$(\textbf{2}\ (\ \textbf{1 nil nil}\ )\ (\textbf{3 nil nil}\ )\ )$$

is a binary search tree, but

$$(\textbf{1}\ (\ \textbf{2 nil nil}\ )\ (\textbf{3 nil nil}\ )\ )$$

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?

(a)

Grammar Rule	Semantic Rules
$btree_1 \rightarrow (\ \textbf{num}\ btree_2\ btree_3\ )$	$btree_1.max = \text{MAX}(btree_3.max, \text{btree.val})$ $btree_1.min = \text{MIN}(btree_2.min, \text{btree.val})$ $btree_1.ordered = btree_2.ordered\ \&\&\ btree_3.ordered$ $\&\&\ btree_2.max <= \text{btree.val}$ $\&\&\ btree_3.min > \text{btree.val}$
$btree \rightarrow \textbf{nil}$	$btree.max = -\infty$ $btree.min = +\infty$ $btree.ordered = \text{true}$

(b) S-attributed

(c) all are synthesised attributes

Question 5. 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.

(a)

```
java VC.vc q5.vc
======= The VC compiler =======

Pass 1: Lexical and syntactic Analysis
Pass 2: Semantic Analysis
ERROR: 2(1)..2(13): "f" redeclared
ERROR: 6(9)..6(16): Attempt to reference variable "putIntLn" as a function.
Compilation was unsuccessful.
```

(b)

```
int f // function   level 1
inf f // parameter  level 2
int f // local      level 2
float f             level 3
int putIntLn        level 3
```

Question 6. 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.

(a)

```
  public Object visitSwitchIfStmt(SwitchIfStmt ast, Object o) {
    Frame frame = (Frame) o;
    String zeroLabel = frame.getNewLabel();
    String negLabel = frame.getNewLabel();
    String endLabel = frame.getNewLabel();
```

```
    ast.E.visit(this, o);

    emit(JVM.DUP);

    emit(JVM.IFLE, zeroLabel);
    emit(JVM.POP);
    ast.S1.visit(this, o);
    emit(JVM.GOTO, endLabel);

    emit(zeroLabel + ":");
    emit(JVM.IFLT, negLabel);
    ast.S2.visit(this, o);
    emit(JVM.GOTO, endLabel);

    emit(negLabel + ":");
    ast.S3.visit(this, o);
    emit(endLabel + ":");

    return null;
  }
```

(b) One possible Jasmin code sequence:

```
         iload_1
         iconst_1
         iadd
         dup
         ifle zeroL
         pop
         iconst_1
         istore_1
         goto endL
zeroL:
         iflt negL
         iconst_0
         istore_1
         goto endL
negL:
         iconst_M1
         istore_1
endL:
```