RHODES UNIVERSITY

November Examinations - 2009

Computer Science 301 - Paper 2

Examiners:	Time 3 hours
Prof P.D. Terry	Marks 180
Prof S. Berman	Pages 13 (please check!)

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

(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 develop an assembler/interpreter system for the Assembler language studied in their second year. 16 hours before the examination a complete grammar and other support files for building such a 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 system, access to a computer, and machine readable copies of the questions.)

Section A: Short questions [100 marks]

A1 *Pragmas* and *comments* often appear in source code submitted to compilers. What property do pragmas and comments have in common, and what is the semantic difference between them? [4 marks]

Solution:

Neither pragmas not comments affect the dynamic semantics of a program, and can be ignored in that respect. They usually can also appear in source anywhere that white space can be, that is, between any two tokens. But pragmas can choose compiler options, as students should have realized from using them in the practical course.

A2 What do you understand by the term *short circuit semantics* as applied to the evaluation of Boolean expressions? Illustrate your answer by means of two specimens of code that might correspond to a simple Boolean expression of your choice. [6 marks]

Solution:

Standard bookwork sort of discussion expected, and students should know about this from first year in any case. An example might be

```
WHILE  (1 < P)  AND  (P < 9)  DO   P := P + Q  END

L0      if 1 < P goto L1   ; Short circuit version
        goto L3
L1      if P < 9 goto L2
        goto L3
L2      P := P + Q
        goto L0
L3      continue


L0      T1 := 1 < P         ; standard Boolean operator approach
        T2 := P < 9
        if T1 and T2 goto L1
        goto L2
L1      P := P + Q
        goto L0
L2      continue
```

Alternatively it is more or less adequate to point out that in short-circuit semantics

```
A and B   is equivalent to    if not A then false else B
A or  B   is equivalent to    if A then true else B
```

Essentially the important point is that it may not be necessary to evaluate both operands. This is useful in writing protective code like

```
if (p != null  &&  p.flag) ...
if (d != 0  &&  n / d > 12)  ...
```

A3 Parva programs are really very similar to Java ones, except for a few simple differences, as the following example will illustrate:

Parva: (Demo.pav)

```
void main () {                              // A Parva program has a single void method
  int year, yourAge;
  const myAge = 64;                         // Parva constants are defined like this
  read("How old are you? ", yourAge);       // Parva has a multiple argument read statement
  if ((yourAge < 0) || (yourAge > 100)) {
    write("I do not believe you!");
    halt;                                   // Parva has a halt statement
  }
  bool olderThanYou = myAge > yourAge;      // Parva uses the keyword bool
  write("A claim that you are older than me is ", !olderThanYou);
}
```

Java equivalent: (Demo.java)

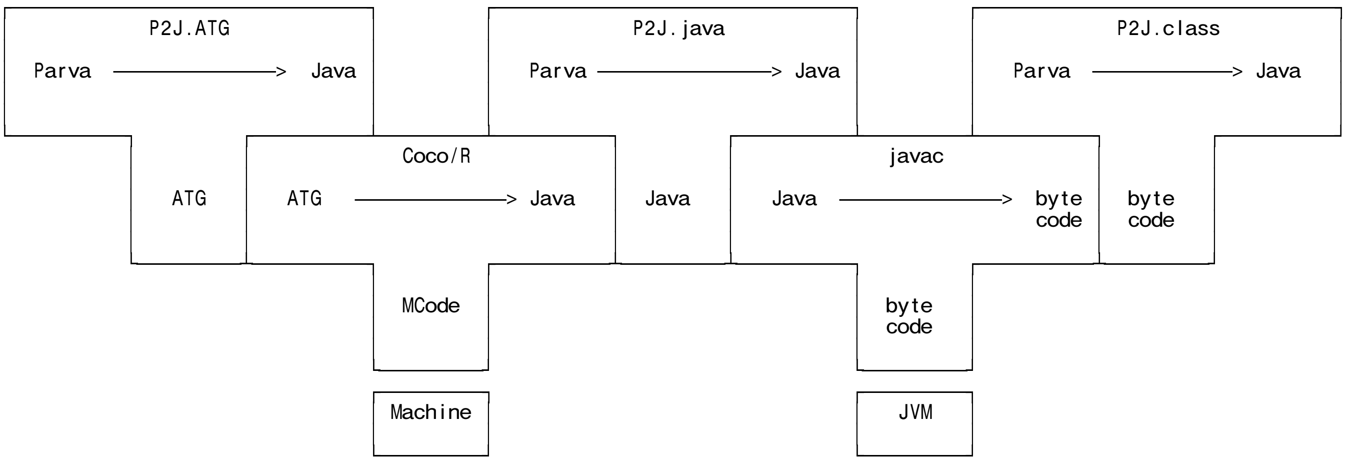
```
import library.*;                           // A simple Java program has a single class
                                            // and usually has to import library classes
class Demo {

  public static void main(String[] args) {  // A simple Java program has a standard main method
    int year, yourAge;
    final int myAge = 64;                   // Java constants are defined like this
    IO.write("How old are you? ");          // Java will use methods from a library for I/O
    yourAge = IO.readInt();
    if ((yourAge < 0) || (yourAge > 100)) {
      IO.write("I do not believe you!");
      System.exit(0);                       // Java has the equivalent of a halt statement
    }
    boolean olderThanYou = myAge > yourAge; // Java uses the keyword boolean
    IO.write("A claim that you are older than me is ");
    IO.write(!olderThanYou);
  }
}
```

