RHODES UNIVERSITY

Supplementary (Aegrotat) Examinations - 2009/2010

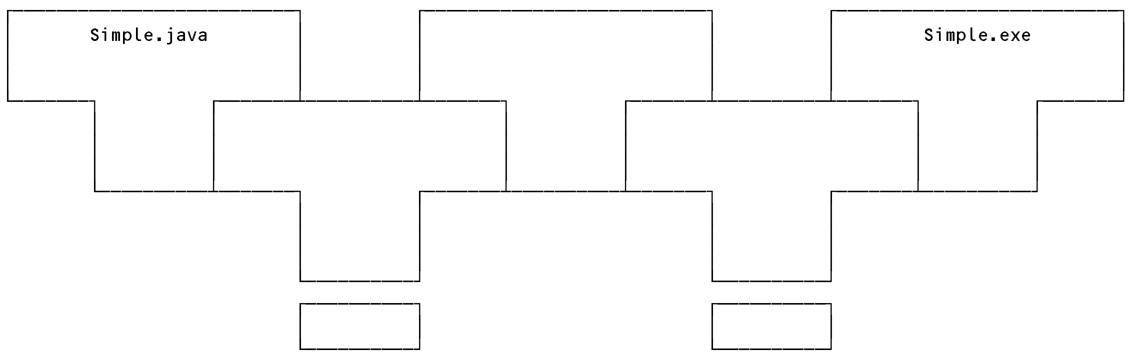
Computer Science 301 - Paper 2

Examiners:	Time 3 hours
Prof P.D. Terry	Marks 180
Prof S. Berman	Pages 14 (please check!)

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

Section A [100 marks]

1 A Java compiler might typically achieve its goals by compiling first to so-called Java byte code (an interpretable pseudo-assembly code) and then using a complementary JIT ("just in time") assembler to finish the job. Complete the T diagram representation of how such an arrangement would handle the compilation of a simple Java program. [8 marks]



2 Formally, a grammar G is 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:

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

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

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

In terms of this notation, express **precisely** (*that is to say, mathematically; we do not want a long essay or English description*) what you understand by [3 marks each]

(a) A context-sensitive grammar

(b) A context-free grammar

(c) Reduced grammar

(d) FIRST(A) where $A \in N$

(e) FOLLOW(A) where $A \in N$

(f) Nullable productions

3 Long ago, the Romans represented 1 through 10 as the strings

```
I    II    III    IV    V    VI    VII    VIII    IX    X
```

The following grammar attempts to recognize a sequence of such numbers, separated by commas and terminated by a period:

```
COMPILER Roman
  PRODUCTIONS
    Roman  = Number { "," Number }  "."  EOF .
    Number = StartI | StartV | StartX .
    StartI = "I" ( "V" | "X" | [ "I" [ "I" ] ] ) .
    StartV = "V" [ "I" ] [ "I" ] [ "I" ] .
    StartX = "X" .
END Roman.
```

(a) What do you understand by the concept of an ambiguous grammar? [2 marks]

(b) Why is this particular grammar ambiguous? [2 marks]

(c) What do you understand by the concept of equivalent grammars? [2 marks]

(d) Give an equivalent grammar to the one above, but which is unambiguous. [4 marks]

(e) Even though the grammar above is ambiguous, develop a matching hand-crafted recursive descent parser for it, similar to those that were developed in the practical course.

Assume that you have `accept` and `abort` routines like those you met in the practical course, and a scanner `getSym()` that can recognise tokens that might be described by an enumeration with names

```
EOFSym, noSym, commaSym, periodSym, iSym, vSym, xSym
```

Your parser should detect and report errors, but there is no need to incorporate "error recovery". [10 marks]

(f) If the grammar is ambiguous (and thus cannot be LL(1)) does it follow that your parser would fail to recognize a correct sequence of Roman numbers, or would report success for an invalid sequence? Justify your answer. [2 marks]

