UNIVERSITY OF MALTA

BOARD OF STUDIES FOR INFORMATION TECHNOLOGY

Department of Computer Science and A.I.

B.Sc. I.T. (Hons) Year II

January 2002 Examination Session

Date???.

Time???

CSM201 – Compiling Techniques

Read the following instructions carefully:

- This paper contains **Six (6)** questions.
- You should attempt any **Four (4)** questions.
- The total mark for this paper is **200**, each question holds **50 marks**.
- Calculators are NOT allowed.
- Algorithms may be expressed in any unambiguous way such as pseudo-code, flowcharts or Pascal. Minor syntactical errors and omissions will not be taken into account.
- This document contains **Eight (8)** pages, including this one.

Question 1.

a) Why are variable declarations used in programming languages?

(4 Marks)

b) Explain what is meant by the Object Description Phase.

(6 Marks)

c) When are variables allowed to be re-declared? In such cases what other information must be stored in the Symbol Table?

(8 Marks)

d) Write the BNF grammar for BASIC variable declarations. Also, in pseudo-code, write the function implementations necessary to parse such declarations and place the required information in the Symbol Table.

Notes:

- You may assume that the *Declare* Object Description Phase function exists without implementing it.
- Examples of variable declarations in BASIC are:

```
Dim counter As Integer, pi As Double
Dim myName As String
```

- The types allowed in the language are 'Integer', 'Double' and 'String'.

(32 Marks)

Question 2.

a) State whether the following statement is true or false and give a brief explanation and example to support your answer. *'If a grammar for a language is ambiguous, then a valid program may have more than one parse tree.'*

(10 Marks)

b)

i. Explain why top-down parsers cannot handle left recursion.

(4 Marks)

ii. How should the rules:

$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

be rewritten to eliminate left recursion.

(10 Marks)

iii. Eliminate left recursion from the following two grammars:

Grammar 1:

S → Xx | y

X → Xp | Sq | ε

Grammar 2:

E → E + T | T

T → T + F | F

F → (E) | Identifier

(16 Marks)

c)

i. Briefly explain what left factoring is.

ii. Why should left factoring be eliminated in order to construct predictive top down parsers?

iii. Rewrite the following two productions without left factoring:

$A \rightarrow \alpha\beta_1$

$A \rightarrow \alpha\beta_2$

(10 Marks)

Question 3.

a) Consider the following grammar for arithmetic expressions:

```
E → E + T
E → T
T → T * F
T → F
F → ( E )
F → ID
```

Use the following table to show how an LR Parser would parse the expression: `id * (id + id)`:

State	Action Table						GoTo Table		
	ID	**+**	*****	**(**	**)**	**EOF**	**E**	**T**	**F**
0	S5			S4			1	2	3
1		S6				Acpt			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	R7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Where:

S*n* means: Shift state *n* and read the new token.
R*k* means: Reduce by production *k*.
Blank entries denote an error.

(You don't need to reproduce the above table in your answer booklet)

(24 Marks)

b) Explain what a shift-reduce conflict is. Illustrate with an example.

(8 Marks)

c) What is the difference between static and dynamic checks made by a compiler? Illustrate with examples.

(8 Marks)

d) What is the equivalent syntax tree and three-address-code for the following assignment statement:

```
a := b * -c + b * -c
```

(10 Marks)

Question 4.

a) Why is it desirable to split a compiler into the front and backend stages? Which compiling phase (e.g. scanning, parsing, etc…) generally goes into each of the two stages?

