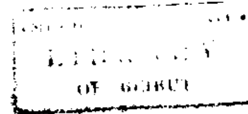
Faculty of Arts & Sciences
Department of Mathematics & Computer Sciences
CMPS 274—Compiler Construction
Fall 2003–2004
Thursday, January 29, 2004



Final Exam

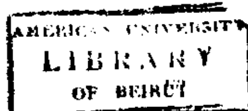
Name:______________________________ Student Id:____________________

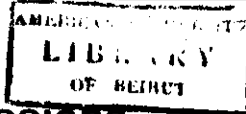
Signature:______________________________

Instructions

- There are 6 problems and 26 pages. Make sure you have all of them.
- The exam is closed book, closed notes, and closed neighbor.
- The questions are organized by topic and are **not sorted by difficulty**. Scan the whole exam before you start working.
- Your handwriting should be readable so it can be graded. Include all work or justification for partial credit.

Problem 1	**20**		
1.1		5	
1.2		5	
1.3		5	
1.4		5	
Problem 2	**35**		
2.1		10	
2.2		5	
2.3		10	
2.4		5	
2.5		5	
Problem 3	**20**		
3.1		5	
3.2		15	
Problem 4	**10**		
Problem 5	**25**		
5.1		10	
5.2		5	
5.3		10	
Problem 6	**15**		
6.1		5	
6.2		10	
Total	**125**		

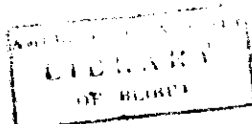

AMERICAN UNIVERSITY
LIBRARY
OF BEIRUT



Problem 1. Regular Expressions (20 = 5 + 5+ 5 + 5)

In Modula-3, integer literals have the form *base_digits* where the *base* component is optional and defaults to 10 (*i.e.*, integer literals represent decimal numbers by default). For example, the literals `2_11010`, `8_32`, `16_1A`, `10_26`, and `26` are all equivalent; their value is decimal 26.

1. Write a regular expression that generates the set of integer literals in Modula-3.

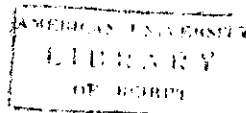


2. Write an equivalent regular grammar that generates the same language (*i.e.*, integer literals in Modula-3).

3. Augment this grammar with attributes and actions to compute the decimal value of a literal. Clearly define the attributes associated with symbols in the grammar and their types (inherited or synthesized), and describe any functions that you use.

4. What would happen to the lexical analysis task concerning Modula-3 integer literals if the pattern for these literals were switched to *digits_base*?

AMERICAN UNIVERSITY LIBRARY OF BEIRUT


AMERICAN UNIVERSITY LIBRARY OF BEIRUT

Problem 2. LL & LR Parsing (35 = 10 + 5 + 10 + 5 + 5)

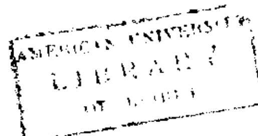
Consider the following simple context-free grammar:

1.	S	$\rightarrow$	ε
2.		|	**a**
3.		|	**b** S **d**
4.		|	**b** S **c** S **d**

1. Is this grammar LL(1)? If yes, prove it. Otherwise, explain why it is not and rewrite the grammar as an LL(1) grammar.

2. Is this grammar LR(0)? Explain.

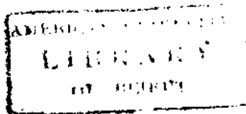
AMERICAN UNIVERSITY LIBRARY OF BEIRUT


AMERICAN UNIVERSITY LIBRARY OF BEIRUT

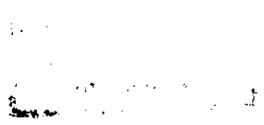
3. Is this grammar LR(1)? Answer this question by first constructing the LR(1) state machine along with its parsing table and then reasoning about it.

4. Is this grammar LALR(1)? Answer this question by first constructing the LALR(1) state machine along with its parsing table and then reasoning about it.

AMERICAN UNIVERSITY
LIBRARY
OF BEIRUT



5. Is the language modeled by this grammar a regular language? Explain.



AMERICAN UNIVERSITY LIBRARY OF BEIRUT

Problem 3. Attribute Grammars (20 = 5 + 15)

Consider the grammar below for arithmetic expressions. This grammar has built into it the precedence and associativity of operators to recognize infix expressions such as "(-2+3)*5;" and to generate their expression trees. These trees can then be traversed to yield the equivalent postfix expressions such as "2 - 3 + 5 *".