4 In the compiler studied in the course the following Cocol code was used to handle code generation for a simple *IfStatement*:

```
IfStatement<StackFrame frame>         (. Label falseLabel = new Label(!known); .)
=  "if" "(" Condition ")"             (. CodeGen.branchFalse(falseLabel); .)
   Statement<frame>                   (. falseLabel.here(); .)
.
```

Suppose it was required to extend the system to provide an optional *else* clause:

```
IfStatement = "if" "(" Condition ")" Statement [ "else" Statement ] .
```

How would the Cocol specification have to be changed to generate efficient code for this extension? [6 marks]

5 In the compiler studied in the course the following Cocol code was used to handle code generation for a *WhileStatement*:

```
WhileStatement<StackFrame frame>      (. Label startLoop = new Label(known); .)
=  "while" "(" Condition ")"          (. Label loopExit = new Label(!known);
                                         CodeGen.branchFalse(loopExit); .)
   Statement<frame>                   (. CodeGen.branch(startLoop);
                                         loopExit.here(); .)
.
```

This generates code that matches the outline

```
startLoop:    Condition
              BZE  loopExit
              Statement
              BRN  startLoop
loopExit:
```

Some authors contend that it would be preferable to generate code that matches the outline

```
whileLabel:   BRN  testLabel
loopLabel:    Statement
testLabel:    Condition
              BNZ  loopLabel
loopExit:
```

Their argument is that in most situations a loop body is executed many times, and that the efficiency of the system will be markedly improved by executing only one conditional branch instruction on each iteration.

(a) Do you think the claim is justified for an interpreted system such as we have used? Explain your reasoning. [3 marks]

(b) If the suggestion is easily implementable in terms of our code generating functions, show how this could be done. If it is not easily implementable, why not? [3 marks]

6 The following Cocol description is of a set of letters in envelopes ready to take to the post office.

```
COMPILER Mail
/* Describe simple set of mailing envelopes */

CHARACTERS
  control   = CHR(0) .. CHR(31) .
  digit     = "0123456789" .
  inaddress = ANY - control - '$:' .
  sp        = CHR(32) .
TOKENS
  number    = "postcode:" sp { sp } digit { digit } .
  info      = inaddress { inaddress } .
IGNORE control

PRODUCTIONS
  Mail      = { Envelope } EOF .
  Envelope  = Stamp { Stamp } Person Address .
  Stamp     = "$1" | "$2" | "$3" /* values of stamps implied */ .
  Address   = Street Town [ PostCode ] .
  Person    = info .
  Street    = info .
  Town      = info .
  PostCode  = number .
END Mail.
```

What would you have to add to this grammar so that the parser system could tell you (a) the total value of the stamps on all the envelopes (make the obvious assumption that the value of a stamp denoted by $x is the number x) (b) the names of all people whose addresses did **not** contain postcodes? For your (and Postman Pat's) convenience the grammar has been spread out on the last page of this question paper, which you may detach and submit with your answer book. [10 marks]

7 The following Parva program exemplifies two oversights of the sort that frequently trouble beginner programmers - the array `list` has been declared, but never referenced, while the variable `total` has been correctly declared, but has not been defined (initialised) before being referenced.

```
void main () {
  int item, total,
  int[] list = new int[10];
  read(item);
  while (item != 0) {
    if (item > 0) total = total + item;
    read(item);
  }
  write("total of positive numbers is ", total);
}
```

Discuss the extent to which these "errors" might be detected by suitable extensions of the Parva compiler/interpreter system developed in this course. You do not need to give the algorithms in detail, but you might like to structure your answer on the following lines: [6 marks]

Variables declared but never referenced:

	Not detectable at compile time because ....
or	Detectable at compile time by ....
and/or	Not detectable at run time because ....
or	Detectable at run time time by ....

Variables referenced before their values are defined:

	Not detectable at compile time because ....
or	Detectable at compile time by ....
and/or	Not detectable at run time because ....
or	Detectable at run time time by ....