This should suggest that it must be relatively easy to develop a translator that will convert a Parva program to a Java equivalent, using a tool like Coco/R, acting on a grammar for Parva and with semantic actions that for the most part simply copy tokens from input to output, but making a few substitutions (such as replacing `bool` by `boolean`) and additions (such as adding a standard header with an `import` directive and `class` declaration). Indeed, such a system was developed by your predecessors in this class some years ago under examination conditions.

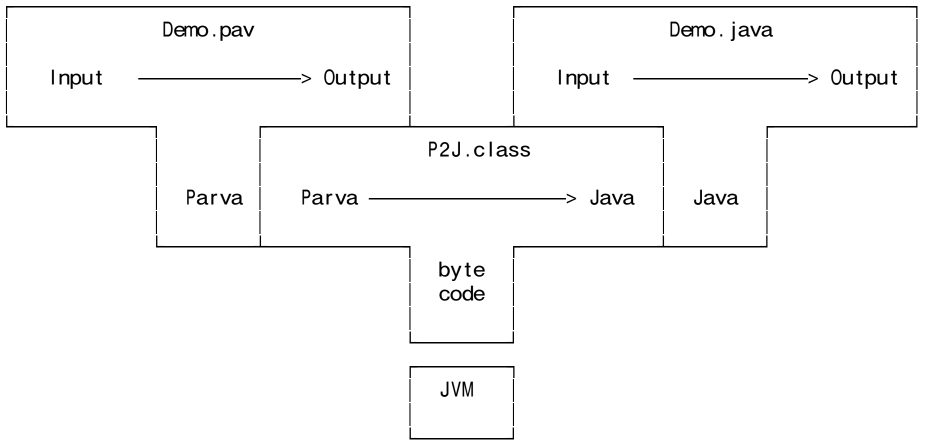
(a) Show, by means of appropriate T diagrams the stages involved in producing an executable version of such a translator (P2J) assuming that you have an executable version of Coco/R for either C# or Java, as well as an executable version of either a C# or Java compiler. [5 marks]

A set of blank T-diagrams is provided in the free information, which you can complete and submit with your answer book.



(b) Also give a T diagram showing the process by which a program like `Demo.pav` would be translated to one like `Demo.java`. [2 marks]

Solution



(c) You are (obviously) not expected to give full details of the system. However, if you can make the assumption that the Parva program to be converted is syntactically and semantically correct, do you foresee that your system would have to incorporate a symbol table handler? Justify your answer; don't just guess! [6 marks]

Solution:

A symbol table **is** needed. The example gave two clues. Firstly, one cannot translate the general "read" statement without knowing and accommodating the type of the designators it mentions (that is to be able to generate a call to `IO.readBool()`, `IO.readInt()` and so on. Secondly, the form of the `const` declaration in Parva has implicit typing, and this has to be made explicit in Java, which suggests that a symbol table might be needed. In fact it is possible to transform the `const` declaration without a symbol table, but to see that requires more analysis than students could have been expected to perform in the time available.

A4 Consider the following Cocol specification:

```
COMPILER Expressions

CHARACTERS
  digit  = "0123456789" .
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .

TOKENS
  ident  = letter { letter | digit } .

IGNORE CHR(0) .. CHR(31)

PRODUCTIONS
  Expressions  = { Term "?" } EOF .
  Term         = Factor { "+" Factor | "-" Factor } .
  Factor       = ident [ "++" | "--" ] | [ "abs" ] "(" Term ")" .
END Declarations.
```

Develop a handcrafted scanner (not a parser!) that will recognize the tokens used in this grammar. Assume that the first few lines of the scanner routine are introduced as

```
static int getSym() {
  while (ch <= '  ') getChar();
```

and that the `getSym()` method will return an integer representing the token that it has recognized, as one of the values that you can denote by the names:

```
plus, minus, plusPlus, minusMinus, query, lParen, rParen, abs, ident, error
```

Assume that when the scanner invokes the source handling method

```
static void getChar()
```

this will assign to a static field `ch` (of the class of which these methods are members) the next available character in the source text. [16 marks]

Solution:

A method might go something like the one below (students had constructed scanners like this in practicals, though not this particular one). Unfortunately the exam paper had a typo - the `error` value had not been enumerated. Several students attempted to compensate for this which was praiseworthy. Also not mention was made of the possibility that an EOF might be detected (this is much the same as the `error` case of course. Typical errors made in submissions were the omission of `getChar()` calls -and few submissions noted that this was not supposed to be a `void` method.

```
static int GetSym() {
  while (ch <= ' ') getChar();;
  if (Character.isLetter(ch)) { // identifier or keyword
    StringBuilder sb = new StringBuilder();
    while (Character.isLetterOrDigit(ch) {
      sb.append(ch);
      getChar();
    }
    if (sb.toString().equals("abs")) return abs; else return ident;
  }
  switch (ch) {
    case '?': getChar(); return query;
    case '(': getChar(); return lparen;
    case ')': getChar(); return rparen;
    case '+': getChar();
              if (ch != '+') return plus;
              getChar(); return plusPlus;
    case '-': getChar();
              if (ch != '-') return minus;
              getChar(); return minusMinus;
    default : getChar(); return error;
  }
}
```

A5 Consider the following grammar, expressed in EBNF, that describes the form of a typical university course:

```
Course       = Introduction Section { Section } Conclusion .
Introduction = "lecture" [ "handout" ] .
Section      = { "lecture" | "prac" "test" | "tutorial" | "handout" } "test" .
Conclusion   = [ "panic" ] "Examination" .
```

(a) What do you understand by the statement "two grammars are equivalent"? [2 marks]

Two grammars are equivalent if they can generate (describe) exactly the same set of sentences (not necessarily yielding the same parse trees or using the same sets of productions).

(b) What do you understand by the statement "a grammar is ambiguous"? [2 marks]

A grammar is ambiguous if there is at least one sentence that can be derived from the goal symbol in more than one way. This is a very simple definition, and it is alarming that students cannot explain ambiguity succinctly.

(c) Rewrite these productions so as to produce an equivalent grammar in which no use is made of the EBNF meta-brackets { ... } or [...]. [8 marks]

```
Course              = Introduction Section Sections Conclusion .
Sections            = Section Sections | ε .
Introduction        = "lecture" OptionalHandout .
OptionalHandOut     = "handout" | ε .
Section             = Components "test" .
Components          = ( "lecture" | "prac" "test" | "tutorial" | "handout" ) Components | ε .
Conclusion          = UnderstandablePanic "Examination" .
UnderstandablePanic = "panic" | ε .
```

(d) Clearly explain why the derived set of productions does not obey the LL(1) constraints? [4 marks]

OptionalHandout is nullable, and *FIRST(OptionalHandOut) = { "handout" }* while *FOLLOW(OptionalHandout)* is the set *FIRST(Section) = {"lecture", "prac", "tutorial", "handout"}*, and these two sets have *"handout"* in common.

(e) The original grammar happens to be ambiguous. Give an example of a sentence that demonstrates this. [2 marks]

We only need to find one sentence that can be parsed in two ways. A fairly obvious example is

```
lecture handout test examination
```

can be parsed with handout regarded as either a component of the Introduction or as a component of the first Section after an Introduction that has no handout.

(f) Why does it not follow that every grammar that is not LL(1) must be ambiguous? Justify your argument by giving an example of a simple non-LL(1) grammar that is not ambiguous. [5 marks]

No, it does not follow - it is unidirectional. Ambiguous implies non-LL(1) but non-LL(1) does not necessarily imply ambiguous! The obvious example to cite here is a left recursive grammar for expressions: Left recursive grammars are always non-LL(1), but do not have to be ambiguous! As simple example would be

```
Expression = Expression ( "+" | "-" Term ) | Term .
Term       = "a' | "b" .
```

(g) Does it follow that if a parser is constructed for this grammar that it is doomed to fail? Justify your answer. [3 marks]

No it does not. This is the famous "dangling else" in disguise. It would of course matter if there was a different semantic "value" for a `handout` that was issued as part of an *Introduction* or as part of a *Section* (which of course there might be - remember that the grammar as it stands just handles syntax). The sentence given earlier would be parsed by a recursive descent parser quite happily, binding the `handout` to the introductory `lecture`.

(h) The University Senate have decreed that courses may no longer be offered if they do not obey various constraints. In particular, an acceptable course may not

(1) Offer fewer than 50 lectures, or subject the candidates to more than 8 tests
(2) Hold a practical until at least 4 lectures have been given
(3) Hold more practicals than tutorials

Show how the grammar may be augmented to impose these constraints. A spaced copy of the grammar appears in the free information, which you are invited to complete and hand in with your answer book. [12 marks]

A simple solution follows. The system is simple enough that the totals needed can be held in static fields of the parser - any attempt to use parameters is rather overkill. The question was capable of various interpretation as it happened, and it would have been acceptable to report on an excessive number of tests at the end, rather than within a section.

