# RHODES UNIVERSITY

## Computer Science 301 - 2010 - Programming Language Translation

Well, here you are. Here is the free information you have all been waiting for, with some extra bits of advice:

- Don't panic. It may be easier than you might at first think.
- The final solution(s) in Section B tomorrow will need rather careful thought. I am looking for evidence of mature solutions and insight, not crude hacks.
- Work in smaller, rather than larger groups. Too many conflicting ideas might be less helpful than a few carefully thought out ones.
- Do make sure you get a good night's sleep!

### How to spend a Special Sunday

From now until about 22h30 tonight, Computer Science 3 students have exclusive use of the Hamilton Laboratory. You are encouraged to throw out anyone else who tries to use it, but we hope that this does not need to happen. At about 22h30 Chris, Dameon, Jill and I may have to move some computers around and prepare things for Monday. If there is still a high demand we shall try to leave some computers for use until later, but by then we hope you will have a good idea of what is involved.

Once again, this year the format of the examination is somewhat different from years gone by, so as to counter what had happened where, essentially, one solution was prepared by quite a large group and then passed around to many others, who had probably not contributed to it in any real way. As in recent years (since 2007), the problem set below is only part of the story. At 16h30 you will receive further hints as to how this problem should be solved (by then you might have worked this out for yourselves, of course). You will be encouraged to study the problem further, and the hints in detail, because during the examination tomorrow you will be set further questions relating to the system - for example, asked to extend the solution you have worked on in certain ways. It is my hope that if you have really understood the material today, these questions will have solutions that come readily to mind, but of course you will have to answer these on your own!

My "24 hour exam" problems have all been designed so that everyone should have been able to produce at least the basic solution, with scope given for top students to demonstrate their understanding of the subtler points of parsing, scanning and compiling. Each year I have been astounded at the quality of some of the solutions received, and I trust that this year will be no exception.

*Please note that there will be no obligation to produce a machine-readable solution in the examination (in fact doing so is quite a risky thing, if things go wrong for you, or if you cannot type quickly). The tools will be provided before the examination so that you can experiment in detail. If you feel confident, then you are free to produce a complete working solution to Section B during the examination. If you feel more confident writing out a neat detailed solution or extensive summary of the important parts of the solution, then that is quite acceptable. Many of the best solutions over the last few years have taken that form.*

During the first few hours you will be able to find the following files in the usual places, or by following links from the course web page:

- All the files as were made available for the practical course.
- The files `FREE1J.ZIP` and `FREE1C.ZIP`, which contain the Java and C# versions of the Coco/R system, and its support files, a basic grammar and skeleton files and test data for today's exercise. The kit also contains a PDF version of the Coco/R user manual (`CocoManual.pdf`).

At 16h30 new versions of the exam kit will be posted (`FREE2J.ZIP` and `FREE2C.ZIP`), and a further handout issued, with extra information.

*Most of the machines in the labs will work in exactly the way they have done this semester. Some are set up to show you the configuration you can expect tomorrow. It is suggested that you work at one of those machines for a short time to make sure you know what to expect. To work on these machines you proceed as follows*

- Open up a DOS window following the usual route

- Give the command

  ```
  CONNECT  TESTxxx   yyyyy
  ```

  where `xx` is a three digit number for the machine (typically in the range `101` through `123`) and `yyyy` is a 5 digit password.

- This will then allow you to log onto the `J:` drive, where you will find the files `FREE1J.ZIP` (and `FREE1C.ZIP`) waiting for you.

- Give a command like

  ```
  UNZIP   FREE1J.ZIP
  ```

  to unpack the "exam kit" of choice.

From here on things should be familiar. You could, for example, log onto the `D:` or `J:` drive, use `UltraEdit` use `CMAKE` and `CRUN` … generally have hours of fun. The C# system will work only on the `D:` drive, however.

But note that the exam set up has *no* connection with the outside world - no Google, FaceBook, ftp client, telnet client, shared directories - not even a printer!