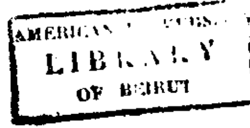
An alternative to using precedences and associativities is to require expressions to be *fully* parenthesized. As such, the previous expression takes the following form "(((- 2) + 3) * 5))"

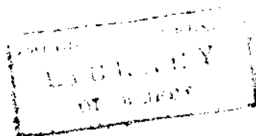
1.	S	$\rightarrow$	E ;
2.	E	$\rightarrow$	$E + T$
3.		|	T
4.	T	$\rightarrow$	$T * F$
5.		|	F
6.	F	$\rightarrow$	$-U$
7.		|	U
8.	U	$\rightarrow$	**num**
9.		|	(E)

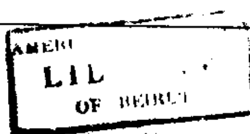
1. Add *synthesized* attributes and *actions* to generate the postfix equivalent of an infix expression. Describe the attributes and functions used in no more than 1–2 sentences each.

AMERICAN UNIVERSITY LIBRARY OF BEIRUT

2. Repeat the previous exercise to assemble a *minimally*-parenthesized infix expression. A minimally-parenthesized expression is one that does have any extra pair of parentheses beyond what is needed to override the precedences and associativities of operators. For example, the two expressions "`(-2+3)*5`" and "`(((-2) + 3) * 5))`" are equivalent with the first one having a minimal number of parentheses.

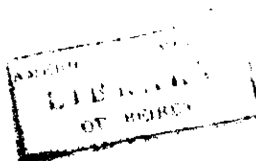

AMERICAN UNIVERSITY LIBRARY OF BEIRUT



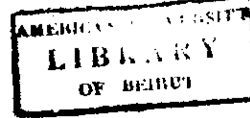


Problem 4. Type Checking (10)

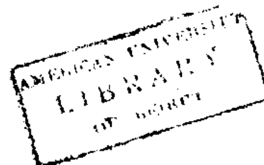
Appendix A (p. 19) reproduces the PCAT grammar as it appeared in the language handout. One of the productions in this grammar requires the compiler to infer the type of a variable (*i.e.*, the variable's type is not explicitly specified by the programmer and must be inferred by the compiler from its context). What rule is this and how does the compiler achieve its task? Illustrate this by augmenting one or more productions with actions.



Problem 5. Runtime Systems (25 = 10 + 5 + 10)

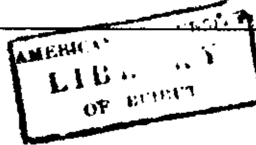

AMERICAN UNIVERSITY LIBRARY OF BEIRUT

1. Consider the PCAT-implementation of quick sort as given in Appendix B (page 21). What are the scoping levels encountered when parsing this program and what are the names declared at each of these levels?


AMERICAN UNIVERSITY LIBRARY OF BEIRUT

2. Consider the PCAT implementation of the solution of the Eight Queens problem as given in Appendix C (p. 23). Show the state of the runtime stack just after `queens (2)` has been invoked. List the names local to each activation frame and draw both the static and dynamic links.

3. Consider the productions used for defining record types. Identify these productions and augment them with code that computes the size of the record and the offsets of the fields in such a record.



Problem 6. Code Generation (15 = 5 + 10)

Quoting from the PCAT Language Handout:

11.7 Loop

`statement → LOOP {statement} END ';'`

The statement sequence is repeatedly executed. The only way to terminate the iteration is by executing an `EXIT` statement within the sequence but not inside any nested `WHILE`, `LOOP`, or `FOR`.

11.9 Exit

`statement → EXIT ';'`

Executing `EXIT` causes control to pass immediately to the next statement following the nearest enclosing `WHILE`, `LOOP` or `FOR` statement. If there is no such enclosing statement, the `EXIT` is illegal.

1. Draw the syntax tree and generate code for the following PCAT code segment:

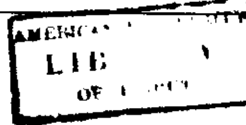
```
LOOP
    I := I + 1;
    IF (I >= 10) THEN
        EXIT;
    END;
END
```

2. Describe the interaction between these two statements when an EXIT statement appears in the body of a LOOP statement. Use code to illustrate a mechanism for handling this interaction.

