RHODES UNIVERSITY

Computer Science 301 - 2005 - 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 is probably easier than you might at first think.
- The solution to section B needs 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 Jody 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 how to solve the problem below. During this time 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 `EXAM.ZIP` (or `EXAMC.ZIP`) which contains the Java (or C#) versions of the Coco/R system, and its support files, and an unattributed grammar for Parva, including the ability to recognize functions with parameters (the grammar relevant to Chapter 14). It also contains a PDF version of the Coco/R user manual (`CocoManual.pdf`).

Most of the machines in the labs will work in exactly the way they have done this semester. A few are set up to show you the configuration you can expect to find 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   TESTxx    demo
  ```

  where `xx` is the two digit number for the machine (typically in the range `01` through `26`).
- This will then allow you to log onto the `J:` drive, where you will find the file `EXAM.ZIP` (and `EXAMC.ZIP`) waiting for you.
- Give a command like

  ```
  UNZIP   EXAM.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.

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.*

(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 really good solutions - 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.

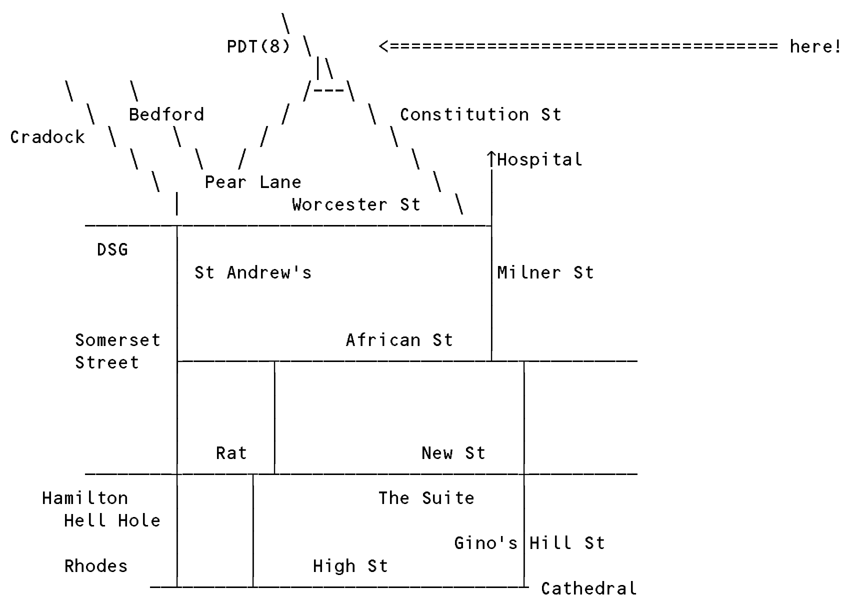
- You will receive a copy of the complete listing of the Parva grammar that accompanies this handout. *You may annotate this during the exam to form part of your solution if you wish to submit a hand-written answer to Section B and need to make reference to the code (possibly by line number).* In this case you should hand in the annotated listing 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 21 November. There's a 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



And now for the next trick - taken as it appears in tomorrow's examination paper:

Section B [90 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.

It had to happen! Over the last eight weeks a team of over 60 dynamic young programmers (DYPs) have been developing sophisticated applications in the new wonder language Parva. Like all DYPs, they have paid far too little attention to documenting what they have done. Unfortunately the day of reckoning is upon them: a directive has gone out from the Project Documentation Tyrant (PDT) that in a mere 24 hours time they will be required to provide a full set of documentation for their code, preferably in the form of a set of HTML (web) pages, one for each completed program.

Their PM (Project Manager) recently browsed through a text on Java and was struck by the existence of a utility called *JavaDoc*. This is a system program that can parse Java source code files and extract documentation from it, if such documentation is embedded in the files in a systematic way. Any "documentation" stored in a source code file must, of course, be distinguishable from the "real" source code which a compiler would process, so the system relies on extracting the documentation from comments - which, however, must not be confused with other commentary in the program. For the JavaDoc utility this is achieved by allowing commentary to take one of three forms:

```
/*  This is a typical comment in familiar form  */

//  This is a typical simple one-line comment in familiar form

/** This is a special JavaDoc comment, where the leading ** acts as
    a special marker to distinguish it from a simple comment
*/
```

Of course, JavaDoc comments would be ignored (along with other comments) if the programs were to be compiled by the standard Java compiler itself.