Today you may use the files and either the "usual" or the "exam" systems in any way that you wish, subject to the following restrictions: *Please observe these in the interests of everyone else in the class*.

(a) When you have finished working, **please** delete any files from the D: drive, so that others are not tempted to come and snoop around to steal ideas from you.

(b) You are permitted to discuss the problem with one another, and with anybody not on the "prohibited" list.

(c) You are also free to consult books in the library. If you cannot find a book that you are looking for, it may well be the case that there is a copy in the Department. Feel free to ask.

(d) Please do not try to write any files onto the C: directory, for example to `C:\TEMP`

(e) If you take the exam kit to a private machine you will need to have Java installed (or the .NET framework or equivalent to use the C# version).

I suggest that you *do* spend some of the next 24 hours in discussion with one another, and some of the time in actually trying out your ideas. You have plenty of time in which to prepare and test your ideas - go for it, and good luck. **Remember that tomorrow you may not bring anything into the room other than your student card and writing utensils, and especially not listings, diskettes, memory sticks, text books or cell phones.**

If you cannot unpack the file, or have trouble getting the familiar tools to work (unlikely!), you may ask me for help. You may also ask for explanation of any points in the question that you do not understand, in the same way that you are allowed to ask before the start of an ordinary examination. You are no longer allowed to ask me questions about any other part of the course. Sorry; you had your chance earlier, and I cannot allow this without risking the chance of sneak questions and suggestions being called for.

If you cannot solve the problem completely, don't panic. It has been designed so that I can recognize that students have reached varying degrees of sophistication and understanding.

## How you will spend a Merry Monday

Before the start of the formal examinations the laboratory will be unavailable. During that time

- The machines will be completely converted to a fresh exam system, with no files left on directories like `D:` or `C:\TEMP` .

- The network connections will be disabled.

At the start of the examination session:

- You will receive ordinary answer books for the preparation of answers, and an examination paper.

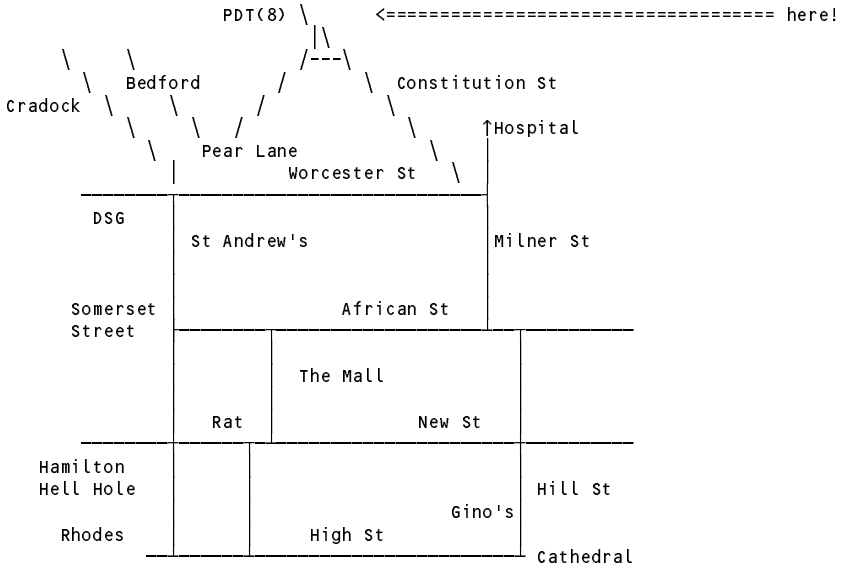
- You will receive a copy of at least the relevant portions of the grammar and support files that you received at 16h30. *You may annotate these during the exam to form part of your solution if you wish to submit a hand-written answer to Section B and need to make reference to the code (possibly by line number).* In this case you should hand in the annotated listings with your answer book.