Appendix A PCAT Grammar

	Symbol		Expansion
1.	program	→	PROGRAM IS body ';'
2.	body	→	{declaration} BEGIN {statement} END
3.	declaration	→	VAR {var-decl}
4.		|	TYPE {type-decl}
5.		|	PROCEDURE {procedure-decl}
6.	var-decl	→	ID { ',' ID } [':' typename] ':=' expression ';'
7.	type-decl	→	ID IS type ';'
8.	procedure-decl	→	ID formal-params [':' typename] IS body ';'
9.	typename	→	ID
10.	type	→	ARRAY OF typename
11.		|	RECORD component {component} END
12.	component	→	ID ':' typename ';'
13.	formal-params	→	'(' fp-section {';' fp-section } ')'
14.		|	'(' ')'
15.	fp-section	→	ID {',' ID} ':' typename
16.	statement	→	lvalue ':=' expression ';'
17.		|	ID actual-params ';'
18.		|	READ '(' lvalue {',' lvalue} ')' ';'
19.		|	WRITE write-params ';'
20.		|	IF expression THEN {statement}
21.		|	{ELSIF expression THEN {statement}}
22.		|	[ELSE {statement}] END ';'
23.		|	WHILE expression DO {statement} END ';'
24.		|	LOOP {statement} END ';'
25.		|	FOR ID ':=' expression TO expression [BY expression] DO {statement} END ';'
26.		|	EXIT ';'
27.		|	RETURN [expression] ';'
28.	write-params	→	'(' write-expr {',' write-expr } ')'
29.		|	'(' ')'
30.	write-expr	→	STRING
31.		|	expression

<table>
<tr><th></th><th>Symbol</th><th></th><th>Expansion</th></tr>
<tr><td>32.</td><td>expression</td><td>→</td><td>number</td></tr>
<tr><td>33.</td><td></td><td>|</td><td>lvalue</td></tr>
<tr><td>34.</td><td></td><td>|</td><td>'(' expression ')'</td></tr>
<tr><td>35.</td><td></td><td>|</td><td>unary-op expression</td></tr>
<tr><td>36.</td><td></td><td>|</td><td>expression binary-op expression</td></tr>
<tr><td>37.</td><td></td><td>|</td><td>ID actual-params</td></tr>
<tr><td>38.</td><td></td><td>|</td><td>ID record-inits</td></tr>
<tr><td>39.</td><td></td><td>|</td><td>ID array-inits</td></tr>
<tr><td>40.</td><td>lvalue</td><td>→</td><td>ID</td></tr>
<tr><td>41.</td><td></td><td>|</td><td>lvalue '[' expression ']'</td></tr>
<tr><td>42.</td><td></td><td>|</td><td>lvalue '.' ID</td></tr>
<tr><td>43.</td><td>actual-params</td><td>→</td><td>'(' expression {',' expression} ')'</td></tr>
<tr><td>44.</td><td></td><td>|</td><td>'(' ')'</td></tr>
<tr><td>45.</td><td>record-inits</td><td>→</td><td>'{' ID ':=' expression { ';' ID ':=' expression} '}'</td></tr>
<tr><td>46.</td><td>array-inits</td><td>→</td><td>'[<' array-init { ',' array-init } '>]'</td></tr>
<tr><td>47.</td><td>array-init</td><td>→</td><td>[expression OF] expression</td></tr>
<tr><td>48.</td><td>number</td><td>→</td><td>INTEGER | REAL</td></tr>
<tr><td>49.</td><td>unary-op</td><td>→</td><td>'+' | '-' | NOT</td></tr>
<tr><td>50.</td><td>binary-op</td><td>→</td><td>'+' | '-' | '*' | '/' | DIV | MOD | OR | AND | '>' | '<' | '=' | '>=' | '<=' | '<>'</td></tr>
</table>