After consulting the PDT, the PM has asked that the team can meet the deadline by developing a utility called ParvaDoc that will do the same sort of thing for Parva. She realises that the ParvaDoc system can be generated by using Coco/R to create a "compiler" that will parse Parva programs - but rather than generate PVM code, this compiler will ignore most of the usual source code and concentrate almost entirely on certain of the comments. The PDT has agreed that the submission of an attributed Cocol grammar for the ParvaDoc utility will serve as proof that the deadline can be met.

Very generously, the PDT has supplied some examples of simple Parva programs marked up in this way, along with the form of output that he suggests should be generated - and, of course, he has also provided a simple Cocol version of the basic grammar for Parva, a copy of the Coco/R compiler generator, and a fully working executable for a Parva compiler. All this should help those 65 DYPs whose qualifications are at stake.

The crucial "ParvaDoc" comments are bound by structural rules of their own. For the task at hand, assume that they can appear only (a) immediately at the start of the code or (b) immediately at the start of a function or method. These comments are intended to document such things as the overall purpose of the application and/or its individual functions, the author, the version, the intent of the parameters or arguments, the value returned (for functions other than "void" ones), and possible references to other documents or resources.

Here is a typical example of ParvaDoc commentary for a function:

```
int ReduceLength(int[] list, int n, int length) {

/** @purpose      Reduces a list of integer numbers by removing every n-th number
    @param length specifies the original length of the list
    @return       length of reduced list
    @param n      specifies that every n-th number is to be removed
    @see          "Algorithms for a Special Sunday" (2005)
    @param list   is a reference to the list of numbers
    @version      1234
    @author       PDT
*/
  ...
}
```

As can be seen, the different components of the documentation are introduced by tags like `@purpose` and `@author`. Not all of these sections need be present, and they may be introduced in any order.

After processing the source file, the ParvaDoc system should produce output like

```
int ReduceLength (int[] list, int n, int length)

Purpose:
 Reduces a list of integer numbers by removing every n - th number

Author:
 PDT

Version:
 1234

References:
 "Algorithms for a Special Sunday" (2005)

Returns:
 length of reduced list

Parameters:

 length specifies the original length of the list

 n specifies that every n - th number is to be removed

 list is a reference to the list of numbers
```

in which we can note that the output consists of the function signature, followed by a sorted and nicely formatted list of the components forming the documentation. (A more extensive set of examples of input and output can be found in the files that have been provided by the PDT.)

The system should provide some minimal checking - for example, that tags like `@param` and `@return` are not applied to an introductory ParvaDoc comment describing an application as a whole, and that the identifiers quoted at the start of any `@param` clauses match those in the function signature. An example of the output from an attempt to analyse an incorrectly documented application might be

```
     1 /**
     2     @author  P.D. Terry, Rhodes University, 2005
     3     @return  Tomorrow's oil price
  ****     ↑ not applicable here
     4     @param   Humidity in Gulf of Mexico
  ****     ↑ not applicable here
     5 */
     6
     7 void Speculate(int[] sharePrice, int n) {
     8 /**
     9     @param   max is the gold price two weeks from today
  ****              ↑ max not in parameter list
    10     @return  to sanity after Wall Street crashes
  ****     ↑ void functions cannot return values
    11 */
    12 }
    13
    14 /** ParvaDoc comments cannot appear in arbitrary places between functions! */
  **** ↑ EOF expected
```

Producing a simple text file (say `Applic.txt`) from a program (say `Applic.pav`) that contains such documentation might be a useful first step. However, as mentioned earlier, a better system would produce an HTML version (say `Applic.htm`) with HTML tags inserted to allow a browser to display the documentation in a convenient form. This is not particularly difficult to do. If you go on to derive a system to generate this format you might like to study the simple web page supplied in the files kit for the examination, which exemplifies the sorts of HTML tags that might be incorporated into the output file.