- You will be allocated to a computer and supplied with a `CONNECT` command for your own use. Once connected you will find an exam kit on the `J:` drive. This will contain the same Coco/R system and other files you have been given today, and in addition there will be "flat ASCII" machine readable parts of the examination paper itself, in files with names like `Q7.TXT` (Question 7). *There is no obligation to use a computer during the exam. You can answer on paper if you prefer - and yes, you can write in pencil if you prefer that.*

- At the end of the exam you will be given a chance to copy any files that you have edited or created on the D: drive back to the server. This will be explained tomorrow.

## Cessation of Hostilities Party

As previously mentioned, Sally and I would like to invite you to an informal end-of-course party at our house on 15 November. There's another copy of the map below to help you find your way there. It would help if you could let me know whether you are coming so that we can borrow enough glasses, plates etc. *Please post the reply slip into the hand-in box during the course of the day.* Time: from 18h30 onwards. Dress: Casual




## Section B [ 70 marks ]

## Extending Parva to allow for enumeration types

Most computer languages provide simple, familiar, notations for handling arithmetic, character and Boolean types of data. Variables, structures and arrays can be declared of these basic types; they may be passed from one routine to another as parameters, and so on.

Some languages, notably Pascal, Modula-2, C, C++, and Ada, allow programmers the flexibility to define what are often known as *enumeration types*, or simply *enumerations*. Here are some examples to illustrate this idea:

```
TYPE  (* Pascal or Modula-2 *)
  COLOURS = ( Red, Orange, Yellow, Green, Blue, Indigo, Violet );
  INSTRUMENTS = ( Drum, Bass, Guitar, Trumpet, Trombone, Saxophone, Bagpipe );
VAR
  Walls, Ceiling, Roof : COLOURS;
  JazzBand : ARRAY [0 .. 40] OF INSTRUMENTS;
```

or the equivalent

```
typedef  /* C or C++ */
  enum { Red, Orange, Yellow, Green, Blue, Indigo, Violet } COLOURS;
typedef
  enum { Drum, Bass, Guitar, Trumpet, Trombone, Saxophone, Bagpipe } INSTRUMENTS;
COLOURS      Walls, Ceiling, Roof;
INSTRUMENTS  JazzBand[41];
```

Sometimes the variables are declared directly in terms of the enumerations:

```
VAR  (* Pascal or Modula-2 *)
  CarHireFleet : ARRAY [1 .. 100] OF ( Golf, Tazz, Sierra, BMW316 );

enum CARS { Golf, Tazz, Sierra, BMW316 } CarHireFleet[101];  /* C or C++ */
```

Java got into the act rather later; the original version of Java did not provide the facility that is now manifest in Java and C#, where we might declare:

```
enum COLOURS { Red, Orange, Yellow, Green, Blue, Indigo, Violet };
enum INSTRUMENTS { Drum, Bass, Guitar, Trumpet, Trombone, Saxophone, Bagpipe };
COLOURS      Walls, Ceiling, Roof;
INSTRUMENTS[] JazzBand = new INSTRUMENTS[41];
```

The big idea here is to introduce a distinct, usually rather small, set of values which a variable can legitimately be assigned. Internally these values are represented by small integers - in the case of the `CarHireFleet` example the "value" of `Golf` would be 0, the value of `Tazz` would be 1, the value of `Sierra` would be 2, and so on.

In the C/C++ development of this idea the enumeration, in fact, results in nothing more than the creation of an implicit list of `const int` declarations. Thus the code

```
   enum CARS { Golf, Tazz, Sierra, BMW316 } CarHireFleet[101];
```

is semantically completely equivalent to

```
const int Golf = 0; const int Tazz = 1;
const int Sierra = 2, const int BMW316 = 3;
int CarHireFleet[101];
```

and to all intents and purposes this gains very little, other than possible readability - an assignment like

```
CarHireFleet[N] = Tazz;
```

might, of course, convey more to a reader than the semantically identical

```
CarHireFleet[N] = 1;
```

