

RHODES UNIVERSITY

Computer Science 301 - 2008 - 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, 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 and I will 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.

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 2007, the problem set below is only part of the story. At 16h00 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, an attributed grammar for a version of Parva, very similar to, but not identical with that in the last practical 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 16h00 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 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.*

- 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.
- You are permitted to discuss the problem with one another, and with anybody not on the "prohibited" list.
- 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.
- Please do not try to write any files onto the C: directory, for example to C:\TEMP
- 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

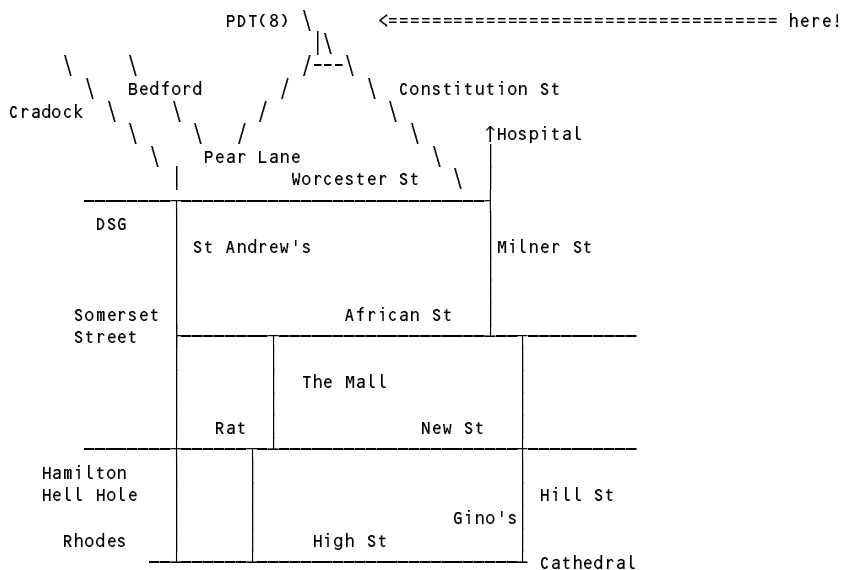
- The machines will be completely converted to a fresh exam system, with no files left on directories like D: or C:\TEMP.
- The network connections will be disabled.

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 16h00. *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 10 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 [80 marks]

During the Programming Language Translation course in 2008 we have repeatedly encountered the PVM - the Parva Virtual Machine - a hypothetical stack-based computer, for which we have explored a simple machine emulator, an assembler (using conventional mnemonics for the opcodes) and a Parva-to-PVM code compiler.

Stack based machines, and stack based machine interpreters and emulators have been at the heart of our subject for many years. In the early 1960s a stack-based low-level language known as FORTH became very popular, mainly due to one Charles Moore, a radio astronomer, who developed the language and its interpreter to control equipment, all of which had to be done within the confines of the very small computer systems of that period. Since then, Forth has continued to attract a devoted band of followers, and it is still quite widely used. In fact, Forth systems were heavily reliant on an interpreter, and on a technique known as "threaded interpretive code", although in principle other compilers and systems could be developed.

Some examples of Forth-like code will show that it appears to be very similar to PVM code, except that it uses a

different system of mnemonics. For example, the code

```
40 5 + 3 * .
```

computes the value of $(40 + 5) * 3$ and then prints it, in the same way that

```
LDC 40
LDC 5
ADD
LDC 3
MUL
PRNI
```

does. And the sequence

```
C A @ B @ / !
```

corresponds to the high-level code $C = A / B$, as does the sequence

```
LDA C
LDA A
LDV
LDA B
LDV
DIV
STO
```

(where, of course, the "variables" A, B and C are assumed to have been mapped into memory at predicted offsets from a frame stack pointer).

Forth programmers think of this sort of source code as consisting of a stream of "words" (many of which to you will seem to correspond directly to a PVM opcode). Assembly from this code to PVM code can proceed simply by identifying each of the words (looking them up in a so-called "dictionary" in which they have first been entered), and making obvious substitutions.

Interestingly, the words in a system like this can take on a variety of lexical forms. Besides some defined by simple characters like + - * and /, there are some that correspond to actual words like NEGATE and MAX, as well as some more cryptic ones like ?, @, ! and []. Words are demarcated clearly by white space, (comments appear in simple parentheses) and the language is deemed to be case-insensitive.

The table below shows the correspondence between the standard words we propose and the PVM opcode set as we have used it in several practicals of the course (our words are not identical to those found in Forth, but are very close).