8 (a) Briefly clarify what you understand by the terms "scope/visibility" and "existence" as they apply to the implementation of variables for block-structured imperative languages such as Parva, Java, C++, Pascal or Modula-2. [5 marks]

(b) Briefly describe a suitable mechanism that could be used for symbol table construction to handle scope rules and offset addressing for variables in a language like Parva. Illustrate your answer by giving a snapshot of the symbol table at each of the points indicated in the code below. (The first one has been done for you, and you can fill in the rest on the addendum page.) [6 marks]

```
void main () {
  int[] list = new int[4];
  int i, j, k;                // compilation point 1

  if (i > 0) {
    int a, b, c;              // compilation point 2

  } else {
    int c, a, d;              // compilation point 3

  }
  int[] b = new int[3], last; // compilation point 4
}
```

Point 1

Name	list	i	j	k							
Offset	0	1	2	3							

(c) If the declaration at point 3 were changed to

```
int c, a, i;          // compilation point 3
```

then the code would be acceptable to a C++ compiler but not to a Parva or Java compiler, as C++ allows programmers much greater freedom in reusing/redeclaring identifiers in inner scopes. What benefits do you suppose language designers would claim for either greater or reduced freedom? [4 marks]

9 Here is a true story. A few years ago I received an e-mail from one of the many users of Coco/R in the world. He was looking for advice on how best to write Cocol productions that would describe a situation in which one non-terminal `A` could derive four other non-terminals `W, X, Y, Z`. These could appear in any order in the sentential form, but there was a restriction that each one of the four had to appear exactly once. He had realised that he could enumerate all 24 possibilities, on the lines of

```
A = W X Y Z | W X Z Y | W Y X Z | ....
```

but observed astutely that this was tedious. Furthermore, it would become extremely tedious if one were to be faced with a more general situation in which one non-terminal could derive *N* alternatives, which could appear in any order, subject to the restriction that each should appear exactly once.

Write the appropriate parts of a Cocol specification that describes the situation and check that the restrictions are correctly met. (Restrict your answer to the case of 4 derived non-terminals, as above.) [9 marks]

Section B [90 marks]

Your examination kit includes a full kit for building the assembler system revealed to the class last November, along with some specimen Assembler programs for testing the system. both as supplied, and for the extensions which you are expected to make in this section. You may attempt these extensions in machine readable form, or you may indicate what is required either in your answer book, or on disk, or in any combination of these three methods. {note: a clerical error saw the question numbers go wrong - there is no q10)

11 The assembler does not allow the `ORG` directive to be labelled - an attempt to do so is reported rather unhelpfully merely as an "invalid statement". It is not difficult to improve on this, so do so. Hint: observe how the assembler deals with a missing label for the `EQU` directive. [6 marks]

```
         BEG            ; 300.ASM - bad labelling
         ORG  120       ; valid
LAB      ORG  130       ; invalid - no label allowed
MAX      EQU  50        ; valid
         LDI  MAX
         OTC            ; unsigned 50
         EQU  34        ; invalid - must be labelled
         HLT
         END
```

12 The virtual machine makes provision for transferring the accumulator to the index register (`TAX`), but not for transferring the index register to the accumulator. Add a `TXA` operation to the machine, and modify the assembler system to handle this operation. [6 marks]

```
         BEG            ; 301.ASM - TXA operation
         LDI  100       ; cpu.a = 100
         OTC            ; 100
         TAX            ; cpu.x = 100
         DEX            ; cpu.x = 99
         TXA            ; cpu.a = 99
         OTC            ; 99
         HLT
         END
```

13 Other operations missing from the virtual machine, but which are often found on real machines, include ones for "rotating" the contents of the accumulator "right" (`ROR`) or "left" (`ROL`).

An `ROR` operation moves all the bits one position to the right. The least significant bit (LSB) falls off into the carry bit, and is also inserted into the most significant bit (MSB) of the accumulator. For example

```
Before  0 1 1 1 0 1 1 1  (LSB is 1)  After ROR  cpu.a =  1 0 1 1 1 0 1 1  cpu.c = true
Before  1 1 1 1 0 1 1 0  (LSB is 0)  After ROR  cpu.a =  0 1 1 1 1 0 1 1  cpu.c = false
```

