RHODES UNIVERSITY

November Examinations - 2011

Computer Science 301 - Paper 2

Examiners:
Prof P.D. Terry
Prof D.G. Kourie

Time 4 hours
Marks 180
Pages 17 (please check!)

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

A word of advice: The influential mathematician R.W. Hamming very aptly and succinctly professed that "the purpose of computing is insight, not numbers".

Several of the questions in this paper are designed to probe your insight - your depth of understanding of the important principles that you have studied in this course. If, as we hope, you have gained such insight, you should find that the answers to many questions take only a few lines of explanation. Please don't write long-winded answers - as Einstein put it "Keep it as simple as you can, but no simpler".

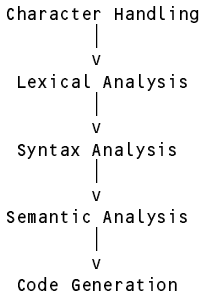
Good luck!

(For the benefit of future readers of this paper, various free information was made available to the students 24 hours before the formal examination. This included an augmented version of "Section C" - a request to devise a pretty-printer for Parva programs compatible with the Parva compiler/interpreter system studied in the course. Some 16 hours before the examination a complete grammar for a working pretty-printer and other support files for building this system were supplied to students, together with a grammar for the Mikra language, along with an appeal to study this in depth (summarized in "Section D"). During the examination, candidates were given machine executable versions of the Coco/R compiler generator, the files needed to build the basic systems, access to a computer, and machine readable copies of the questions.)

Section A: Conventional questions [102 marks]

QUESTION A1 **[5 + 6 + 4 = 15 marks]**

A compiler is often described as incorporating several "phases", as shown in the diagram below



These phases are often developed in conjunction with an Error Handler and a Symbol Table Handler.

(a) Annotate the diagram to suggest the changes that take place to the representation of a program as it is transformed from source code to object code by the compilation process. [5 marks]

(b) Suggest - perhaps by example - the sort of errors that might be detected in each of the phases of the compilation process. [6 marks]

(c) The Parva compiler studied in the course is an example of what is often called a "one pass incremental compiler" - one in which the various phases summarized above are interleaved, so that the compiler makes a single pass over the source text of a program and generates code as it does so, without building an explicit AST. Discuss briefly whether the same technique could be used to compile one class of a Java program (consider the features of Java and Parva that support the use of such a technique, and identify features that seem to act against it). [4 marks]

QUESTION A2 **[3 + 6 + 4 + 4 = 17 marks]**

(a) The regular expression 1 (1 | 0)* 0 describes the binary representations of the non-zero even integers (2, 4, 6, 8 …). Give a regular expression that describes the binary representations of non-negative integers that are exactly divisible by 4 (four), that is (0, 4, 8, 12, 16 ..).[3 marks]

(b) Currency values are sometimes expressed in a format exemplified by

R12,345,101.99

where, for large amounts, the digits are grouped in threes to the left of the point. Exactly two digits appear to the right of the point. The leftmost group might only have one or two digits. Either complete the Cocol description of a token that would be recognized as a currency value or, equivalently, use conventional regular expression notation to describe a currency value. [6 marks]

Your solution should not accept a representation that has leading zeroes, like R001,123.00.

```
CHARACTERS
   digit =

TOKENS
   currency =
```

(c) The Parva compiler supports the integer and Boolean types. Integer constants (literals) are represented in the usual "decimal" system, for example 1234.

Suppose we wished to allow hexadecimal and binary representations of integer literals as well, for example 012FH and 01101% respectively. Which of the phases of compilation identified in question A1 would this affect, which would it not affect, and why? [4 marks]

(d) A Cocol grammar for constructing a Parva compiler might contain the productions below for handling the declaration (and possible initialization) of variables.

```
VarDeclarations<StackFrame frame>     (. int type; .)
=  Type<out type>
   OneVar<frame, type>
   { "," OneVar<frame, type> } ";" .

OneVar<StackFrame frame, int type>    (. int expType; .)                // version 1
=                                     (. Entry var = new Entry(); .)
   Ident<out var.name>                (. var.kind = Entry.Var;
                                         var.type = type;
                                         var.offset = frame.size;
                                         frame.size++; .)
   [ AssignOp                         (. CodeGen.loadAddress(var); .)
     Expression<out expType>          (. if (!compatible(var.type, expType))
                                           SemError("incompatible types in assignment");
                                         CodeGen.assign(var.type); .)
   ]                                  (. Table.insert(var); .)
.
```