!	PVM.sto	.	PVM.prni
@	PVM.ldv	.C	PVM.prnc
+	PVM.add	.B	PVM.prnb
-	PVM.sub	CR	PVM.prnl
*	PVM.mul	++	PVM.inc
/	PVM.div	--	PVM.dec
NEGATE	PVM.neg	HALT	PVM.halt
REM	PVM.rem	NEW	PVM.anew
AND	PVM.and	[]	PVM.ldxa
OR	PVM.or	STK	PVM.stk
==	PVM.ceq	?	PVM.inpi
!=	PVM.cne	?C	PVM.inpc
<=	PVM.cle	?B	PVM.inpb
>=	PVM.cge	TRUE	PVM.ldc 1
<	PVM.clt	FALSE	PVM.ldc 0
>	PVM.cgt	CAP	PVM.cap
NOT	PVM.not	ISLD	PVM.isld

What makes the system rather different from ones you may have seen previously is that the dictionary is not static. As strings, integers and characters, constants and variables are introduced, so new words are added to the

dictionary. So, for example, the sequence

```
VARIABLE A 42 . "Hello World" A 42 ! "Hello World" HALT
```

introduces the identifier "A" as a variable name and adds it to the end of the dictionary, with the implication that quoting A thereafter will instigate a dictionary search, find the entry, and then generate an LDA opcode with the appropriate offset. The appearance of the integer 42 will add the word "42" to the dictionary, with the implication that when a 42 is encountered from then on it will be recognized again, and generate an LDC 42 operation. Similarly, introduction of a string like "Hello World" will add this as a new word in the dictionary, with the implication that whenever this string is encountered it will be recognized as corresponding to, and generating a PRNS X instruction, where X corresponds to the position of the actual string in the string pool of the PVM. Assembling the above "program" would thus generate code equivalent to

```
0 DSP 1 ; A is at offset 0; one variable
2 LDC 42 ; push 42
4 PRNI ; print it
5 PRNS 0 ; print "Hello World"
7 LDA 0 ;
9 LDC 42 ;
11 STO ; A = 42
12 PRNS 0 ; print "Hello World" a second time (same string!)
14 HALT ;
```

The dictionary is searched from the last entry towards the first, which also gives rise to the interesting concept that one can redefine words - for example

```
CONSTANT AND 67
```

would mean that AND would be taken from that point onwards as a word that generated an LDC 67 operation rather than a logical conjunction operation.

Besides the standard words that map easily directly onto PVM opcodes, there are also ones like IF and ENDIF and REPEAT and UNTIL that imply the presence of control structures, and which have to give rise to appropriate BRN and BZE opcodes. In the system being proposed here, a Parva code fragment like

```
int age;
const retirement = 60;
read(age);
if (age > retirement)
    write("The meaning of life was really ", 42);
```

would be coded as

```
VARIABLE age
CONSTANT retirement 60
age ? ( read age )
age @ retirement > IF ( age > 60? )
    "The meaning of life was really " 42 . ( write("The meaning of life was really ", 42) )
ENDIF
HALT
```

and produce the code

```
0 DSP 1 ; age is at offset 0; one variable
2 LDA 0
4 INPI ; read(age)
5 LDA 0
7 LDV
8 LDC 60 ; 60 is the value of retirement
10 CGT
11 BZE 17 ; if (age > retirement) {
12 PRNS 0 ; write("The meaning of life was really ")
14 LDC 42 ; write(42);
16 PRNI ; }
17 HALT
```

Similarly, the sample program on page 58 of the textbook

```

Total = 0;
REPEAT read(x); Total := x + Total; UNTIL x == 0;
write("Total is ", Total);

```

would be coded as

```

VARIABLE x VARIABLE Total
Total 0 ! ( Total = 0 )
REPEAT
  x ? ( Read(x) )
  Total x @ Total @ + ! ( Total = x + Total )
  x @ 0 == ( x == 0 ? )
UNTIL
"Total is" Total @ . ( write ("Total is" , Total) )
HALT

```

and produce the code

```

0  DSP  2  ; x is variable 0, Total is variable 1
2  LDA  1
4  LDC  0
6  STO   ; Total = 0;
7  LDA  0 ; repeat
9  INPI  ; read(x);
10 LDA  1
12 LDA  0
14 LDV
15 LDA  1
17 LDV
18 ADD
19 STO   ; Total = x + Total;
20 LDA  0
22 LDV
23 LDC  0
25 CEQ   ; (* check equality of x and 0 *)
26 BZE  7 ; until (x == 0);
28 PRNS 0 ' write ("Total is");
30 LDA  1
32 LDV
33 PRNI  ; write(Total);
34 HALT

```

Your challenge in the next 24 hours is to develop an assembler/interpreter for generating PVM code from our proposed version of Forth-like source code. Limit yourself to the features discussed above (the words corresponding directly to PVM opcodes, integer and character constants (both implicit and named), variables, strings and simple IF .. ENDIF and REPEAT ... UNTIL structures).

