RHODES UNIVERSITY

November Examinations - 2012

Computer Science 301 - Paper 2

Examiners:
Prof P.D. Terry
Prof D.G. Kourie

Time 4 hours
Marks 180
Pages 16 (please check!)

Answer all questions. Answers may be written in any medium except red ink.

A word of advice: The influential mathematician R.W. Hamming very aptly and succinctly professed that "the purpose of computing is insight, not numbers".

Several of the questions in this paper are designed to probe your insight - your depth of understanding of the important principles that you have studied in this course. If, as we hope, you have gained such insight, you should find that the answers to many questions take only a few lines of explanation. Please don't write long-winded answers - as Einstein put it "Keep it as simple as you can, but no simpler".

Good luck!

(For the benefit of future readers of this paper, various free information was made available to the students 24 hours before the formal examination. This included an augmented version of "Section C" - a request to devise a simple calculator based on the Parva Virtual Machine interpreter system studied in the course. Some 16 hours before the examination a complete grammar for a calculator and other support files for building this system were supplied to students, along with an appeal to study this in depth (summarized in "Section D"). During the examination, candidates were given machine executable versions of the Coco/R compiler generator, the files needed to build the basic systems, access to a computer, and machine readable copies of the questions.)

Section A: Conventional questions [100 marks]

QUESTION A1 **[10 marks]**

(*Compiler structure)* A syntax-directed compiler usually incorporates various components, of which the most important are the scanner, parser, constraint analyser, error reporter, code generator, symbol table handler and I/O routines. Draw a diagram indicating the dependence of these components on one another, and in particular the dependence of the central syntax analyser on the other components. Also indicate which components constitute the *front end* and which the *back end* of the compiler. [10 marks]

QUESTION A2 **[16 marks]**

(Recursive descent parsers) The following Cocol grammar describes the form of an index to a textbook and should be familiar from the practical course.

```
COMPILER Index $CN
/* Grammar describing index in a book
   P.D. Terry, Rhodes University, 2012 */

CHARACTERS
/* Notice the careful and unusual choice of character sets */
  digit      = "0123456789" .
  startword  = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz('" + '"' .
  inword     = startword + digit + "-+)" .
  eol        = CHR(10) .

TOKENS
/* Notice the careful and unusual definition for word */
  word       = startword { inword } .
  number     = digit { digit } .
  EOL        = eol .

IGNORE CHR(0) .. CHR(9) + CHR(11) .. CHR(31)

PRODUCTIONS
  Index      = { Entry } EOF .
  Entry      = Key References EOL .
  Key        = word { "," word | word } .
  References = DirectRefs | CrossRef .
  DirectRefs = PageRefs { "," PageRefs  } .
  PageRefs   = [ "Appendix" ] number [ "-" number ] .
  CrossRef   = "--" "see" Key .
END Index .
```

Assume that you have available a suitable scanner method called `getSym` that can recognize the terminals of *Index* and classify them appropriately as members of the following enumeration

```
EOFSym, noSym, EOLSym, wordSym, numberSym, appendixSym, commaSym,
dashSym, dashDashSym, seeSym
```

Develop a hand-crafted recursive descent parser for recognizing the index of a book based on the grammar above. *(Your parser can take drastic action if an error is detected. Simply call methods like* `accept` *and* `abort` *to produce appropriate error messages and then terminate parsing. You are not required to write any code to implement the* `getSym`*,* `accept` *or* `abort` *methods.)* [16 marks]

QUESTION A3 **[8 + 6 + 2 + 3 + 2 + 3 = 24 marks]**

(Grammars) Formally, a grammar G is defined by a quadruple $\{ N, T, S, P \}$ with the four components

(a) N - a finite set of **non-terminal** symbols,
(b) T - a finite set of **terminal** symbols,
(c) S - a special **goal** or **start** or **distinguished** symbol,
(d) P - a finite set of **production rules** or, simply, **productions**.

where a production relates to a pair of strings, say α and β, specifying how one may be transformed into the other:

$\alpha \rightarrow \beta$ where $\alpha \in (N \cup T)^* N (N \cup T)^*$, $\beta \in (N \cup T)^*$

and we can then define the language $L(G)$ produced by the grammar G by the relation

$L(G) = \{ w \mid S \Rightarrow^* w \wedge w \in T^* \}$

(a) In terms of this style of notation, define **precisely** (*that is to say, mathematically; we do not want a long-winded essay*) what you understand by [2 marks each]