Lastly, a system like ParvaDoc is easily enhanced to allow users to insert their own HTML tags into the documentation comments. For example, if the ParvaDoc comment above were extended to have clauses reading

```
    int ReduceLength(int[] list, int n, int length) {
    /** ...
        @see @<a href=sunday.htm>"Algorithms for a Special Sunday" (2005)@</a>
        @author PDT @<i>with a little help from my friends@</i>@<ul>
            @<li>John
            @<li>Paul
            @<li>George
            @<li>Ringo
        @</ul>
        ...
    */
```

a hyperlink to "`sunday.htm`" would appear in the web page in the "References:" section, and the names of the four friends would be rendered by the browser as a bulleted list in the "Author:" section. It is suggested that HTML tags can be introduced into commentary in this way with an `@` character immediately preceding the tag, in the same way that the `@` character is the first one in a tag like `@see`.

Hints: Do not alter the basic Parva grammar more than is absolutely necessary - do not be tempted to add the sorts of extensions that have formed part of the practical course this year. If you cannot develop a solution that correctly deals with all three kinds of comments, try to develop one assuming that the source program will only incorporate "ParvaDoc" comments. It is not required that you produce a solution that can derive both "simple text" and/or "html" text - develop a solution for one of these only.

Now that you have got over the shock…

It may be helpful to make a few suggestions as to how best to tackle this exercise, and the examination itself.

(a) I trust that it is obvious that you should use the Parva grammar as the basis of your solution.

(b) Do not alter the basic Parva grammar more than is absolutely necessary - do not be tempted to add the sorts of extensions that have formed part of the practical course this year. As with the EBNF cross-referencer, there are, in fact, relatively few places where the grammar may have to be altered and the actions added.

(c) The grammar has been called "ParvaDoc" - this will allow you to run the supplied Parva compiler itself if you feel you need to do so.

(d) The exam kit contains quite a large number of simple examples of Parva programs with "ParvaDoc" commentary and files showing the sort of output that a completed system would produce. They appear on an attached listing in an order that you may find useful in completing the exercise in an incremental fashion.

(e) If you cannot develop a solution that correctly deals with all three kinds of comments in Parva programs, try to develop one assuming that the Parva source programs will *only* incorporate "ParvaDoc" comments.

(f) If you work towards a solution that will produce "HTML" output you may find it useful to look at the attached summary of the format that a web page takes. There are relatively few "tags" needed for this application.

(g) It is not required that you produce a solution that can derive both "simple text" and/or "html" text - develop a solution for one of these only.

(h) You have been supplied with a listing of the Parva grammar (`ParvaDoc.atg`, and the `Parva.frame` file and you will receive a copy of this listing again during the examination tomorrow. Although you are free to write your answers "in any medium except red ink" tomorrow, I suggest that the best way to present your answer in the exam itself may be to mark up these listings with the changes you suggest - there is plenty of space to do so. I suspect that trying to type in the alterations during the exam will take you far too long. The point of the exercise is not to see how accurately you can type, but to demonstrate to the examiners that you understand how to use Coco to help develop a simple syntax-directed application.

(i) Having said that, you will surely find it very useful today (Sunday) to see how much of the answer you can get to work.

(j) Your answers should be self-contained. It will not convince the examiners if (for example) you write down claims like "we could do this part as we did in Task 4 of Practical 24".

Summary of useful library classes

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

```
class SymSet  { // simple set handling routines
  public SymSet()
  public SymSet(int[] members)
  public boolean equals(Symset s)
  public void incl(int i)
  public void excl(int i)
  public boolean contains(int i)
  public boolean isEmpty()
  public int members()
  public SymSet union(SymSet s)
  public SymSet intersection(SymSet s)
  public SymSet difference(SymSet s)
  public SymSet symDiff(SymSet s)
  public void write()
  public String toString()
} // SymSet

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

class ArrayList {   //  Maintenance of simple lists of objects
  public ArrayList()
  public void clear()
  public int size()
  public boolean isEmpty()
  public void add(Object o)
  public Object get(int index)
  public Object remove(int index)
} // ArrayList
```