An `ROL` operation moves all the bits one position to the left. The most significant (MSB) bit falls off into the carry bit, and is also inserted into the least significant bit (LSB) of the accumulator. For example

```
Before  0 1 1 1 0 1 1 1  (MSB is 0)  After ROL  cpu.a =  1 1 1 0 1 1 1 0  cpu.c = false
Before  1 1 1 1 0 1 1 0  (MSB is 1)  After ROL  cpu.a =  1 1 1 0 1 1 0 1  cpu.c = true
```

Add these operations to the virtual machine and modify the assembler system to handle them. Both operations must unset the overflow bit `cpu.v`, but might affect the `cpu.z` and `cpu.n` flags. [20 marks]

```
        BEG             ; 301.ASM - rotation right - see also 302.asm
        LDI  01110110%  ; 01110110
        OTB
        ROR
        OTB             ; 00111011
        ROR
        OTB             ; 10011101
        HLT
        END
```

14 It is often useful to produce a cross reference listing of the labels introduced in an Assembler program. For the program

```
        BEG             ; 305.ASM - cross references
        LDI  100        ; cpu.a = 100
        BRN  STOP
MAX     EQU  150
MIN     EQU  10
HERE    HLT
        NOP
STOP    LDI  MAX        ; 150
        ADI  MAX        ; 300 % 256 = 44
        STA  ANSWER
        OTC             ; 44
        HLT
ANSWER  DS   1
        END
```

this might take the form below. "Defining" references are marked by the negative of the line number on which the label was defined; "applied" references are marked by the line number on which the label was referenced:

```
Cross reference list of labels in this program

   STOP          3  -8
   MAX          -4   8   9
   MIN          -5
   HERE         -6
   ANSWER       10 -13
```

Extend the assembler, and in particular the table handler, to provide this facility. It will suffice to display the cross reference table on the standard output (use the familiar IO library). [22 marks]

15 A cross reference table may not always be required (and if not, tends to become a nuisance). Modify the assembler system (in particular the `Assem.frame` file) to require the user to give a command line directive if the table is required. [6 marks]

```
Assem 305.asm -x       (cross reference table required)
Assem 305.asm          (cross reference table suppressed)
```

16 Although not incorrect, labels that are defined but never referenced may indicate a possible error on the part of the programmer. Modify the assembler so that if a program like the one below is assembled, the user will receive a warning that `MIN` and `HERE` were never referenced. [6 marks]

```
        BEG             ; 304.ASM - unused labels
        LDI  100        ; cpu.a = 100
        BRN  STOP
MAX     EQU  120
MIN     EQU  10
HERE    HLT
        NOP
STOP    LDI  MAX
        OTC             ; 120
        HLT
        END
```

17 Sophisticated assemblers often provide what is called "conditional assembly". A section of source code may be enclosed in a directive pair `IF ... ENDIF`, but this code is only assembled and included in the generated code if the argument to the `IF` directive is non-zero. An example will clarify this further

```
        BEG                ; conditional assembly
DEBUG   EQU  1             ; choose appropriately

        LDI  50            ; this will be assembled unconditionally
        OTC                ; so will this

        IF   DEBUG         ; since DEBUG is non-zero the next 3 statements will be assembled
                           ; but had we defined DEBUG  EQU  0 they would be omitted
        ADI  50            ; 100
        OTC                ; 100
        OTI                ; 100
        ENDIF              ;
                           ; code from here on will be assembled unconditionally
        SHR                ; 50
        OTC                ; 50
        HLT

        END
```

The system can be enhanced further by allowing an `ELSIF` directive to appear between `IF` and `ENDIF`:

```
        BEG                ; 306.ASM - conditional assembly
DEBUG   EQU  1             ; choose appropriately

        LDI  50
        OTC                ;

        IF   DEBUG         ; in this example the next lines will be included
        ADI  50            ; 100
        OTC                ; 100
        OTI

        ELSE               ; in this example the next line will not included
        OTC                ;
        ENDIF              ;

        SHR                ; 50
        OTC                ; 50
        HLT

        END
```

This may look complicated but is, in fact, easily implemented; show how to do this. [24 marks]

Hint: bear in mind the following:

(a) `IF`, `ELSE` and `ENDIF` are directives, not executable opcodes, and these directives (like `ORG`) may not be labelled;

(b) the argument to `IF` (like those to `EQU` and `ORG`) must be one that can be completely evaluated at the point where it is encounterd;

(b) conditional assembly is a compile-time feature - it is very different from the familiar *if-then-else* construction with a condition that is only evaluated at run-time;

(c) code that is *not* assembled may not define labels that might otherwise be referred to;

(d) for the purpose of this question you are not required to allow the directives to be "nested".