These are rather trite questions - one does not really need a system like Coco/R to develop simple counting applications (although, as we can see, it makes it very easy to do so). This sort of question is useful for probing whether students can appreciate where attributes must be added to the grammar; beginners tend to insert them almost at random, but their placement is usually critical.

```
COMPILER Course

  static int lectures = 0, tutorials = 0, tests = 0, practicals = 0;

IGNORE CHR(0) .. CHR(31)
```

```
PRODUCTIONS
  Course
  = Introduction
    Section
    { Section
    } Conclusion              (. if (lectures < 50 || practicals > tutorials)
                                   SemError("Constraints on course not met") .) .

  Introduction
  = "lecture"                (. lectures++ .)
    [ "handout" ] .


  Section
  = {   "lecture"            (. lectures++; .)
      | "prac"               (. practicals++;
                                if (lectures < 4)
                                  SemError("Not enough lectures before this prac"); .)
        "test"               (. tests++;
                                if (tests > 8)
                                  SemError("Too many tests");
      | "tutorial"           (. tutorials++; .)
      | "handout"
    }
    "test"                   (. tests++;
                                if (tests > 8)
                                  SemError("Too many tests"); .) .

  Conclusion
  = [ "panic" ]
    "Examination" .

END .
```

A6 Generations of Parva programmers have complained about the absence of a *for* loop, and it has been decided to add this feature, using syntax suggested by Pascal, and exemplified by

```
for i = 1 to 10 write(i);
for j = i + 1 to i - 4 {
  read(i); total = total + i;
}
```

(a) Write an EBNF description that should allow you to recognize such a statement, taking care not to be over restrictive. [2 marks]

Solution:

```
ForStatement = Designator "=" Expression "to" Expression Statement .
```

The unwarranted restriction might arise if one limited *Designator* to `identifier` (say) or used `number` instead of *Expression*. Actually, using `identifier` would be usful if one wished to deal with (e) below more restrictively. There is no need to use *Block* in place of *Statement* - a *Block* is an acceptable form of *Statement*!

Several submissions suggested

```
ForStatement = Assignment "to" Expression Statement .
```

but this is not a good idea. In Parva an *Assignment* is terminated by a semicolon, just for a start.

(b) What static semantic constraints must be satisfied by the various components of your production(s)? You might illustrate this by writing the Cocol attributes; alternatively just specify them in concise English. [4 marks]

Solution:

The two *Expressions* must yield values of a type that is at least assignment compatible with the type of the *Designator*, and these must be ordinal types (that is, map to simple integers). So in the context of the compiler students had worked with, one could have used an integer control variable with expressions of integer or character type, or a character control variable with expressions of character type, but not control variables and expressions of Boolean type or a control variable of character type and expressions of integer type. It would be too much to hope that candidates would see all those points, especially those who have been corrupted by coding in a near typeless language like C.

(c) Critically examine and comment in detail on the suggestion that a *for* loop of the form

```
for i = start to stop {
  // something
}
```

should be regarded as (dynamically) semantically exactly equivalent to the following: [10 marks]

```
i = start;
while (i <= stop) {
  // something
  i++;
}
```

Solution:

Aha - a language lawyer question! The breed known as the language lawyer was mentioned in class discussions - the sort of person that writes down silly example code and challenges the reader to say what it really means. Simple as they seem to be, *for* loops abound with possibilities for constructing funny examples that could get one into trouble.

There are two aspects of the *for* loop that should be emphasized that suggest it could be rather different from a simple *while* loop as suggested above. Firstly, in the *for* loop the system usually arranges to evaluate the limits imposed by the two *Expressions* once only, before the loop can begin. Secondly, it may be desirable to regard the *for* loop as creating another level of scope, with the scope if the control variable confined to the statement as a whole. To my delight (and, I must say, astonishment) several submissions raised the scope issue (which I do not recall mentioning in class for a *for* statement; in fact I don't really remember disussing the *for* statement at all!)

To show up these problems, consider

```
for i = i to i + 4  {  // in principle never terminates if you interpret it as above
  // something
}

for i = i to maxInt {  // in practice never terminates (overflows and becomes negative)
  // something
}

for i = i to maxInt {  // in practice might never terminate - something messes with i
  // something
  i = 0;
}
```

Many of these issues can be overcome by a fairly dramatic process. We arrange that the code generation for the generic loops (the *downto* loop is a pretty obvious variation and extension)

```
for Variable = Expression1 to Expression2 Statement
for Variable = Expression1 downto Expression2 Statement
```

must yield lower level code of the form

```
         Temp1 := Expression1                          Temp1 := Expression1
         Temp2 := Expression2                          Temp2 := Expression2
         IF Temp1 > Temp2 THEN GOTO EXIT               IF Temp1 < Temp2 THEN GOTO EXIT
         Variable := Temp1;                            Variable := Temp1;
BODY:    Statement                            BODY:    Statement
         IF Variable = Temp2 THEN GOTO EXIT            IF Variable = Temp2 THEN GOTO EXIT
         Variable := Variable + 1                      Variable := Variable - 1
         GOTO BODY                                     GOTO BODY
EXIT:                                         EXIT:
```

respectively, where `Temp1` and `Temp2` are temporary variables. This code will not assign a value to the control variable at all if the loop body is not executed, and will leave the control variable with the "obvious" final value if the loop body is executed. It may appear to be a little awkward for an incremental compiler, since there are now multiple apparent references to extra variables and to the control variable. However, it can be done for Parva and the PVM if one resorts to the usual trick of defining a few more opcodes.