Appendix B Quick Sort

```
1.   PROGRAM IS
2.   VAR N := 10;
3.   TYPE list IS ARRAY OF INTEGER;
4.   VAR a := list [< N OF 0 >];
5.
6.   PROCEDURE quicksort(a:list;m,n: INTEGER) IS
7.     VAR i :INTEGER := 0;
8.     PROCEDURE
9.     partition(y,z:INTEGER) : INTEGER IS
10.      VAR i := y;
11.          j := z + 1;
12.      PROCEDURE meet() IS
13.        PROCEDURE up() IS
14.          BEGIN
15.            i := i + 1;
16.            IF a[i] < a[y] THEN
17.              up();
18.            END;
19.          END;
20.        PROCEDURE down() IS
21.          BEGIN
22.            j := j - 1;
23.            IF a[j] > a[y] THEN
24.              down();
25.            END;
26.          END;
27.        BEGIN
28.          IF i < j THEN
29.             up();
30.             down();
31.             IF i < j THEN exchange(i,j); END;
32.             meet();
33.          END;
34.        END;
35.      BEGIN
36.        meet();
37.        exchange(y,j);
38.        RETURN j;
39.      END;
40.    exchange(p,q: INTEGER) IS
41.      VAR x := a[p];
42.      BEGIN
43.        a[p] := a[q];
44.        a[q] := x;
45.      END;
46.    BEGIN
47.      IF n > m THEN
48.        i := partition(m,n);
49.        quicksort(a,m,i-1);
50.        quicksort(a,i+1,n);
51.      END;
52.    END;
53.
54.  PROCEDURE readarray() IS
55.    VAR i := 0;
56.    BEGIN
57.      FOR i := 0 TO N-2 DO
```

```
58.         READ (a[i]);
59.       END;
60.       a[i] := -1000;
61.     END;
62.
63.   PROCEDURE writearray() IS
64.     VAR i := 0;
65.     BEGIN
66.       FOR i := 0 TO N-2 DO
67.         WRITE (a[i]);
68.       END;
69.     END;
70.
71.   BEGIN
72.     readarray();
73.     quicksort(a,0,N-2);
74.     writearray();
75.   END;
```

Appendix C Eight Queens

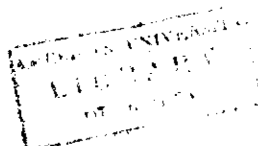
```
1.     PROGRAM IS
2.     TYPE BARRAY IS ARRAY OF BOOLEAN;
3.          IARRAY IS ARRAY OF INTEGER;
4.     VAR      i: INTEGER := 0;
5.              up, down := BARRAY [<15 OF TRUE>];
6.              rows := BARRAY [<15 OF TRUE>];
7.              x := IARRAY [<8 OF 0 >];
8.
9.     PROCEDURE print() IS
10.      BEGIN
11.        WRITE(x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7]);
12.      END;
13.
14.    PROCEDURE queens(c : INTEGER) IS
15.      VAR r := 0;
16.      BEGIN
17.        FOR r := 0 TO 7 DO
18.          IF rows[r] AND up[r-c+7] AND down[r+c] THEN
19.            rows[r] := FALSE;
20.            up[r-c+7] := FALSE;
21.            down[r+c] := FALSE;
22.            x[c] := r;
23.            IF c = 7 THEN print(); ELSE queens (c+1); END;
24.            rows[r] := TRUE; up[r-c+7] := TRUE; down[r+c] := TRUE;
25.          END;
26.        END;
27.      END;
28.
29.    BEGIN
30.      queens(0);
31.    END;
```

AMERICAN UNIVERSITY LIBRARY OF BEIRUT

Appendix D Quadruple Instructions

The table below lists the quadruple instructions used in Phase-5 of the course project.

Quadruple				Meaning
				Copying and Converting Scalars
COPYB	b1		b3	b3 := b1
COPYI	i1		i3	i3 := i1
COPYR	r1		r3	r3 := r1
COPYI2R	i1		r3	r3 := i1 converted to a real.
COPYR2I	r1		i3	i3 := r1 truncated to an integer.
				Reading and Writing Array Elements
LDB	i1	b2	b3	b3 := b2[i1]
LDI	i1	i2	i3	i3 := i2[i1]
LDR	i1	r2	r3	r3 := r2[i1]
STB	i1	b2	b3	b3[i1] := b2
STI	i1	i2	i3	i3[i1] := i2
STR	i1	r2	r3	r3[i1] := r2
				Integer and Character Arithmetic
ADDI	i1	i2	i3	i3 := i1 + i2
SUBI	i1	i2	i3	i3 := i1 – i2
MULI	i1	i2	i3	i3 := i1 * i2
DIVI	i1	i2	i3	i3 := i1 / i2
MODI	i1	i2	i3	i3 := i1 mod i2
NEGI	i1		i3	i3 := – i1
ADDR	r1	r2	r3	r3 := r1 + r2
SUBR	r1	r2	c3	r3 := r1 – r2
MULR	r1	r2	c3	r3 := r1 * r2
DIVR	r1	r2	c3	r3 := r1 / r2
NEGR	r1		c3	r3 := – r1
				Relational Operators
LTI	i1	i2	b3	b3 := i1 < i2
LTEQI	i1	i2	b3	b3 := i1 ≤ i2
EQI	i1	i2	b3	b3 := i1 = i2
NEQI	i1	i2	b3	b3 := i1 ≠ i2
LTR	c1	c2	b3	b3 := r1 < r2
LTEQR	c1	c2	b3	b3 := r1 ≤ r2
EQR	c1	c2	b3	b3 := r1 = r2
NEQR	c1	c2	b3	b3 := r1 ≠ r2
EQR	b1	b2	b3	b3 := b1 = b2
NEQR	b1	b2	b3	b3 := b1 ≠ b2
EQB	b1	b2	b3	b3 := b1 = b2
				Logical Operators
AND	b1	b2	b3	b3 := b1 and b2
OR	b1	b2	b3	b3 := b1 or b2
NOT	b1		b3	b3 := not b1
				Branches, Jumps, and Labels
BFALSE	b1	L2		branch to L2 if b1 is false
JUMP	L1			jump to L1
LABEL	L1			mark location of label L1