Free information

Summary of useful library classes

The following summarizes some of the most useful aspects of the available simple I/O classes.

```
public class OutFile {  // text file output
  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
  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
```

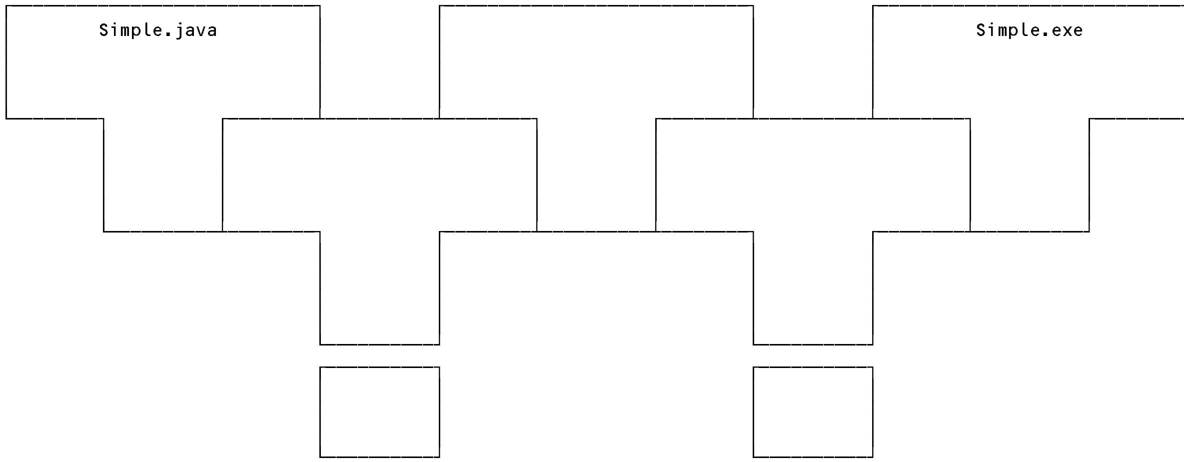
Simple list handling in C#

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
```

Question 1 - Java with JIT (Detach this page and complete your solution on it)



Question 8 - Scope rules (Detach this page and complete your solution on it)

```
void main () {
  int[] list = new int[4];
  int i, j, k;                // compilation point 1

  if (i > 0) {
    int a, b, c;              // compilation point 2

  } else {
    int c, a, d;              // compilation point 3

  }
  int[] b = new int[3];
  bool last;                  // compilation point 4
}
```

Point 1

1

Name	list	i	j	k											
Offset	0	1	2	3											

2

Name															
Offset															

3

Name															
Offset															

4

Name															
Offset															

Question 6 - Postman Pat (Detach this page and complete your solution on it)

```
COMPILER Mail
/* Describe simple set of mailing envelopes */




CHARACTERS
  control   = CHR(0) .. CHR(31) .
  digit     = "0123456789" .
  inaddress = ANY - control - '$:' .
  sp        = CHR(32) .
TOKENS
  number    = "postcode:" sp { sp } digit { digit } .
  info      = inaddress { inaddress } .
IGNORE control

PRODUCTIONS
  Mail
  = { Envelope
  } EOF
  .

  Envelope
  =   Stamp
    { Stamp
    } Person
    Address
  .

  Stamp
  =   "$1"
    | "$2"
    | "$3"
  .

  Address
  = Street
    Town
    [ PostCode
    ]
  .

  Person
  = info
  .

  Street
  = info
  .

  Town
  = info
  .

  PostCode
  = number
  .

END Mail.
```