The symbol table insertion for each variable has been made at the end of the `OneVar` production. A student has queried whether this production could be changed to read as follows, where the symbol table entry is made earlier. The lecturer, by way of reply, asked her to consider what the outcome would be of trying to compile the statement

```
int i = i + 1;
```

What would your considered reaction be? [4 marks]

```
OneVar<StackFrame frame, int type>    (. int expType; .)                // version 2
=                                     (. Entry var = new Entry(); .)
   Ident<out var.name>                (. var.kind = Entry.Var;
                                         var.type = type;
                                         var.offset = frame.size;
                                         Table.insert(var);            // +++++ moved
                                         frame.size++; .)
   [ AssignOp                         (. CodeGen.loadAddress(var); .)
     Expression<out expType>          (. if (!compatible(var.type, expType))
                                           SemError("incompatible types in assignment");
                                         CodeGen.assign(var.type); .)
   ]
.
```

QUESTION A3 **[2 + 3 + 2 + 4 + 12 + 3 = 26 marks]**

Long ago, the Romans represented 1 through 10 by the strings

```
I   II   III   IV   V   VI   VII   VIII   IX   X
```

The following grammar attempts to recognize a sequence of such numbers, separated by commas and terminated by a period:

```
COMPILER Roman
  PRODUCTIONS
    Roman  = Number { "," Number }  "."  EOF .
    Number = StartI | StartV | StartX .
    StartI = "I" ( "V" | "X" | [ "I" [ "I" ] ] ) .
    StartV = "V" [ "I" ] [ "I" ] [ "I" ] .
    StartX = "X" .
END Roman.
```

(a) What do you understand by the concept of an *ambiguous* grammar? [2 marks]

(b) Explain carefully why this particular grammar is ambiguous? [3 marks]

(c) What do you understand by the concept of *equivalent* grammars? [2 marks]

(d) Give an equivalent grammar to the one above, but which is *unambiguous?* [4 marks]

(e) Even though the grammar above is ambiguous, develop a matching hand-crafted recursive descent parser for it, similar to those that were developed in the practical course.

Assume that you have `accept` and `abort` routines like those you met in the practical course, and a scanner `getSym()` that can recognise tokens that might be described by an

```
EOFSym, noSym, commaSym, periodSym, iSym, vSym, xSym
```

Your parser should detect and report errors, but there is no need to incorporate "error recovery". [12 marks]

(f) If the grammar is ambiguous (and thus cannot be LL(1)), does it follow that your parser might fail to recognize a correct sequence of Roman numbers, or might report success for an invalid sequence? Justify your answer. [3 marks]

QUESTION A4 **[3 + 3 = 6 marks]**

In the compiler studied in the course the following production was used to handle code generation for a *WhileStatement*:

```
WhileStatement<StackFrame frame>      (. Label startLoop = new Label(known); .)
=  "while" "(" Condition ")"          (. Label loopExit = new Label(!known);
                                         CodeGen.branchFalse(loopExit); .)
   Statement<frame>                   (. CodeGen.branch(startLoop);
                                         loopExit.here(); .) .
```

This generates code that matches the template

```
startLoop:    Condition
              BZE  loopExit
              Statement
              BRN  startLoop
loopExit:
```

Some authors contend that it would be preferable to generate code that matches the template

```
whileLabel:   BRN  testLabel
loopLabel:    Statement
testLabel:    Condition
              BNZ  loopLabel
loopExit:
```

Their argument is that in most situations a loop body *is* executed many times, and that the efficiency of the system will be markedly improved by executing only one conditional branch instruction on each iteration.

(a) Do you think the claim is justified for an interpreted system such as we have used? Explain your reasoning. [3 marks]

(b) If the suggestion is easily implementable in terms of our code generating functions, show how this could be done. If it is not easily implementable, why not? [3 marks]

QUESTION A5 **[6 + 10 + 4 = 20 marks]**

Scope and *Existence/Extent* are two terms that come up in any discussion of the implementation of block-structured languages.