Quadruple				Meaning
				Procedures and Function Calls
PARAMB	b1			B1 is an argument in the next CALL, CALLB, CALLI, or CALLR quad instruction.
PARAMI	i1			i1 is an argument in the next CALL, CALLB, CALLI, or CALLR quad instruction.
PARAMR	c1			C1 is an argument in the next CALL, CALLB, CALLI, or CALLR quad instruction.
CALL	n	L2		call function L2 using the previous n arguments.
CALLB	n	L2	b3	call function L2 using the previous n arguments and put returned value in b3.
CALLI	n	L2	i3	call fucntion L2 using the previous n arguments and put returned value in i3.
CALLR	n	L2	c3	call function L2 using the previous n arguments and put returned value in c3.
				Procedure, Function, and Main Program Returns
PROC	n	L2		declare void function starting at label L2 with n parameters.
PROCB	n	L2		declare boolean function starting at label L2 with n parameters.
PROCI	n	L2		declare integer function starting at label L2 with n parameters.
PROCR	n	L2		declare real function starting at label L2 with n parameters.
LOCALS	n			allocate space for n local and temporary variables.
RET				return from a procedure.
RETB	b1			return from a Boolean function with b1 as the returned value.
RETI	i1			return from an integer function with i1 as the returned value.
RETR	c1			return from a char function with c1 as the returned value.
EXIT				exit from main program.

Appendix E Syntax Tree Node Types

```
#ifndef _NODE_TYPES_H_
#define _NODE_TYPES_H_

typedef enum
{
  NODE_ILLEGAL = -1,

  NODE_EMPTY = 0,              /* place holder */

  NODE_PROGRAM,                /* top level node */

  NODE_LIST,                   /* aggregation of children nodes */

  NODE_LIST_ID_INIT,           /* init of list of ids */

  NODE_ID,                     /* identifier */

  NODE_INIT_ARRAY_ELTS,        /* init of array elts (val, count) */
  NODE_INIT_REC_FIELD,         /* init of rec field (name, expr) */

  NODE_FUN,                    /* function or procedure */

  NODE_STMT_ASSIGN,            /* lvalue, expr */
  NODE_LVALUE,                 /* id, accessor nodes */
  NODE_LVALUE_ARRAY_ELT,       /* index expr */
  NODE_LVALUE_REC_FIELD,       /* field name */

  NODE_FUN_CALL,               /* fn/proc call: id, expr list */

  NODE_STMT_READ,              /* lvalue list */

  NODE_STMT_WRITE,             /* expr list */

  NODE_STMT_IF,                /* list cond nodes, else-node (stmt) */
  NODE_IF_COND,                /* expr, stmt pair */

  NODE_STMT_WHILE,             /* expr, stmt list */
  NODE_STMT_LOOP,              /* stmt list */
  NODE_STMT_FOR,               /* id, expr (init), expr (limit),
                                  expr (step--may be NODE_EMPTY),
                                  stmt list */
  NODE_STMT_EXIT,
  NODE_STMT_RETURN,            /* expr--may be NODE_EMPTY */

  NODE_EXPR,

  NODE_LIT_INT,
  NODE_LIT_REAL,
  NODE_LIT_BOOL,
  NODE_LIT_STRING,

  NODE_SENTINEL                /* guard for defensive prog. */
}
node_type_t;
```