Strings and Characters in Java

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

```
import java.util.*;

class demo {
  public static void main(String[] args) {
    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]
    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
      sb = new StringBuffer(),                //
      sb1 = new StringBuffer("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]
    sb.replace(i1, i2, s1);                   // replace sb[i1 .. i2] 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());
  }
}
```

Simple list handling in Java

The following is the specification of useful members of a Java (1.4) list handling class useful in developing translators as discussed in this course. This class will "work" with Java 5.0, but the compiler will issue warnings, as `ArrayList` has been redefined to be a "generic" class.

```
class ArrayList
// Class for constructing a list of objects

  public ArrayList()
  // Empty list constructor

  public void add(Object o)
  // Appends o to end of list

  public void add(int index, Object o)
  // Inserts o at position index
```

```
    public Object get(int index)
    // Retrieves an object from 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(Object o)
    // Returns true if o is in the list

    public boolean indexOf(Object o)
    // Returns position of o in the list

    public Object remove(int index)
    // Removes the object at position index

  } // ArrayList
```

A quick guide to simple HTML authoring

These notes serve as a sort of "learn by studying an example" guide to simple HTML authoring. The source of a simple web page is stored in a simple text file, with the extension `.htm` or `.html`. A close equivalent of the HTML used in this demonstration can be found in the file `sample.htm`, which can be opened in raw form using an editor like UltraEdit, or opened as a web page from within your favourite browser.

Within an HTML file:

`<!--` HTML comments appear bracketed in this sort of way `-->`

The information to be displayed in the top bar appears in a section demarcated by "head" and "/head> tags, and the text itself is bracketed by "title" and "/title" tags:

`<head>`
    `<title>`A simple guide to HTML authoring`</title>`
`</head>`

`<h2>`**Some simple things to remember about HTML**`</h2>`

Within a web page itself, use is made of various tags as directives to the browser. These are enclosed in < angle brackets >, and an opening tag generally matches a closing tag - so, for example an "`<h3>`" tag intoduces a type of highlighting that remains in force until a matching "`</h3>`" tag is encountered. (The "hN" tags are used to highlight section headings.)

`<h3>`**Section One: Paragraphs and line breaks**`</h3>`

Within a section, the "p" tag introduces a new paragraph.

`<p>`
As you can see, we have a new paragraph!

`<p>`
Here is another new paragraph. Within a paragraph, text is free form and is arranged by the browser as it sees fit and no account is taken of line breaks that appear in the source. To force a line break, use is made of a "br" tag.

`<br>`This sentence starts on a new line.

`<br>`Here is another sentence starting on a new line.

<p>One can indent information by bracketing it with the "blockquote" and "/blockquote" tags. So for example:

<blockquote>

Students' attention is drawn to the need to attend lectures regularly.

</blockquote>

<p>A section separator can be introduced quite effectively using an "hr" tag:

<p><hr><p>

<h3>**Section Two: Highlighting and bullets**</h2>

<i>Text can be italicized by bracketing it with "i" and "/i" tags</i>;
<b>text can be also emphasized by bracketing it with "b" and "/b" tags</b>.

<p>The useful "ul" and "/ul" tags can be used in conjunction with "li" tags to create bulleted lists:

Attention is drawn to

<ul>
  <li> the first point

  <li> the second point

  <li> the third point
</ul>

<p><hr><p>

<h3>**Section Three: Hyperlinks**</h2>

References to web pages are created by using the "a" and "/a" tags in a construction that incorporates the URL, as exemplified by:

<blockquote>

Further information is available from the
<a href=http://www.cs.ru.ac.za/courses/CSC301/Translators/trans.htm>
course web page</a>.

</blockquote>

<p>As an alternative to an URL, one sometimes uses a "mailto" variant, as in

<blockquote>

Direct questions to
<a href=mailto:p.terry@ru.ac.za>
Pat Terry (p.terry@ru.ac.za)</a>.

</blockquote>