RHODES UNIVERSITY

Computer Science 301 - 2004 - 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 10h30 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 10h30 pm 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 the complete sources for the Parva compiler you were supplied in Practical 25. 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` run `CMAKE Parva` ... generally have hours of fun.

But note that the exam set up has *no* connection with the outside world - no web browser, 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 compiler 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 15 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

```
                         \
                    PDT(8) \      <==================================== here!
                            |\
     \     \               /---\
       \   Bedford       /       \  Constitution St
Cradock  \     \       /          \
           \     \   /              \      ↑Hospital
             \    Pear Lane           \     |
               |          Worcester St  \   |
       --------+----------------------------+
        DSG    |                            |
               | St Andrew's                |Milner St
               |                            |
      Somerset |              African St    |
      Street   +--------+----------------------+----------
               |        |                      |
               |        |                      |
               |        |                      |
       --------+   Rat  |          New St      |
               +------+-+----------------------+----------
   Hamilton    |      |           The Suite    |
     Hell Hole |      |                        |
               |      |                  Gino's|Hill St
     Rhodes    |      |     High St            |
             --+------+------------------------+
                                                 Cathedral
```

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

Section B [95 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.

2004 has been a great year for anniversaries: 10 years of democracy; 10 years of 24-hour compiler course examinations; 100 years of excellence at Rhodes University; 60 years since the allies invaded Europe on D-Day; 40 years since the Beatles invaded the United States; 50 years of MacDonald's Hamburgers; 50 years of Elvis Presley recordings. The list is endless.

It is also 50 years after Backus started work on the programming language FORTRAN.

Regular readers of this column - the Compiler Course Examination Archives (CCEA) - will know that this time of the year usually sees a crisis develop in the Computer Science Department, and this year is no exception. As part of the Rhodes Centenary Celebrations, each department has been mounting exhibitions that incorporate their most important relics of the past. A potential rich research funder is to visit our exhibition on the day after tomorrow, and a lot is at stake. His silver hair suggests that he only ever programmed in FORTRAN, and so it is slightly unfortunate that we have not found a FORTRAN compiler, let alone one that targets the ground-breaking PVM (Parva Virtual Machine) on which the Department's research reputation is increasingly based.

"No problem", exclaimed the illustrious Head of Department. "Write one! I know that the first FORTRAN compiler is reputed to have taken 18 person-years of effort, but we don't need a full FORTRAN compiler - we need only demonstrate a carefully chosen subset compiler and that should easily convince the potential funder that we have the Real Thing".

Very simple FORTRAN programs are not that hard to code or understand. They have a single program unit that starts with a `PROGRAM` line and ends with an `END` line. In between these come, firstly, a list of variable declarations, and then, secondly a sequence of executable statements. In the original FORTRAN, only upper-case characters were allowed, but today it is generally taken as a case-insensitive language. Only one statement may appear on a line, so a simple example that would impress our visitor immensely might be provided by:

```
      PROGRAM Greeting
  C:  Comments start with C: and go on to the end of the line
      INTEGER Year, Born
      Year = 2004
      PRINT *, 'When were you born?'
      READ  *, Born
      PRINT *, 'That means you''re ', Year - Born, ' years old!', EOLN
      STOP
      END
```

The asterisks in the READ and PRINT statements denote input from the "standard input" (keyboard) and output to the "standard output" (screen) devices respectively. The asterisk in READ statements is followed by a list of designators in a familiar way, and in PRINT statements by an obvious list of expressions and strings. Unlike Java, FORTRAN literal strings are bracketed with single apostrophes. If a string is to contain an apostrophe, this is denoted by two apostrophes in succession, as in the example just given, which would display something like:

```
 That means you're 59 years old!
```

if the program were executed. Other escape sequences like the familiar \n and \t found in Java strings are not allowed. Although it is not really part of standard FORTRAN, we suggest using the token EOLN to represent "output an end of line sequence".

For the purposes of this exercise, limit variables to being of only two types, denoted by INTEGER (int) and LOGICAL (Boolean), and demand that they be declared as in the following examples:

```
      INTEGER I, J, K, List(12)
      LOGICAL Sieve(4000), IsEasy, IsOld, CanRetire
      INTEGER N, Age
```

where arrays are indicated (and storage automatically allocated to them) by indicating the uppermost permitted value of the (integer) subscript in parentheses, for those identifiers that are to denote arrays.

Arithmetic (integer) expressions can contain the usual +, -, * and / operators. In forming logical (Boolean) expressions the tokens .EQ., .NE., .LT., .LE., .GT., .GE., .AND., .OR. and .NOT. are used, and the logical constants are denoted by .FALSE. and .TRUE. Within expressions, array elements are selected using index expressions contained in (round) parentheses rather than [square] brackets. All this rather clumsy notation comes about because of the limited character sets available on computers 50 years ago. Precedence rules are effectively the same as we still have in Java. Here are some examples of simple assignments in FORTRAN:

```
      IsOld = Age .GE. 25
      Profit = Items * (Sell - Cost)
      CanRetire = Age .GT. 55 .AND. Pension .GT. 100000 .OR. WifeInsists
      Average = (List(1) + List(2) + List(3)) / 3
```