(d) Why do you suppose that language designers take the easy way out and state that "the value of the control variable of a *for* loop is to be regarded as undefined after the loop terminates?" [2 marks]

Solution:

To give implementors the possibility of writing loops that really behave nicely, imposing the scope restrictions mentioned earlier, and so on.

```
for i = 2 to 10 {  // you would expect 10, but probably get 11
  // something
}

for i = 10 to 2 {  // in practice never starts, but why should i not be left at 2
  // something
}

i = 4;
for i = i to i + 5 {  // should this stop at 9, or 10, or go on indefinitely:
  // something
}
```

Best just to duck all these!

(e) If a *for* loop is implemented as in (c) - or, indeed, in any other way - problems would arise if the code indicated by "something" were to alter the value of the control variable. Do you suppose a compiler could forbid this practice? If so, how - and if not, why not? [5 marks]

Solution: As we discovered when trying to standardize Modula, it is almost impossible to be completely watertight. However, if the symbol table entry for the control variable is marked in some way as "cannot be changed for the duration of the loop body" and if the productions for input statements, assignment statements and statements like `i++;` check that their "target" is not marked in this way, one can certainly handle the situation in simple Parva programs like those that students compiled, which had only a single main method. It gets more difficult if one can pass control variables as reference parameters, or if global variables (static fields) are permitted as control variables, and even more awkward if one can find other aliased pointers to control variables and attack them in that way! But one hopes that students will at least have thought of the idea of marking a symbol table entry as "don't touch" for the duration of parsing the loop body.

Section B: Constructing an assembler [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.

Yesterday you made history when you were invited to develop a basic assembler/interpreter for the Assembler language that was used in our second year course for many years.

Later in the day you were provided with a sample solution to that challenge, and the files needed to build that system have been provided to you again today.

Since the basic system was built under great pressure, some subtle points were conveniently overlooked, and some features were omitted. So continue now to answer the following unseen questions that will test your ability to refine the system further. As a hint, many of these refinements only require you to add or modify a few lines of code as it is currently written. Your answers are best given by showing the actual code you need, not by giving a few sentences of explanation!

B7 The assembler currently will not allow you to have blank lines or comments before `BEG` or after `END`, as illustrated by

```
                                 ; This is an introductory comment
                                 ; extending over a few lines before
                BEG              ; the program code actually starts

                END              ; not quite the final comment
                                 ; because there is still more to say
```

Show how to modify the system to allow this. [4 marks]

```
         Begin
   ***   = { CommentEOL }
           "BEG" CommentEOL .

         End
   ***   = "END" { CommentEOL } .
```

B8 The assembler currently will allow you to define a label more than once, as in

```
                BEG
          A     LDI  23
                OTI
          A     HLT              ; duplicated label
                END
```

Show how to modify the system to report duplication of labels as incorrect. [4 marks]

```
         Statement                          (. AdrRec constant;
                                               Entry tableEntry;
                                               string name; .)
         =
             (   Unlabelled
               | Label                      (. name = token.val; tableEntry = Table.Find(name);
   ***                                         if (tableEntry != null && tableEntry.isDefined)
   ***                                           SemError("redefined label"); .)
         ...
```

B9 The assembler currently performs no checks to ensure that all labels have been correctly defined, as in

```
                BEG
                LDA  A           ; A is never defined
                OTI
                HLT
                END
```

Show in some detail how to modify the system to report on any labels that are never defined. [10 marks]

```
         Assem
   ***   = Begin StatementSequence End      (. Table.CheckLabels(); .) .
```

with the `Table` class extended to include

```
   ***     public static void CheckLabels() {
   ***     // Checks that all labels have been defined (no forward references outstanding)
   ***       foreach (Entry e in list)
   ***         if (!e.isDefined)
   ***           Parser.SemError("undefined label - " + e.name);
   ***     } // CheckLabels
```

An even better way is as follows, since the system just given will list only one undefined label, as Coco/R suppresses multiple error messages at one point:

```
***     public static void CheckLabels() {
***     // Checks that all labels have been defined (no forward references outstanding)
***       string undefined = "";
***       bool badLabels = false;
***       foreach (Entry e in list)
***         if (!e.isDefined) {
***           badLabels = true;
***           undefined = undefined + " " + e.name;
***         }
***       if (badLabels) Parser.SemError("undefined labels - " + undefined);
***     } // CheckLabels
```

B10 Some assemblers allow you to use the mnemonics of opcodes as operands in addresses, as in

```
                BEG
                LDI  OTA         ; Equivalent to  LDI  27 (see table of opcodes)
                ADI  PSH + POP   ; Equivalent to  ADI  16 + 17
                HLT
                END
```

Show how to modify the system to allow this. [6 marks]