(a) Briefly explain what these terms mean, and the difference between them. [6 marks]

(b) Briefly describe a suitable mechanism that could be used for symbol table construction to handle scope rules and offset addressing for variables in a language like Parva. Illustrate your answer by giving a snapshot of the symbol table at each of the points indicated in the code below. (The first one has been done for you, and you can fill in the rest on the addendum page.) [10 marks]

```
void main () {
  int[] list = new int[4];
  int i, j, k;                 // compilation point 1

  if (i > 0) {
    int a, b, c;               // compilation point 2

  } else {
    int c, a, d;               // compilation point 3

  }
  int[] b = new int[3], last;  // compilation point 4
}
```

Point 1

Name	list	i	j	k							
Offset	0	1	2	3							

(c) If the declaration at point 3 were changed to

```
int c, a, i;                 // compilation point 3
```

then the code would be acceptable to a C++ compiler but not to a Parva, Java or C# compiler, as C++ allows programmers much greater freedom in reusing/redeclaring identifiers in inner scopes. What benefits do you suppose language designers would claim for either greater or reduced freedom? [4 marks]

QUESTION A6 **[8 marks]**

The following Parva program exemplifies two oversights of the sort that frequently trouble beginner programmers - the array `list` has been declared, but never referenced, while the variable `total` has been correctly declared, but has not been defined (initialised) before being referenced.

```
void main () {
  int item, total,
  int[] list = new int[10];
  read(item);
  while (item != 0) {
    if (item > 0) total = total + item;
    read(item);
  }
  write("total of positive numbers is ", total);
}
```

Discuss the extent to which these "errors" might be detected by suitable extensions of the Parva compiler/interpreter system developed in this course. You do not need to give the algorithms in detail, but you might like to structure your answer on the following lines: [8 marks]

Variables declared but never referenced:

Not detectable at compile time because ….
or Detectable at compile time by ….
and/or Not detectable at run time because ….
or Detectable at run time by ….

Variables referenced before their values are defined:

Not detectable at compile time because ….
or Detectable at compile time by ….
and/or Not detectable at run time because ….
or Detectable at run time by ….

QUESTION A7 **[10 marks]**

Here is a true story. A few years ago I received an e-mail from one of the many users of Coco/R in the world. He was looking for advice on how best to write Cocol productions that would describe a situation in which one non-terminal `A` could derive four other non-terminals `W, X, Y, Z`. These could appear in any order in the sentential form, but there was a restriction that each one of the four had to appear exactly once. He had realised that he could enumerate all 24 possibilities, on the lines of

```
A = W X Y Z | W X Z Y | W Y X Z | ....
```

but observed astutely that this was tedious. Furthermore, it would become extremely tedious if one were to be faced with a more general situation in which one non-terminal could derive *N* alternatives, which could appear in any order, subject to the restriction that each should appear exactly once.

Write the appropriate parts of a better Cocol specification that describes the situation and checks that the restrictions are correctly met. (Restrict your answer to the case of 4 derived non-terminals, as above.) [10 marks]

Section B: Converting Mikra programs to Parva [78 marks]

Please note that there is no obligation to produce machine readable solutions 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 EXAMC.ZIP or EXAMJ.ZIP, modify any files that you like, and then copy all the files back onto an exam folder on the network.

Yesterday you made history when you were invited to produce a pretty-printer system for Parva programs.

Later in the day you were provided with a sample solution to that challenge, and the files needed to build that system have been provided to you again today. You were also provided with an unattributed grammar for Mikra, another small language.

Today we have gone one step further and provided you not only with these systems, but also with all the files needed to produce a pretty-printer for Mikra programs.

Regular readers of the Journal of Anxiety and Stress will have realized that at about this time every year the Department of Computer Science has a last minute crisis, and 2011 is proving to be no exception. Believe it or not, the Department is due to demonstrate the use of Mikra this afternoon, but have managed to lose every single copy of the Mikra compilers once so popular among students!

What should one do in such an emergency? Dial 911? Dial the Emeritus Professor?

If you try the latter he will put on his wicked smile and ask: "Are you not one of those people who spent a happy weekend developing a pretty-printer for Parva?" And when you admit that you are, he will smile even more broadly and then say: "So what's the problem? You can construct a system that will pretty-print Mikra programs. Simply modify this so that the pretty-printer produces Parva output in place of Mikra output, and then the Department will never notice the difference - they will automagically be able to convert their Mikra programs to Parva and then compile them with the Parva compiler instead."