Where FORTRAN differs significantly from the languages most familiar to you is in the way in which it handles branching and looping. As FORTRAN evolved, so too did its control structures, but our old visitor might not recognize all of those, so we should rather cater for the traditional forms. Chief among these is the GOTO statement. An executable FORTRAN statement can be associated with a unique label, and such labelled statements can be the target of GOTO statements, as exemplified by the mindless program:

```
      PROGRAM Parrot
  10  PRINT *, 'Pretty Polly '
      GOTO 10
      END
```

Of course, one needs somewhat more sophistication. A rather strange statement found in the original FORTRAN is the so-called "arithmetic IF" statement, exemplified by:

```
      IF (A - B * C) 10, 20, 30
```

The dynamic semantics of this statement form are as follows: the parenthesized expression - which has to be "arithmetic" rather than "logical" - is evaluated, followed by one of a GOTO 10 (the first label) if the result is negative, a GOTO 20 (the second label) if the result is zero, and a GOTO 30 if the result is positive. All three labels have to be provided (and, of course, each label has to be attached to a statement somewhere within the program). Here is a more complete example:

```
      PROGRAM Decide
      INTEGER I, J
  90  READ *, I, J
      IF (I - J) 20, 500, 500
  20  PRINT *, 'I is less than J'
      GOTO 30
  500 PRINT *, 'I is greater than or equal to J'
  30  STOP
      END
```

This may strike you as a bit tortuous, and it is not hard to see that a program with many `GOTO` statements and labels (which could be assigned to statements in any order) can become hard for a human reader to understand.

A little later in the history of FORTRAN came the introduction of the "logical `IF`" statement. In this statement the parenthesized expression after `IF` has to be "logical" rather than "arithmetic", and is followed by a single statement which is executed if the expression evaluates to *true*. Again some examples will clarify:

```
      IF (A .GT. B) PRINT *, 'A is greater than B'

      Total = 0
  10  READ *, I
      Total = Total + I
      IF (I .NE. 0) GOTO 10
      PRINT *, 'Total = ', Total
```

This "logical `IF`" statement did not provide for an `ELSE` clause (that came even later in the history of FORTRAN) and the auxiliary statement could only be one of a limited set of possibilities - it could be a `READ, PRINT, STOP, CONTINUE, GOTO,` or an assignment, but not another `IF` statement.

The `STOP` statement does the obvious thing (halts program execution) and the `CONTINUE` statement does "nothing" - it is a useful way of introducing an extra label into a program if that is ever needed.

The last control statement we should like to demonstrate to our visitor is the `WHILE` statement, which is exemplified by the following code (which also incorporates simple array handling):

```
      Total = 0
      N = 1
      Read *, Item
      WHILE (Item .NE. 0)
        List(N) = Item
        N = N + 1
        READ *, Item
      ENDWHILE
```

Here the parenthesized expression in the `WHILE` statement must be a "logical expression", and the body of the loop consists of the statements between the `WHILE` statement itself and the distinctive `ENDWHILE` statement. `WHILE` loops can be nested, and `ENDWHILE` statements can be labelled, so a larger example might be:

```
      I = 0
      WHILE (I .LE. 10)
        J = 0
        WHILE (J .LE. 0)
          PRINT *, I * J
        ENDWHILE
        PRINT *, EOLN
  10  ENDWHILE
```

`WHILE` and `ENDWHILE` statements cannot form part of a "logical `IF`" statement.

Save the honour of the Department! Spend the next 24 hours using Coco/R to develop a subset FORTRAN compiler that targets the PVM and handles the set of statements loosely described above, and then present a report and a Cocol grammar showing how you would do this. To assist you in this task we shall provide you with an attributed grammar and the usual support modules, from which a working Parva compiler/interpreter system can be constructed. This is essentially the same as the one which you explored in the practical course, but with the part of the compiler that deals with expressions already modified to incorporate the C#/Java rules for precedence. It should be apparent that large parts of the Parva compiler can be incorporated into the FORTRAN compiler almost unchanged, and you are encouraged to do so, or to modify components (such as the PVM or symbol table handlers) as you see fit. The Parva system forms part of a kit that also includes various other sample FORTRAN programs that you may find useful in developing and testing your compiler.

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) You should be able to use the Parva compiler as the basis of your solution. For example the main production in the original Parva compiler is

```
Parva
=  "void" Ident "(" ")" "{"
     { Statement }
   "}" .
```

It does not give much away to suggest that to make simple changes to this production to yield a very similar one

```
Parva
=  { EOL }                      /* allow for leading blank lines */
   "PROGRAM" Ident EOL          /* the first line */
     { Statement }              /* the statements */
   "END" EOL .                  /* the last line */
```

will get you going on the development of a toy Fortran compiler. Various other productions for the Fortran system will also be very similar to their Parva equivalents.

(b) Continue to call the system "Parva" - this will avoid complications with changing namespaces, packages, libraries and so on.

(c) The exam kit contains quite a large number of simple Fortran examples. They appear on an attached listing in an order that you may find useful in completing the exercise in an incremental fashion. For the most part the early ones relate to fairly simple changes to the Parva compiler. The later ones are generally harder, and you can expect to be quite challenged when finding solutions.

(d) You have been supplied with a listing of the entire Parva compiler and its support modules, 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 the listing 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 compiler.

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

(f) 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 Practical 18".

Summary of useful library classes

```
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
```

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 will be found to be 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
  }
}
```