Solution (the example was unfortunate - it illustrated only "one-word" operations, but opcodes for "two-word" operations would also be allowed):

```
        Term<out AdrRec term, int sign>      (. term = new AdrRec(); .)
        =  (   Label                         (. Entry tableEntry = Table.Find(token.val);
                                                if (tableEntry == null) {
                                                  Table.Insert(new Entry(token.val, new Ref(sign, CodeGen.Curren
                                                }
                                                else if (!tableEntry.isDefined)
                                                  tableEntry.AddRef(new Ref(sign, CodeGen.CurrentAddress()));
                                                else {
                                                  term.value   = tableEntry.value;
                                                  term.isKnown = true;
                                                } .)
             | IntConst<out term.value>      (. term.isKnown = true; .)
             | CharConst<out term.value>     (. term.isKnown = true; .)
***          | ( OneByteOp | TwoByteOp )     (. term.isKnown = true; term.value = VM.OpCode(token.val);.)
             | '*'                           (. term.isKnown = true; term.value = CodeGen.CurrentAddress(); .)
           ) .
```

B11 The virtual machine for this system should ensure that the values of all registers are confined to the range 0 … 255. Currently this is not the case (although for simple programs you may not have noticed!) There are several places that need attention, including the code that handles input operations and routines for incrementing and decrementing registers (there may be others as well).

Indicate the changes needed to the virtual machine emulator to handle this correctly. [8 marks]

Solution: This is much simpler than it looks - provided that you realise that the one-bye model and the memory addressin should all simply wrap around). All we have to do is modify the helper methods to achieve this (the VM supplied had carefully been crafted to use the `Pred()` and `Succ()` methods, rather than yielding to the temptation to write something`++ all over the place!)`

So we require several opcodes to manipulate values modulo 256:

```
        static int Succ(int x) {
        // Increment x
***       return (x + 1) % 256;
        } // Succ

        static int Pred(int x) {
        // Decrement x
***       return (x + 255) % 256;
        } // Pred

        static int Index() {
        // Compute indexed address and bump cpu.pc
***       int x = (mem[cpu.pc] + cpu.x) % 256;
          cpu.pc = Succ(cpu.pc);
          return x;
        } // Index
```

Handling the input operations has to be done separately, and can be done like this

```
                 case VM.ini:
                   if (tracing) results.Write(' ', 75);
***                cpu.a = (data.ReadInt() % 256 + 256) % 256;
                   cpu.v = false; SetFlags(cpu.a);
                   if (data.NoMoreData()) ps = noData;
                   else if (data.Error()) ps = badData;
                   else if (tracing) data.ReadLine();
                   break;
```

with similar modifications made to the code for `VM.inb` and `VM.inh`. Note that code like

```
                 cpu.a = data.ReadInt() % 256;
```

would only work if the input data happened to be positive.

B12 Similarly, there may be places in the assembler, code generator and table handler that are not paying attention to the same requirements as in question B11. Indicate the changes needed to improve the situation. [10 marks]

Solution:

For safety one might modify the `IntConst` production in a similar way to the modifications just suggested for the VM input operations:

```
              IntConst<out int value>            (. value = 0; .)
             = (   decNumber ...
                 | binNumber ...
                 | hexNumber ...
***            )                                 (. value = (value % 256 + 256) % 256; .) .
```

but this is unnecessary. It is the `CodeGen` class that has to be careful. Here is one suggestion. Introduce a simple helper method that will normalise values to produce their equvalent in the range 0 .. 255, and then call this from other methods in the code generator:

```
             class CodeGen {
               static int codeLen = 0;

***            private static int ToByte(int value) {
***            // Returns value reduced to 0 .. 255
***              return (value % 256 + 256) % 256;
***            } // ToByte

               private static void Emit(int value) {
               // Code generator for single value
***              VM.mem[codeLen] = ToByte(value); codeLen = (codeLen + 1) % 256;
               } // Emit

               public static void ReserveBytes(int length) {
               // Reserve length bytes of memory
***              if (length > 0) codeLen = ToByte(codeLen + length);
                 else Parser.SemError("Invalid value for DS");
               } // ReserveBytes

               public static void SetOrg(int start) {
               // Redefine location counter
***              if (start >= 0) codeLen = ToByte(start);
                 else Parser.SemError("Invalid value for ORG");
               } // ReserveBytes

               public static void PatchRef(Ref oneRef, int value) {
               // Patch oneRef to accommodate known value
***              VM.mem[oneRef.address] = ToByte(VM.mem[oneRef.address] + oneRef.sign * value);
               } // AmendByte

             } // end CodeGen
```

B13 In a single pass assembler like this, the `DS` directive becomes very awkward unless the value of its address field is known at the point that assembly has reached. The following code is meaningless:

```
           BEG
           LDA  LIST       ; Equivalent to LDA 3
           HLT
     LIST  DS   A          ; ??
     A     DS   LIST - 12  ; ??
           END
```

Show how to modify the system to report this sort of nonsense as incorrect. [4 marks]

Solution - see below:

B14 Add an ORG directive to the assembly language that will allow you to specify the address from which future assembly will follow, as in

```
           BEG
           LDA  LIST       ; Equivalent to LDI 100
           BNZ  ADD5       ; Equivalent to BNZ 110
           HLT
           ORG  100        ; shift assembly
     LIST  DS   10         ; LIST occupies bytes 100 ... 109
     ADD5  ADI  5          ; ADI is in byte 110
           HLT
           END
```

Show how to modify the system to allow this. [8 marks]

Solution:

The solutions for B13 and B14 are as follows. Note that ORG may not be labelled:

```
          Statement                       (. AdrRec constant;
                                             Entry tableEntry;
                                             string name; .)
          =
              (   Unlabelled
                | Label<out name>          (. tableEntry = Table.Find(name);
                                              if (tableEntry != null && tableEntry.isDefined)
                                                SemError("redefined label"); .)
                                              if (tableEntry == null)
                                                Table.Insert(new Entry(name, CodeGen.CurrentAddress()));
                                             else if (!tableEntry.isDefined)
                                                tableEntry.CompleteReferences(CodeGen.CurrentAddress()); .)
                  Unlabelled
              )
***         | "ORG" Address<out constant>  (. if (constant.isKnown)
***                                              CodeGen.SetOrg(constant.value);
***                                            else
***                                              SemError("ORG must have a constant address field"); .) .

          Unlabelled                       (. AdrRec address;
                                              string str; .)
          = [   OneByteOp                  (. CodeGen.OpCode(token.val); .)
              | TwoByteOp                  (. CodeGen.OpCode(token.val); .)
                Address<out address>       (. CodeGen.OneByte(address.value); .)
              | "DC"
                  (   Address<out address>     (. CodeGen.OneByte(address.value); .)
                    | StringConst<out str>     (. for (int i = 0; i < str.Length; i++)
                                                    CodeGen.OneByte(str[i]); .)
                  )
***           | "DS" Address<out address>  (. if (address.isKnown)
***                                              CodeGen.ReserveBytes(address.value);
***                                            else
***                                              SemError("DS must have a constant address field"); .)
            ] .
```

with the CodeGen class modified to include

```
            public static void ReserveBytes(int length) {
            // Reserve length bytes of memory
***           if (length > 0) codeLen = ToByte(codeLen + length);
***           else Parser.SemError("Invalid value for DS");
            } // ReserveBytes
```

```
          public static void SetOrg(int start) {
          // Redefine location counter
***         if (start >= 0) codeLen = ToByte(start);
***         else Parser.SemError("Invalid value for ORG");
          } // ReserveBytes
```

A large nunber of submissions made a glaring error - `ORG` is not an executable operation - it is a directive!

B15 The assembler currently allows addresses to be formed by the simple addition and subtraction of terms, as in

```
                LDA  LIST + 1
                ADI  10 - TOTAL
```

but it does not allow the first (or maybe only) term in such an address field to be preceded by an optional minus, as in

```
                ADI  - A - B + 8
```

Show how to modify the system to allow this. [10 marks]

Solution:

The solution is not hard once one has thought about it for a bit, and if one has understood how the expression formed of several terms might generate several forward references. One approach is

```
        Address<out AdrRec adr>              (. AdrRec adr2 = null;
***                                             adr = null;
***                                             int sign = 1; .)
***     =  (        Term<out adr, 1>
***          | '-' Term<out adr, -1>         (. adr.value = -adr.value; .)
           )
```

Another way is as follows

```
        Address<out AdrRec adr>              (. AdrRec adr2 = null;
***                                             int sign = 1; .)
***     =  [ '-'                             (. sign = -1; .)
***        ] Term<out adr, sign>             (. adr.value   = sign * adr.value; .)
```

in either case, followed by the extant code:

```
           {   '+' Term<out adr2, 1>         (. adr.value   = adr.value + adr2.value;
                                                adr.isKnown = adr.isKnown && adr2.isKnown; .)
             | '-' Term<out adr2, -1>        (. adr.value   = adr.value - adr2.value;
                                                adr.isKnown = adr.isKnown && adr2.isKnown; .)
           } .
```

It is, however, easy to forget to "apply" the effect of the sign to the first call to `Term`.

B16 Most assemblers will allow an `EQU` directive that can be used to define a label to have a particular value - useful for giving names to magic numbers as in

```
           CR    EQU  13
                 LDI  CR
                 OTA
                 LDI  LF          ; IO.writeLine()  CR/LF
                 OTA
           LF    EQU  10
```

Show how to modify the system to allow this. Be careful - the label defined by an `EQU` directive might legally have been "used" before the directive is encountered. [16 marks]

Solution:

This is "harder" because `EQU` is a directive, not an opcode (which, again, several people missed, if indeed they got this far), and its label is not a label in the usual sense. In fact the solution is simple once one sees it, but one has to guard against `EQU` being used without a label!

```
          Statement                         (. AdrRec constant;
                                               Entry tableEntry;
                                               string name; .)
          =
              (   Unlabelled
                | Label<out name>           (. tableEntry = Table.Find(name);
                                               if (tableEntry != null && tableEntry.isDefined)
                                                 SemError("redefined label"); .)
                  (
                                            (. if (tableEntry == null)
                                                 Table.Insert(new Entry(name, CodeGen.CurrentAddress()));
                                               else if (!tableEntry.isDefined)
                                                 tableEntry.CompleteReferences(CodeGen.CurrentAddress()); .)
                       Unlabelled
***                  | "EQU"
***                    Address<out constant>  (. if (!constant.isKnown)
***                                              SemError("EQU must have a constant address field");
***                                            else if (tableEntry == null)
***                                              Table.Insert(new Entry(name, constant.value));
***                                            else if (!tableEntry.isDefined)
***                                              tableEntry.CompleteReferences(constant.value); .)
***               )
***           )
            | "ORG" Address<out constant>   (. if (constant.isKnown)
                                                 CodeGen.SetOrg(constant.value);
                                               else
                                                 SemError("ORG must have a constant address field"); .)
***         | "EQU"                         (. Parser.SemError("EQU must be labelled"); .)
***            Address<out constant> .
```

Section C

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

Candidates were reminded of the Assembler language that they had encountered in an earlier course, and were invited to create an assembler/interpreter that would generate VM code from such source.

An informal description (from the second year course) and a basic Cocol grammar for the language were provided. This grammar is reproduced below.

```
COMPILER Assem $NC
/* Describe Assembler language as used in CSC 201 */

IGNORECASE

CHARACTERS
  lf         = CHR(10) .
  cr         = CHR(13) .
  backslash  = CHR(92) .
  control    = CHR(0) .. CHR(31) .
  letter     = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  digit      = "0123456789" .
  binDigit   = "01" .
  hexDigit   = digit + "ABCDEFabcdef" .
  stringCh   = ANY - '"' - control - backslash .
  charCh     = ANY - "'" - control - backslash .
  printable  = ANY - control .

TOKENS
  decNumber  = digit { digit } .
  binNumber  = binDigit { binDigit } "%" .
  hexNumber  = digit { hexDigit } ( "H" | "h" ) .
  identifier = letter { letter | digit } .
  stringLit  = '"' { stringCh | backslash printable } '"' .
  charLit    = "'" ( charCh   | backslash printable ) "'" .
  EOL        = cr lf | lf .
  comment    = ";" { printable } .

PRODUCTIONS
  Assem             = Begin StatementSequence End .
  Begin             = "BEG" CommentEOL .
  End               = "END" CommentEOL .
  CommentEOL        = [ comment ] SYNC EOL .
  StatementSequence = { Statement CommentEOL } .
  Statement         = [ Label ] [ OneByteOp | TwoByteOp Address ] .
  OneByteOp        = (   "ASR"  | "CLA"  | "CLC"  | "CLV"  | "CLX"  | "CMA"  | "CMC"
                        | "DEC"  | "DEX"  | "HLT"  | "INA"  | "INB"  | "INC"  | "INH"
                        | "INI"  | "INX"  | "NOP"  | "OTA"  | "OTB"  | "OTC"  | "OTH"
                        | "OTI"  | "POP"  | "PSH"  | "RET"  | "SHL"  | "SHR"  | "TAX" ) .
  TwoByteOp        = (   "ACI"  | "ACX"  | "ADC"  | "ADD"  | "ADI"  | "ADX"  | "ANA"
                        | "ANI"  | "ANX"  | "BCC"  | "BCS"  | "BGE"  | "BGT"  | "BLE"
                        | "BLT"  | "BNG"  | "BNZ"  | "BPZ"  | "BRN"  | "BVC"  | "BVS"
                        | "BZE"  | "CMP"  | "CPI"  | "CPX"  | "DC"   | "DS"   | "JGE"
                        | "JGT"  | "JLE"  | "JLT"  | "JSR"  | "LDA"  | "LDI"  | "LDX"
                        | "LSI"  | "LSP"  | "ORA"  | "ORI"  | "ORX"  | "SBC"  | "SBI"
                        | "SBX"  | "SCI"  | "SCX"  | "STA"  | "STX"  | "SUB" ) .
  Address          = Term { '+' Term | '-' Term } .
  Term             = Label | IntConst | StringConst | CharConst | '*' .
  Label            = identifier .
  IntConst         = decNumber | binNumber | hexNumber .
  StringConst      = stringLit .
  CharConst        = charLit .

END Assem.
```

Candidates were provided with an exam kit for Java or C#, containing files like those which they had used in the practical course, and with skeleton files for creating a tailored assembler, as well as the source code for a complete VM emulator. The kit also contained the executable of a assembler/interpreter (developed in Pascal) that would allow them to compare the code generated by this program with that produced by the system they were developing. They were also given a suite of simple, suggestive test programs. Finally, 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 assembler system incorporating the features they had been asked to implement 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. No 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. The system as supplied at this point was deliberately naive in some respects, in order to lay the ground for the unseen questions of the next day.

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
```