(1) FIRST(σ) where $\sigma \in (N \cup T)^+$
(2) FOLLOW(A) where $A \in N$
(3) A context-free grammar
(4) A reduced grammar

(b) In terms of the notation here, concisely state the two rules that must be satisfied by the productions of a context-free grammar in order for it to be classified as an LL(1) grammar. [6 marks]

(c) Describe the language generated by the following grammar, using English or simple mathematics. [2 marks]

$S \rightarrow A\ B$
$A \rightarrow a\ A \mid a$
$B \rightarrow b\ B\ c \mid bc$

(d) Is the grammar in (c) an LL(1) grammar? If not, why not, and can you find an equivalent grammar that *is* LL(1)? [3 marks]

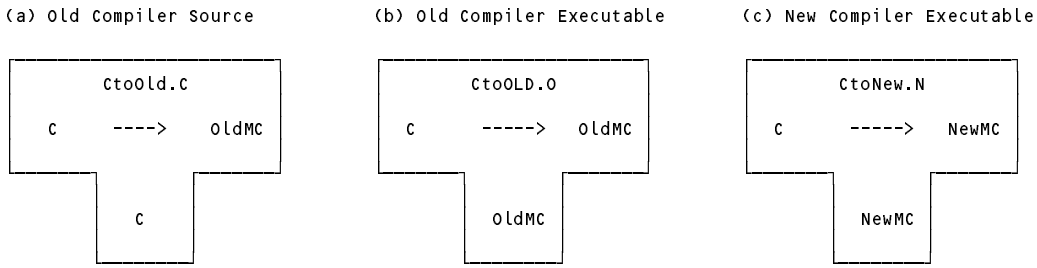
(e) The following grammar describes strings comprised of an equal number of the characters a and b, terminated by a period, such as *aababbba*. Is this an LL(1) grammar? Explain. [2 marks]

$S \rightarrow B\ .$
$B \rightarrow a\ B\ b\ B \mid b\ B\ a\ B \mid \varepsilon$

(f) "Keep it as simple as you can, but no simpler" said Einstein. Strings that might be members of the language of (e) can surely be accepted or rejected by a very simple algorithm, without recourse to the direct use of a grammar. Give such an algorithm, using a Java-like notation. [3 marks]

QUESTION A4 **[16 marks]**

(T diagrams) The process of "porting" a compiler to a new computer incorporates a *retargetting* phase (modifying the compiler to produce target code for the new machine) and a *rehosting* phase (modifying the compiler to run on the new machine). Illustrate these two phases for porting a C compiler, by drawing a set of T diagrams. Assume that you have available the compilers (a) and (b) below and wish to produce compiler (c). [16 marks]



(You may conveniently make use of the outline T-diagrams at the end of this paper; complete these and attach the page to your answer book.)

QUESTION A5 **[18 marks]**

(Attributed Grammars in Cocol) XML (eXtensible Markup Language) is a powerful notation for marking up data documents in a portable way. XML code looks rather like HTML code, but has no predefined tags. Instead a user can create customized markup tags, similar to those shown in the following extract.

```
<!-- comment - a sample extract from an XML file -->
<personnel>
  <entry>
    <name>John Smith</name>
  </entry>
  <entry_2>
    <name>Joan Smith</name>
    <address/>
    <gender>female</gender>
  </entry_2>
</personnel>
```

An *element* within the document is introduced by an *opening tag* (like `<personnel>`) and terminated by a *closing tag* (like `</personnel>`), where the obvious correspondence in spelling is required. The name within a tag must start with a letter or lowline character (_), and may then incorporate letters, lowlines, digits, periods or hyphens before it is terminated with a > character. Between the opening and closing tags may appear a sequence of free format *text* (like `John Smith`) and further *elements* in arbitrary order. The free format text may not contain a < character - this is reserved for the beginning of a tag. An *empty element* - one that has no internal members - may be terminated by a closing tag, or may be denoted by an *empty tag* - an opening tag that is terminated by `/>` (as in `<address/>` in the above example). Comments may be introduced and terminated by the sequences `<!--` and `-->` respectively, but may not contain the pair of characters `--` internally (as exemplified above).

Develop a Cocol specification, incorporating suitable `CHARACTER` sets and `TOKEN` definitions for

(a) opening tags,
(b) closing tags,
(c) empty tags,
(d) free format text

and give `PRODUCTIONS` that describe complete documents like the one illustrated. You may do this conveniently on the page supplied at the end of the examination paper.