As usual, he's right! Mikra programs are really very similar to Parva ones, except for a little "syntactic sugar", as the following examples will illustrate:

```
program Entrance;
  begin
    writeLn("Hello world")
  end Entrance.
```

```
void main() {
  writeLine("Hello world");
} // main
```

```
program Exit;
  begin
    writeLn("Goodbye cruel world");
    writeLn("I'm off to join the circus")
  end Exit.
```

```
void main() {
  writeLine("Goodbye cruel world");
  writeLine("I'm off to join the circus");
} // main
```

```
program TimesTable;
  const limit = 12;
  var i, n : int;
  begin
    read("Which times table?", n);
    i := 1;
    while i <= limit do
      writeLn(i, n * i);
      inc(i)
    end (* while *)
  end TimesTable.
```

```
void main() {
  const limit = 12;
  int i, n;
  read("Which times table?", n);
  i = 1;
  while (i <= limit) {
    writeLine(i, n * i);
    i++;
  } /* while */
} // main
```

Go on now to answer the following questions explaining the details of the process. It is unlikely that you can complete a full system in the time available, so limit yourself to the modifications asked. As usual, a suite of simple test programs has been supplied for your use, suitably arranged to allow an incremental approach to be followed.

As a hint, many of these refinements only require you to add or modify a few lines of the attributed grammar you have been given. Your answers should be given by showing the actual code you need, not by giving a long-winded description in English! Finally, not only have you been given a `PrettyMikra.atg` file, you have also been given a `Mikra2Parva.atg` file that aleady incorporates a few of the modifications.

QUESTION B8 **[3 marks]**

How would you classify a piece of software like `Mikra2Parva` - for example, is it a compiler, an assembler, a decompiler, an interpreter, some other name … Explain.

QUESTION B9 **[5 marks]**

If Parva and Mikra are so similar, why is the grammar for one of them non-LL(1), while the grammar for the other one *is* LL(1)? Do you think this is significant? Explain briefly.

QUESTION B10 **[6 + 8 + 5 + 5 = 24 marks]**

A few of the productions for `Mikra2Parva` have already been converted in the grammar supplied to you, so that you can get an idea of how to complete many of the others. For example, you will find

```
PRODUCTIONS
  Mikra2Parva                  (. String name; .)
  = "program"                  (. CodeGen.append("void main() {"); .)
     Ident<out name>
     ";"                       (. CodeGen.indentNewLine(); .)
     Block
     Ident<out name>
     "."
  .

  Block
  = {   ConstDeclarations
      | VarDeclarations }
    "begin"
    StatementSequence
    "end"                      (. CodeGen.append(" // main"); .)
  .

  StatementSequence
  = Statement { ";"            (. CodeGen.newLine(); .)
    Statement }                (. CodeGen.exdentNewLine(); CodeGen.append("}"); .)
  .

  WhileStatement
  = "while"                    (. CodeGen.append("while ("); .)
    Condition "do"             (. CodeGen.append(") {"); CodeGen.indentNewLine(); .)
    StatementSequence
    "end"
  .
```

```
HaltStatement
= "halt"                    (. CodeGen.append("halt;"); .)
.

AssignOp
= ":="                      (. CodeGen.append(" = "); .)
.
```

which means that if you build the system and restrict the form of the Mikra programs to very simple ones like

```
program Stop;
  begin
    halt
  end Stop.
```

```
void main() {
  halt;
} // main
```

you should be able to convert them immediately. A few more changes on the same lines and you should be able to handle examples like

```
program DeepPhilosophy;
  begin
    write("I maintain that ");
    write((3 + 4) <> 6, " is true");
    writeLn
  end DeepPhilosophy.
```

```
void main() {
  write("I maintain that ");
  write((3 + 4) != 6, " is true");
  writeLine();
} // main
```

(a) Make the appropriate changes to handle the differences in the syntax for arithmetic, boolean and assignment operators. [6 marks]

(b) Make the appropriate changes to handle the differences in the syntax for *read* and *write* statements. [8 marks]

(c) Mikra has a *repeat* loop where Parva has a *do-while* loop. How do you modify the system to transform a *repeat* loop to a *do-while* loop? [5 marks]

(d) Mikra has a *loop … exit … end* loop which has no direct equivalent in Parva, but is still easily handled. Make the appropriate changes to handle this construct. [5 marks]

QUESTION B11 **[5 marks]**

If Mikra can define a *repeat* loop by a production equivalent to

RepeatStatement = "repeat" Statement { ";" Statement } "until" Condition .

could the designer of Parva have defined a *do-while* loop by a production equivalent to

DoWhileStatement = "do" Statement { Statement } "while" "(" Condition ")" ";" .