Hints:

- To refresh your memory, the examination kit includes all the files needed to build a working Parva compiler similar to the one used in your last practical exercise (it does not have all the extensions you were asked to make in that practical, and you need not bother to try to add all of those extensions again). As usual, there is a Java version and an equivalent C# version. You may use either version, and build the compiler with a command `CMAKE Parva`.
- There are also outline grammars and support files for the assembler, which you will be able to make after you have developed them further. Once again, there is a Java version and an equivalent C# version. You may use either version, and build the assembler with the command `CMAKE Assem`. Take care not to confuse the files for Parva with the rather similar files for Assem.
- You are advised to study the outline grammar and support files before you rush in to code, and to think carefully about the form the Dictionary will take. It will suffice to use the generic `ArrayList` (Java) or `List` (C#) class as the basis of the Dictionary. Bearing in mind that looping and decision structures may be nested, you may also find the `Stack` class of use. A synopsis of these classes can be found on the course web page, and will be supplied again during the examination proper.
- Note that the standard words like `NEGATE` are not key or reserved words in the usual sense - they are also to be stored in the Dictionary.

- (e) The kit also includes various simple test programs of the sort outlined above (in files named `egXX.asm`) along with the corresponding PVM code (in files named `egXX.PVM`, to distinguish them from files named `egXX.COD` which will be produced by the assembler system itself). A listing of these files has been provided in a separate handout.
- (f) Two simple batch files `PARVA.BAT` and `ASSEM.BAT` have been provided in the java kits to make testing easier - after making the system a command like `ASSEM eg01.asm` may be used in place of `CRUN Assem eg01.asm`. C# users will already know to give the command `Assem eg01` in place of using `CRUN`.
- (g) A simple system for helping to check on `TOKENS` definitions has been provided in the grammar `Tokens.atg`. You may find it useful to make this system and then to run it using the test data in the file `Tokens.txt`, and to modify it to help you secure the token definitions you need for the present exercise.
- (h) The original Parva compiler stored string constants character by character in high memory, as you should remember. However, it is suggested that you do not try to incorporate that idea, but rather use the one developed in later practicals, in which we created a "string pool" dynamically, using the `ArrayList` (or `List`) class. Store string values in this pool, and use the index into the pool as the "value" of a string.
- (i) You will recall that the PVM was easily extended to support further opcodes. Two other useful Forth words not currently supported are `DUP` (which has the effect of duplicating the top of stack) and `ABS` (which replaces the value currently on the top of stack by its absolute value). While you are at it, and to refamiliarise yourself with the PVM, add these words and opcodes to your system.
- (j) Later in the day - at 16h00 - 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.
- (k) Rest assured that you will not be expected to reproduce a complete Forth-like assembler system from memory under examination conditions, but you may be asked to make some additions or improvements to the system developed today.
- (l) 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!

Free information

Summary of useful library classes

The following summarizes some 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(int 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(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(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(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 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
    sb = new StringBuffer();         // build strings (Java 1.4)
    sb1 = new StringBuffer("original"); //
StringBuilder
    sb = new StringBuilder();         // build strings (Java 1.5 and 1.6)
    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
    st = new StringTokenizer(s, ".,"); // tokenize strings
    st = new StringTokenizer(s, ".,", true); // delimiters are . and ,
    while (st.hasMoreTokens())         // delimiters are also tokens
        process(st.nextToken());       // process successive tokens

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

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 and stack 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
```

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

```
import java.util.*;

class Stack<E>
// class for constructing a stack of elements of type E

    public Stack<E>()
    // Empty stack constructor

    public void push(E element)
    // Appends element to top of stack

    public E pop()
    // Retrieves an element from top of stack

    public E peek()
    // Returns element on top of stack without removing it

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

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

    public boolean empty()
    // Returns true if stack is empty

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

} // Stack
```

Simple list and stack 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
```

The following is the specification of useful members of a C# 2.0 generic stack handling class

```
using System.Collections.Generic;

class Stack<E>
// Class for constructing a stack of elements of type E

    public Stack<E>()
    // Empty stack constructor

    public void Push(E element)
    // Appends element to top of stack

    public E Pop()
    // Retrieves an element from top of stack

    public E Peek()
    // Returns element on top of stack without removing it

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

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

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

} // Stack
```