Tags must be properly matched. A document like the following must be rejected

```
<bad.Tag>
   This is valid internal text
   <okayTag>
     More internal stuff
   </okayTag>
</badTag>  <!-- badTag should have been written as bad.Tag -->
```

Show how your grammar should be attributed to perform such checks. [18 marks]

Incidentally, it should be noted that the full XML specification defines far more features than those considered here!

QUESTION A6 **[16 marks]**

(Code generation) A BYT (Bright Young Thing) has been nibbling away at writing extensions to her first Parva compiler. It has been suggested that a function that will return the maximum element from a variable number of arguments would be a desirable addition, one that might form part of an expression as in

```
a = max(x, y, z) + 5 - max(a, max(c, d));
```

and that this could be achieved by adding a keyword `max`, extending the production for a *Factor* in a

fairly obvious way, and adding a suitable opcode to the PVM. To refresh your memory, the production for *Factor* in the simple Parva compiler is defined as follows:

```
Factor<out int type>                (. int value = 0;
                                       type = Types.noType;
                                       int size;
                                       DesType des;
                                       ConstRec con; .)
=    Designator<out des>            (. type = des.type;
                                       switch (des.entry.kind) {
                                         case Kinds.Var:
                                           CodeGen.dereference();
                                           break;
                                         case Kinds.Con:
                                           CodeGen.loadConstant(des.entry.value);
                                           break;
                                         default:
                                           SemError("wrong kind of identifier");
                                           break;
                                       } .)
   | Constant<out con>              (. type = con.type;
                                       CodeGen.loadConstant(con.value); .)
   | "new" BasicType<out type>      (. type++; .)
     "[" Expression<out size>       (. if (!isArith(size))
                                         SemError("array size must be integer");
                                       CodeGen.allocate(); .)
     "]"
   | "!" Factor<out type>           (. if (!isBool(type)) SemError("boolean operand needed");
                                       else CodeGen.negateBoolean();
                                       type = Types.boolType; .)
   | "(" Expression<out type> ")"
.
```

while a sample of the opcodes in the PVM that deal with simple arithmetic and logic arithmetic are interpreted with code of the form

```
case PVM.ldc:           // push constant value
  push(next());
  break;
case PVM.add:           // integer addition
  tos = pop(); push(pop() + tos);
  break;
case PVM.sub:           // integer subtraction
  tos = pop(); push(pop() - tos);
  break;
case PVM.not:           // logical negation
  push(pop() == 0 ? 1 : 0);
  break;
```

Suggest, in as much detail as time will allow, how the *Factor* production and the interpreter would need to be changed to support this language extension. Allow your `max` function to take one or more arguments, and ensure that it can only be applied to arithmetic arguments. Assume that a suitable `CodeGen` routine can be introduced to generate any new opcodes introduced. [16 marks]

Section B [80 marks]

Please note that there is no obligation to produce a machine readable solution for this section. Coco/R and other files are provided so that you can enhance, refine, or test your solution if you desire. If you choose to produce a machine readable solution, you should create a working directory, unpack EXAM.ZIP, modify any files that you like, and then copy all the files back onto an exam folder on the network.

Your answers to the following questions should, whenever possible, take the form of actual code, and not simply vague discussion.

QUESTION B7 **[4 marks]**

After studying the `CalcPVM` grammar that has been provided, you should realize that the compiler derived from it takes a very casual approach to ensuring that the static semantics of the language are obeyed. What, in general, do you understand by the concept of *static semantics* and what is the difference between *static semantics* and *syntax*?

QUESTION B8 **[6 marks]**

The grammar makes no provision for checking that an expression on the right hand side of an assignment must be of integer type, that the value returned by a function must be of integer type, and that the arguments used in a function call must be of integer type. Modify it so that these checks will be carried out.

QUESTION B9 **[3 + 3 = 6 marks]**

(a) Can you use constants as arguments in a function call - for example

```
Fun(x, y) returns x + y;
write(Fun(200, 400));
```

Explain! If you think it *is* possible, might there be any constants that you could not use in this way? Explain.

(b) Discuss (giving reasons) whether or not the above definition of `Fun(x,y)` can be followed by calls like

```
a = Fun(y, x);      // parameters seem to have been inverted
```

or

```
a = Fun(x, x);      // the same parameter has been used twice
```

QUESTION B10 **[8 + 4 = 12 marks]**

(a) The system as supplied makes no attempt to verify that the number of arguments supplied in a function call is the same as the number of parameters specified in the function definition. Remedy this deficiency.