In the much more rigorous Pascal and Modula-2 approach one would not be allowed this freedom; one would be forced to write

```
CarHireFleet[N] := Tazz;
```

Furthermore, whereas in C/C++ one could write code with rather dubious meaning like

```
CarHireFleet[4] = 45;            /* Even though 45 does not correspond to any known car! */
CarHireFleet[1] = Tazz / Sierra;  /* Oh come, come! */
Walls = Sierra;                  /* Whatever turns you on is allowed in C++ */
```

in Pascal, Modula-2, Java and C# one cannot perform arithmetic on variables of these types directly, or assign

values of one type to variables of an explicitly different type. In short, the idea is to promote "safe" programming - if variables can meaningfully only assume one of a small set of values, the compiler (and/or run-time system) should prevent the programmer from writing (or executing) meaningless statements.

Clearly there are some operations that could have sensible meaning. Looping and comparison statements like

```
if (Walls == Indigo) Redecorate(Blue);
```

or

```
for (Roof = Red; Roof <= Violet; Roof++) DiscussWithNeighbours(Roof);
```

or

```
if (JazzBand[N] >= Saxophone) Shoot(JazzBand[N]);
```

might reasonably be thought to make perfect sense - and would be easy to "implement" in terms of the underlying integer values.

In fact, the idea of a limited enumeration is already embodied in the standard character (and, in some languages, Boolean) types - type Boolean is, in a sense the enumeration of the values {0, 1} identified as {false, true}, although this type is so common that the programmer is not required to declare the type explicitly. Similarly, the character type is really an enumeration of a sequence of (typically) ASCII codes, and so on.

Although languages that support enumeration types forbid programmers from abusing variables and constants of any enumeration types that they might declare, the idea of "casting" allows programmers to bypass the security where necessary. The reader will be familiar with the (rather strange) notation in the C family of languages that allows code like

```
char uc = (char) 65;                // 'A'
char lc = (char) ( (int) uc + 32 );  // 'a'
```

In Pascal and Modula-2 a standard function ORD(x) can be applied to a value of an enumeration type to do little more than cheat the compiler into extracting the underlying integral value. This, and the inverse operation of cheating the compiler into thinking that it is dealing with a user-defined value when you want to map it from an integer, are exemplified by Modula-2 code like

```
IF (ORD(Bagpipe) > 4) THEN .....
Roof := VAL(COLOURS, I + 5));
```

Rather annoyingly, in Pascal and Modula-2 one cannot read and write values of enumeration types directly - one has to use these casting functions and switching statements to achieve the desired effects.

Enumerations are a "luxury" - clearly they are not really *needed*, as all they provide is a slightly safer way of programming with small integers. Not surprisingly, therefore, they are not found in languages like Oberon (simplified from Modula-2).

In recent times you have studied and extended a compiler for a small language, Parva, in the implementation of which we have repeatedly stressed the ideas and merits of safe programming.

How would you add the ability to define enumeration types in Parva programs and to implement these types, at the same time providing safeguards to ensure that they could not be abused? Initially, strive to develop a system that will allow

- the declaration of an enumeration type name and its associated values
- the declaration of variables (and arrays) of those types
- assignment of assignment-compatible values of those types to these variables
- comparison of values of compatible types for equality and ordering
- casting between types using a notation like

```
Roof = cast(COLOURS, someIntegerValue);
```

- incrementing and decrementing variables using the familiar `++` and `--` notation
- *for* loops that can be controlled by variables of an enumerated type as well as by integers and characters
- values of enumerated types to be written (for simplicity, as the value of the underlying integer).

You may wish to read up a little more on enumeration types as they are used in languages like Modula-2 and Pascal. Enumeration types in Java and C# are rather more complex in their full ramifications, however.

## Hints:

- The examination kit includes all the files needed to build a Coco-defined compiler *similar* to the one used in in practical 25 (but not the complete "solution"). As usual, there is a Java version of the kit and an equivalent C# version.