(10 Marks)

b) The following pseudo-code program reads an expression from the user, scans it, parses it and evaluates it. Draw a possible activation tree given the expression `(3 + 4) * 5`:

```
Program Evaluate
Var e: String;
Var r: NodeType;

{Read an expression from the user}
Procedure ReadExpression;
Begin ... End;

{Scan and parse the expression return the
 pointer to the root node or -1 on error}
Procedure ProcessExpression(exp: String):NodeType;
Begin ... End;

Function Eval(root: NodeType):Integer;
Begin
  If root.Type = Constant Then
    Function = Root.Value;
  Else If Root.Type = Plus Then
    Function = Eval(Root.LeftChild) + Eval(Root.RightChild);
  Else If Root.Type = Times Then
    Function = Eval(Root.LeftChild) * Eval(Root.RightChild);
  Else If Root.Type = Minus Then
    Function = Eval(Root.LeftChild) - Eval(Root.RightChild);
  Else If Root.Type = Division Then
    Function = Eval(Root.LeftChild) / Eval(Root.RightChild);
End;

Begin
  e := ReadExpression();
  r := ProcessExpression(e);

  If (r <> -1) Then
    WriteLn Eval(r);
End.
```

(12 Marks)

c) Briefly explain what a control stack is in relation to procedure activations.

(6 Marks)

d) What is an activation record? What information is generally stored in it?

(10 Marks)

e) Write down the types of grammars in the Chomsky hierarchy giving some detail on each type.

(12 Marks)

Question 5.

a) Describe briefly the general format of a FLEX program and explain what each section is used for.

(8 Marks)

b) Consider the following two FLEX programs (line numbers are for illustrative purposes only):

Example 1:

```
01    %{
02    char name[50];
03    %}
04    %%
05    username                printf("%s",name);
06    %%
07    int main(int argc, char* argv[])
08    {
09      if (argc < 2)
10      {
11        printf("Usage: scanner <name>");
12        exit(1);
13      }
14      strcopy(name, argv[1]);
15      yylex();
16    }
```

For example 1,

i. Explain what the FLEX program does in general.
ii. Explain what is happening on lines 01 to 03, line 05 and lines 07 to 16.

Example 2:

```
01    %{
02    int num_lines = 0, num_chars = 0;
03    %}
04    %%
05    \n          ++num_lines; ++num_chars;
06    .           ++num_chars;
07    %%
08    int main(void)
09    {
09      yylex();
10      printf("Lines: %d\n", num_lines);
11      printf("Chars: %d\n", num_chars);
12    }
```

For example 2,

iii. Explain what the FLEX program is doing in general.

iv. Explain what is happening on lines 01 to 03, lines 05 to 06 and lines 08 to 12.

(16 Marks)

c) Briefly describe the general format of a BISON program and explain what each section is used for.

(8 Marks)

d) Describe the following:

i. Leftmost and rightmost derivations.

ii. The relationship in terms of expressive power and design complexity of LALR(k), SLR(k) and LR(k) parsers.

iii. The checks a compiler has to make when examining the statement:

```
x := min(p,q) + r;
```

(18 Marks)

Question 6.

a) Briefly describe two ways in which the process of code optimization can be approached.

(4 Marks)

b) Explain the following:

i. Elimination of common sub-expressions,

ii. Elimination of copy propagation,

iii. Dead-Code Elimination,

iv. Loop Optimisation and Code Motion.

(16 Marks)

c) Rewrite the following fragment of code after it is optimised by eliminating copy propagation and dead code?

```
01    x = counter
02    y = limit
03    arr[q] = z

04    for i =  1 to z
05         arr2[i] = x
06    next

07    arr[q+1] = x + y
```

Note: Line numbers are for illustrative purposes only and every step should be properly explained.

(8 Marks)

d) A simple translation scheme is associating a 'template' to each three-address-code statement. Explain with the help of an example why this technique is likely to produce inefficient code. You may use the following template for statements of the form `p = q + r` to help you in your example:

```
MOV q, R0       ; move value of q in register R0
ADD r, R0       ; add value of r to register R0
MOV R0, p       ; move value of R0 in p
```

(8 Marks)

e) Write short notes on the following:

i. Assemblers,
ii. Cross Compilers,
iii. Interpreters.

(6 Marks)

f) Explain briefly with the help of an example what backtracking is in top-down parsing.

(8 Marks)