(b) What might be the run-time effect of omitting this compile-time check if, for example, you compiled and then ran a program of the form

```
Fun(x, y) returns x + y;      // two parameters
...
a = Fun(x) + Fun(x, y, z);    // one and then three arguments
```

QUESTION B11 **[3 + 8 = 11 marks]**

(a) Should it be regarded as an error if a function appears not to refer to, or to use, some of its parameters - for example

```
Fun(x, y, z)  returns x;
```

Justify your answer.

(b) If you wished to alert a user to this situation, show the changes to the code needed to check for it.

QUESTION B12 **[4 + 4 = 8 marks]**

Incorporate code that allows the system to detect and report erroneous function definitions like

```
Fun(x, y)  returns x + y + z;
```

and

```
Fun(x, x)  returns x * x;
```

QUESTION B13 **[3 + 3 + 3 = 9 marks]**

(a) Under what conditions can one function call another - in other words, can one write code like

```
Double(x)   returns 2 * x;
Triple(x)   returns x + Double(x);
```

(b) Given that there may be conditions in which this is possible, what would be the effect of using the supplied system with the following code

```
Silly(x)   returns 2 * Silly(x);
write ( Silly(x) );
```

(c) What modification to the supplied system will prevent that `Silly` sort of behaviour? Explain.

QUESTION B14 **[14 marks]**

You may have wondered why it was suggested that you leave all the machinery for evaluating Boolean expressions in place when the whole system seems geared to integers only. But consider - if we redefine the form of a function specification to be

```
FunDefinition
=  "(" ParamList ")"
     "returns" Definition .

Definition
=    Expression ";"
   |  "if" "(" Expression ")" [ "returns" ] Definition "else" [ "returns" ] Definition .
```

then we should presumably be allowed to write a recursive definition like

```
Factorial(n) returns if (n <= 0) returns 1; else returns n * Factorial(n - 1);
```

Add the necessary extensions to the system to incorporate this powerful feature.

QUESTION B15 **[4 + 6 = 10 marks]**

(a) What would be the effect of replacing the syntax suggested in question B14 by

```
FunDefinition
=  "(" ParamList ")"
     "returns" Definition .

Definition
=    Expression ";"
   | "if" "(" Expression ")" Expression ";" "else" Expression ";" .
```

(b) What would be the effect of replacing the syntax suggested in question B14 by

```
FunDefinition
=  "(" ParamList ")"
     "returns" Definition .

Definition
=    Expression ";"
   | "if" "(" Expression ")" Definition [ "else" Definition ] .
```

END OF EXAMINATION QUESTIONS

Section C

(Summary of free information made available to the students 24 hours before the formal examination.)

Candidates were provided with the basic ideas, and were invited to extend a version of the supplied calculator grammar so as to define simple "one-liner" functions that could be incorporated into the evaluation of expressions.

It was pointed out that the PVM supplied to them incorporated the codes needed to support function calls, on the lines discussed in chapter 14 of the text book. Although there had been some discussion of the mechanisms in lectures, there had been no room to explore these in the practical course. However, a CFG for the Parva language (but devoid of attributes and actions) had been extended in an earlier practical, so candidates who had completed that exercise should surely have seen the connection. A skeleton symbol table handler was provided, as was a code generator virtually identical to the one they had seen previously.

They were provided with an exam kit for Java or C#, containing the Coco/R system, along with a suite of simple, suggestive test programs. They were told that later in the day some further ideas and hints would be provided.

Section D

(Summary of free information made available to the students 16 hours before the formal examination.)

A complete grammar for a rudimentary solution to the exercise posed earlier in the day was supplied to candidates in a later version of the examination kit. They were encouraged to study it in depth and warned that questions in Section B would probe this understanding; few hints were given as to what to expect, other than that they might be called on to comment on the solution, and perhaps to make some modifications and extensions. They were also encouraged to spend some time thinking how any other ideas they had during the earlier part of the day would need modification to fit in with the solution kit presented to them.