- It gives little away to advise you to think carefully about the form the symbol table and its entries will take.

- The example Parva programs in the source kit range from very short, simple ones, to more advanced ones. You are well advised to start with the very simple ones that are only a few statements long!

- A simple batch file `PARVA.BAT` has been provided in the Java kits to make testing easier - after making the system, a command like `PARVA t01.pav` may be used in place of `CJRUN Parva t01.pav`. C# users will already know to give the command `Parva t01.pav`, of course.

- The systems allow you to use a command-line parameter `-D` to display the symbol table at the close of every block, and a parameter `-C` to request a listing of the generated code, which you might find useful.

- Later in the day - at 16h30 - we shall release more information, to help those of you who may not have completed the exercise to do so. Section B of the examination tomorrow will include a set of unseen questions probing your understanding of the system.

- Rest assured that you will not be expected to reproduce a complete system from memory under examination conditions, but you may be asked to make some additions or improvements to the system developed today.

- Remember Einstein's Advice: "Keep it as simple as you can but no simpler" and Terry's Corollary: "For every apparently complex programming problem there is an elegant solution waiting to be discovered".

  Good luck!

The most complicated test program in the kit is given below, but there are several simpler ones that you could use to develop the system in an incremental fashion, with names like `t01.pav, t02.pav` etc.

```
void main () { // enumtest.pav
// Illustrate some simple enumeration types in extended Parva
// Some valid declarations

  enum DAYS    { Mon, Tues, Wed, Thurs, Fri, Sat, Sun };
  enum WORKERS { BlueCollar, WhiteCollar, Manager, Boss };
  enum DEGREE  { BSc, BA, BCom, MSc, PhD };
  enum FRUIT   { Orange, Pear, Banana, Grape };

  const pay = 100;

  DAYS yesterday, today;
  WORKERS[] staff = new WORKERS[12];
  int[] payPacket;
  int i;
  bool rich;
  FRUIT juice = Orange;
  DEGREE popular = BSc;

// Some potentially sensible statements
  today = Tues;
  yesterday = Mon;                                    // That follows!
  if (today < yesterday) write("Compiler error");     // Should not occur
  today++;                                            // Working past midnight?
  if (today != Wed) write("another compiler error");

  int totalPay = 0;
  for (today = Mon; today <= Fri; today++)
```

```
    totalPay = totalPay + pay;
  for today = Sat to Sun
    totalPay = totalPay + 2 * pay;

  rich = staff[i] > Manager;
  yesterday = cast(DAYS, (int) today - 1);
  DAYS tomorrow = cast(DAYS, (int) today + 1);

// Some possible meaningless statements - be careful
  enum COLOURS { Red, Orange, Green };  // Is this valid?
  juice = cast(FRUIT, Pear);           // Is this valid?
  juice = cast(FRUIT, popular);        // Is this valid?
  Sun++;                               // Cannot increment a constant
  today = Sun; yesterday = today - 1;  // Sounds reasonable?
  if (today == 4)                      // Invalid comparison - incompatibility
    staff[1] = rich;                   // Invalid assignment - incompatibility
  Manager = Boss;                      // Cannot assign to a constant
  payPacket[Boss] = 1000;              // Is this a valid subscript expression?
  payPacket[tomorrow] = 100000;        // Is this a valid subscript expression?
}
```

# Free information

## Summary of useful library classes

The following summarizes some of the available simple I/O classes in Java (The C# ones are equivalent, but for small differences in `MethodNames` and `methodNames`). Note that the input methods allow you to specify a base - these methods have only been added to the library very recently. The `IO` library has static versions of the methods in the `InFile` and `OutFile` classes, but these should be familiar to you.

```
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 long    readLong(int radix)
  public short   readShort()
  public short   readShort(int radix)
  public byte    readByte()
  public byte    readByte(int radix)
  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 boolean 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 boolean 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
```