Free information

Summary of useful library classes

The following summarizes the simple set handling and I/O classes that have been useful in the development of applications using the Coco/R compiler generator.

```
class IntSet  { // simple set handling routines - There are matching versions for C#
  public IntSet()
  public IntSet(int ... members)
  public Object clone()
  public IntSet copy() {
  public boolean equals(Symset s)
  public void incl(int i)
  public void excl(int i)
  public boolean contains(int i)
  public boolean isEmpty()
  public int members()
  public IntSet union(IntSet s)
  public IntSet intersection(IntSet s)
  public IntSet difference(IntSet s)
  public IntSet symDiff(IntSet s)
  public void write()
  public String toString()
} // IntSet

public class OutFile {  // text file output - There are matching versions for C#
  public static OutFile StdOut
  public static OutFile StdErr
  public OutFile()
  public OutFile(String fileName)
  public boolean openError()
```

```
  public void write(String s)
  public void write(Object o)
  public void write(byte o)
  public void write(short o)
  public void write(long o)
  public void write(boolean o)
  public void write(float o)
  public void write(double o)
  public void write(char o)
  public void writeLine()
  public void writeLine(String s)
  public void writeLine(Object o)
  public void writeLine(byte o)
  public void writeLine(short o)
  public void writeLine(int o)
  public void writeLine(long o)
  public void writeLine(boolean o)
  public void writeLine(float o)
  public void writeLine(double o)
  public void writeLine(char o)
  public void write(String o,  int width)
  public void write(Object o,  int width)
  public void write(byte o,    int width)
  public void write(short o,   int width)
  public void write(int o,     int width)
  public void write(long o,    int width)
  public void write(boolean o, int width)
  public void write(float o,   int width)
  public void write(double o,  int width)
  public void write(char o,    int width)
  public void writeLine(String o,  int width)
  public void writeLine(Object o,  int width)
  public void writeLine(byte o,    int width)
  public void writeLine(short o,   int width)
  public void writeLine(int o,     int width)
  public void writeLine(long o,    int width)
  public void writeLine(boolean o, int width)
  public void writeLine(float o,   int width)
  public void writeLine(double o,  int width)
  public void writeLine(char o,    int width)
  public void close()
} // OutFile

public class InFile {    // text file input - There are matching versions for C#
  public static InFile StdIn
  public InFile()
  public InFile(String fileName)
  public boolean openError()
  public int errorCount()
  public static boolean done()
  public void showErrors()
  public void hideErrors()
  public boolean eof()
  public boolean eol()
  public boolean error()
  public boolean noMoreData()
  public char readChar()
  public void readAgain()
  public void skipSpaces()
  public void readLn()
  public String readString()
  public String readString(int max)
  public String readLine()
  public String readWord()
  public int readInt()
  public int readInt(int radix)
  public long readLong()
  public int readShort()
  public float readFloat()
  public double readDouble()
  public boolean readBool()
  public void close()
} // InFile
```

Strings and Characters in Java

The following rather meaningless code illustrates various of the string and character manipulation methods that are available in Java and which are useful in developing translators.

```
import java.util.*;

    char c, c1, c2;
    boolean b, b1, b2;
    String s, s1, s2;
    int i, i1, i2;

    b = Character.isLetter(c);                // true if letter
    b = Character.isDigit(c);                 // true if digit
    b = Character.isLetterOrDigit(c);         // true if letter or digit
    b = Character.isWhitespace(c);            // true if white space
    b = Character.isLowerCase(c);             // true if lowercase
    b = Character.isUpperCase(c);             // true if uppercase
    c = Character.toLowerCase(c);             // equivalent lowercase
    c = Character.toUpperCase(c);             // equivalent uppercase
    s = Character.toString(c);                // convert to string
    i = s.length();                           // length of string
    b = s.equals(s1);                         // true if s == s1
    b = s.equalsIgnoreCase(s1);               // true if s == s1, case irrelevant
    i = s1.compareTo(s2);                     // i = -1, 0, 1 if s1 < = > s2
    s = s.trim();                             // remove leading/trailing whitespace
    s = s.toUpperCase();                      // equivalent uppercase string
    s = s.toLowerCase();                      // equivalent lowercase string
    char[] ca = s.toCharArray();              // create character array
    s = s1.concat(s2);                        // s1 + s2
    s = s.substring(i1);                      // substring starting at s[i1]
    s = s.substring(i1, i2);                  // substring s[i1 ... i2-1]
    s = s.replace(c1, c2);                    // replace all c1 by c2
    c = s.charAt(i);                          // extract i-th character of s
//    s[i] = c;                               // not allowed
    i = s.indexOf(c);                         // position of c in s[0 ...
    i = s.indexOf(c, i1);                     // position of c in s[i1 ...
    i = s.indexOf(s1);                        // position of s1 in s[0 ...
    i = s.indexOf(s1, i1);                    // position of s1 in s[i1 ...
    i = s.lastIndexOf(c);                     // last position of c in s
    i = s.lastIndexOf(c, i1);                 // last position of c in s, <= i1
    i = s.lastIndexOf(s1);                    // last position of s1 in s
    i = s.lastIndexOf(s1, i1);                // last position of s1 in s, <= i1
    i = Integer.parseInt(s);                  // convert string to integer
    i = Integer.parseInt(s, i1);              // convert string to integer, base i1
    s = Integer.toString(i);                  // convert integer to string

    StringBuffer                              // build strings (Java 1.4)
      sb = new StringBuffer(),                //
      sb1 = new StringBuffer("original");     //
    StringBuilder                             // build strings (Jaba 1.5 and 1.6)
      sb = new StringBuilder(),               //
      sb1 = new StringBuilder("original");    //
    sb.append(c);                             // append c to end of sb
    sb.append(s);                             // append s to end of sb
    sb.insert(i, c);                          // insert c in position i
    sb.insert(i, s);                          // insert s in position i
    b = sb.equals(sb1);                       // true if sb == sb1
    i = sb.length();                          // length of sb
    i = sb.indexOf(s1);                       // position of s1 in sb
    sb.delete(i1, i2);                        // remove sb[i1 .. i2-1]
    sb.deleteCharAt(i1);                      // remove sb[i1]
    sb.replace(i1, i2, s1);                   // replace sb[i1 .. i2-1] by s1
    s = sb.toString();                        // convert sb to real string
    c = sb.charAt(i);                         // extract sb[i]
    sb.setCharAt(i, c);                       // sb[i] = c

    StringTokenizer                           // tokenize strings
      st = new StringTokenizer(s, ".,");      // delimiters are . and ,
      st = new StringTokenizer(s, ".,", true); // delimiters are also tokens
      while (st.hasMoreTokens())              // process successive tokens
        process(st.nextToken());

    String[]                                  // tokenize strings
      tokens = s.split(".;");                 // delimiters are defined by a regexp
    for (i = 0; i < tokens.length; i++)       // process successive tokens
      process(tokens[i]);
```

Strings and Characters in C#

The following rather meaningless code illustrates various of the string and character manipulation methods that are available in C# and which will be found to be useful in developing translators.

```
using System.Text;   // for StringBuilder
using System;        // for Char

    char c, c1, c2;
    bool b, b1, b2;
    string s, s1, s2;
    int i, i1, i2;

    b = Char.IsLetter(c);                    // true if letter
    b = Char.IsDigit(c);                     // true if digit
    b = Char.IsLetterOrDigit(c);             // true if letter or digit
    b = Char.IsWhiteSpace(c);                // true if white space
    b = Char.IsLower(c);                     // true if lowercase
    b = Char.IsUpper(c);                     // true if uppercase
    c = Char.ToLower(c);                     // equivalent lowercase
    c = Char.ToUpper(c);                     // equivalent uppercase
    s = c.ToString();                        // convert to string
    i = s.Length;                            // length of string
    b = s.Equals(s1);                        // true if s == s1
    b = String.Equals(s1, s2);               // true if s1 == s2
    i = String.Compare(s1, s2);              // i = -1, 0, 1 if s1 < = > s2
    i = String.Compare(s1, s2, true);        // i = -1, 0, 1 if s1 < = > s2, ignoring case
    s = s.Trim();                            // remove leading/trailing whitespace
    s = s.ToUpper();                         // equivalent uppercase string
    s = s.ToLower();                         // equivalent lowercase string
    char[] ca = s.ToCharArray();             // create character array
    s = String.Concat(s1, s2);               // s1 + s2
    s = s.Substring(i1);                     // substring starting at s[i1]
    s = s.Substring(i1, i2);                 // substring s[i1 ... i1+i2-1] (i2 is length)
    s = s.Remove(i1, i2);                    // remove i2 chars from s[i1]
    s = s.Replace(c1, c2);                   // replace all c1 by c2
    s = s.Replace(s1, s2);                   // replace all s1 by s2
    c = s[i];                                // extract i-th character of s
//    s[i] = c;                              // not allowed
    i = s.IndexOf(c);                        // position of c in s[0 ...
    i = s.IndexOf(c, i1);                    // position of c in s[i1 ...
    i = s.IndexOf(s1);                       // position of s1 in s[0 ...
    i = s.IndexOf(s1, i1);                   // position of s1 in s[i1 ...
    i = s.LastIndexOf(c);                    // last position of c in s
    i = s.LastIndexOf(c, i1);                // last position of c in s, <= i1
    i = s.LastIndexOf(s1);                   // last position of s1 in s
    i = s.LastIndexOf(s1, i1);               // last position of s1 in s, <= i1
    i = Convert.ToInt32(s);                  // convert string to integer
    i = Convert.ToInt32(s, i1);              // convert string to integer, base i1
    s = Convert.ToString(i);                 // convert integer to string

    StringBuilder                            // build strings
      sb = new StringBuilder(),              //
      sb1 = new StringBuilder("original");   //
    sb.Append(c);                            // append c to end of sb
    sb.Append(s);                            // append s to end of sb
    sb.Insert(i, c);                         // insert c in position i
    sb.Insert(i, s);                         // insert s in position i
    b = sb.Equals(sb1);                      // true if sb == sb1
    i = sb.Length;                           // length of sb
    sb.Remove(i1, i2);                       // remove i2 chars from sb[i1]
    sb.Replace(c1, c2);                      // replace all c1 by c2
    sb.Replace(s1, s2);                      // replace all s1 by s2
    s = sb.ToString();                       // convert sb to real string
    c = sb[i];                               // extract sb[i]
    sb[i] = c;                               // sb[i] = c

    char[] delim = new char[] {'a', 'b'};
    string[]  tokens;                        // tokenize strings
    tokens = s.Split(delim);                 // delimiters are a and b
    tokens = s.Split('.' ,':', '@');         // delimiters are . : and @
    tokens = s.Split(new char[] {'+', '-'});  // delimiters are + -?
    for (int i = 0; i < tokens.Length; i++)  // process successive tokens
      Process(tokens[i]);
  }
}
```

Simple list handling in Java

The following is the specification of useful members of a Java (1.5/1.6) list handling class

```
import java.util.*;

class ArrayList
// Class for constructing a list of elements of type E

  public ArrayList<E>()
  // Empty list constructor

  public void add(E element)
  // Appends element to end of list

  public void add(int index, E element)
  // Inserts element at position index

  public E get(int index)
  // Retrieves an element from position index

  public E set(int index, E element)
  // Stores an element at position index

  public void clear()
  // Clears all elements from list

  public int size()
  // Returns number of elements in list

  public boolean isEmpty()
  // Returns true if list is empty

  public boolean contains(E element)
  // Returns true if element is in the list

  public int indexOf(E element)
  // Returns position of element in the list

  public E remove(int index)
  // Removes the element at position index

} // ArrayList
```

Simple list handling in C#

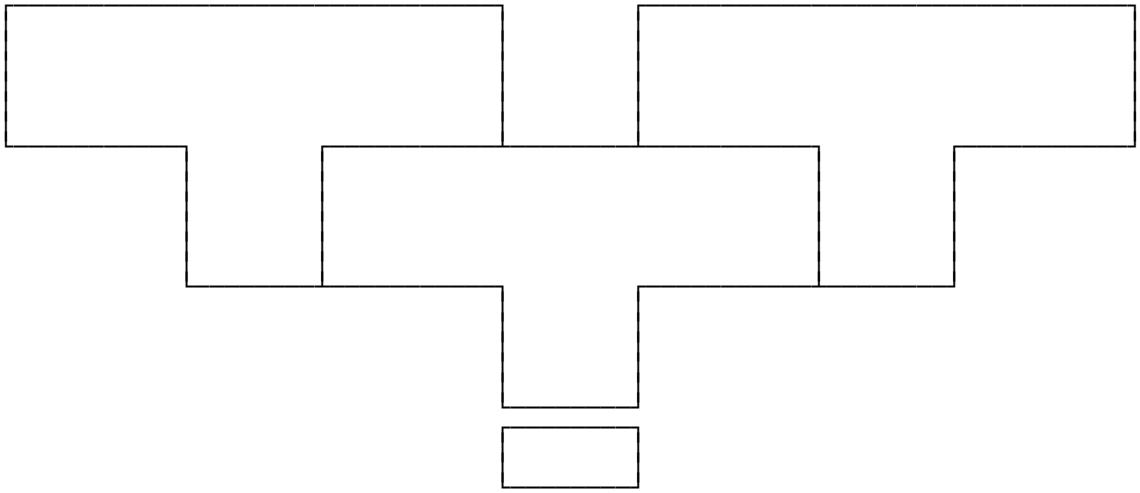
The following is the specification of useful members of a C# (2.0/3.0) list handling class.

```
using System.Collections.Generic;

class List
// Class for constructing a list of elements of type E

  public List<E> ()
  // Empty list constructor

  public int Add(E element)
  // Appends element to end of list

  public element this [int index] {set; get; }
  // Inserts or retrieves an element in position index
  // list[index] = element;  element = list[index]

  public void Clear()
  // Clears all elements from list

  public int Count { get; }
  // Returns number of elements in list

  public boolean Contains(E element)
  // Returns true if element is in the list

  public int IndexOf(E element)
  // Returns position of element in the list

  public void Remove(E element)
  // Removes element from list

  public void RemoveAt(int index)
  // Removes the element at position index

} // List
```

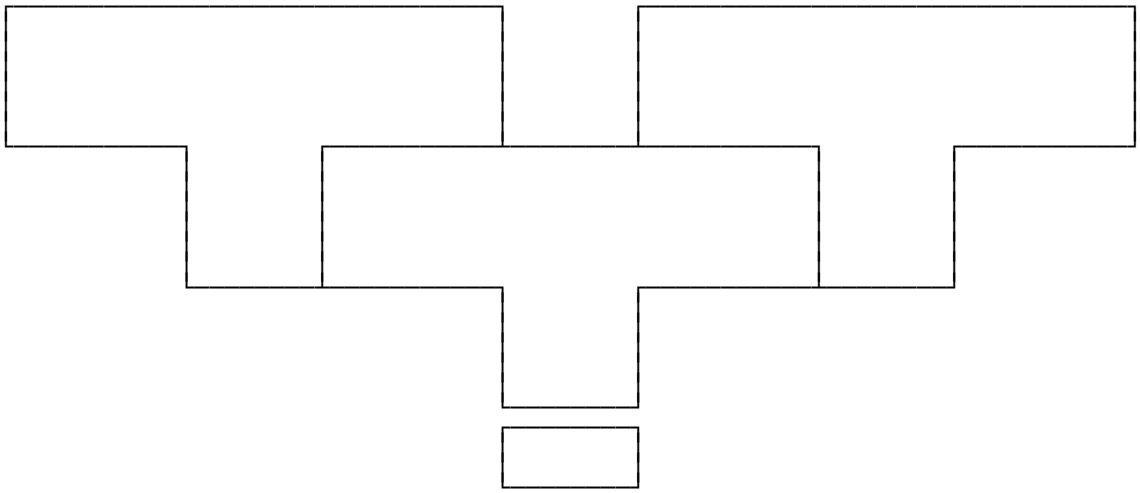
Page deliberately left blank.

A4. T Diagrams for a compiler port Give your student number

Retargetting the compiler



Rehosting the compiler



A5. Cocol grammar for describing elements of an XML subset Give your student number

```
using Library;

COMPILER XML $CN
/* Parse a set of simple XML elements */

CHARACTERS
  letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  incomment = ANY - "-" .









TOKENS
  opentag  =


  closetag =


  emptytag =


  text     =



PRAGMAS /* We cannot use the comment feature of Cocol which only allows
           two character delimiters */

  comment = "<!--" { incomment | '-' incomment } "-->" .

IGNORE  CHR(0) .. CHR(31)

PRODUCTIONS
   XML   =
```

```
END XML.
```

This question was derived, but on the advice of our local shredding panel we have decided not to incorporate it. Pity, becuase it is an excellent discriminator! It is included here merely to keep it in mind for future excercises.

QUESTION B17 **[25 marks]**

Mutually recursive functions cause problems for a system like this, which essentially requires "declare before use" almost everywhere. One way around this problem is to allow for the use of "function prototypes", as illustrated by

```
Fun(x, y);                      // notice that there is no "returns" clause here
Gun(x, y) returns Fun(x, y);    // Gun(x, y) knows that Fun(x,y) requires 2 arguments
Fun(x, y) returns Gun(x,y);     // Fun(x,y)  knows that Gun(x, y) requires 2 arguments
```

and for which the syntax is described by a very simple modification

```
FunDefinition
=  "(" ParamList ")"
    (    "returns" Definition    // complete
      |  ";"                     // prototype only
     ) .

Definition
=    Expression ";"
  |  "if" "(" Expression ")" Definition "else" Definition .
```

A full implementation of this idea would probably take longer than time allows, but discuss in general terms how you would go about implementing it, and what precautions you would have to take to get it correct.