If not, why not, and if so, why do you suppose Parva did not define it that way?

QUESTION B12 **[5 marks]**

Make the appropriate changes to handle the differences in the syntax for casting operations.

QUESTION B13 **[12 marks]**

Make the appropriate changes to handle the differences in the way in which variables are declared.

After completing Questions B10 through B13 you will be able to handle quite a variety of programs. Do not bother to make further changes - they can wait for a later day!

QUESTION B14 **[5 marks]**

Do you suppose it is significant that the system will simply discard comments? Explain your reasoning, but do not bother to try to retain comments - programs without comments are found all too often in any case!

QUESTION B15 **[10 marks]**

On the addendum page, complete the T-diagrams to denote

(a) the process of creating the `Mikra2Parva` executable;
(b) the process by which it is used to convert and then compile a Mikra program.

Hand this page in with your answer book.

QUESTION B16 **[9 marks]**

It should not have escaped your attention that these pretty-printers have made no use of symbol tables or of semantic analysis. While this simplifies their construction, it might have implications for their use. Discuss this point in a little detail.

END OF EXAMINATION QUESTIONS

Free information

Section C

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

A pretty-printer is a form of translator that takes source code and translates this into object code that is expressed in the same language as the source. This probably does not sound very useful! However, the main aim is to format the "object code" neatly and consistently, according to some simple conventions, making it far easier for humans to understand.

For example, a system that will read a set of EBNF productions, rather badly laid out as

```
Goal={One}.
One

= Two "plus" Four ".".

Four =Five {"is" Five|(Six Seven)}.
Five = Six [Six       ].
Six=  Two| Three    |"(*" Four "*)"|'(' Four ')'.
```

and produce the much neater output

```
Goal
  = { One } .

One
  = Two "plus" Four "." .

Four
  = Five { "is" Five | ( Six Seven ) } .

Five
  = Six  [ Six ] .

Six
  = Two | Three | "(*" Four "*)" | '(' Four ')' .
```

is easily developed by using a Cocol grammar *(supplied in the free information kit, but not reproduced here)* attributed with calls to a "code generator" consisting of some simple methods `append`, `indent`, `exdent` and so on.

As the first step in the examination candidates were invited to experiment with the EBNF system, and then to go on to develop a pretty-printer for programs expressed in a version of Parva, as described in a supplied, but unattributed grammar. A version of this grammar - spread out to make the addition of actions easy - was supplied in the pre-examination kit, along with the necessary frame files.

Section D

(Summary of free information made available to the students 18 hours before the formal examination.)

Candidates were informed that if one could assume that the Parva programs submitted to it were correct, a basic pretty-printer for Parva programs could be developed from a fairly straightforward set of attributes added to the grammar supplied at the outset *(this attributed system was supplied in full in the auxiliary examination kit and is not listed here)*. These modifications assumed the existence of a `CodeGen` class - which essentially contained the same methods as students had seen embedded in the EBNF example supplied earlier in the day.

The attention of candidates was drawn to the way in which an `indented` argument to the `Statement` production was used to handle the indentation of the subsidiary statements found in *if, while* and *do-while* statements.

Candidates were also presented with an unattributed grammar for Mikra, a small language sematically similar to Parva, but with syntax somewhat like Modula-2 *(again, supplied in full in the auxiliary pre-examination kit, and during the exam itself),*

To prepare themselves to answer the examination proper, candidates were encouraged to study the various grammars in depth, and warned that examination question would probe this understanding, and that they might be called on to make some modifications and extensions to either or both of them.

Summary of useful library classes

The following summarizes some of the most useful aspects of the available simple I/O classes.

```
public class OutFile {  // text file output
  public static OutFile StdOut
  public static OutFile StdErr
  public OutFile()
  public OutFile(String fileName)
  public boolean openError()
  public void write(String s)
  public void write(Object o)
  public void write(byte o)
  public void write(short o)
  public void write(long o)
  public void write(boolean o)
  public void write(float o)
  public void write(double o)
  public void write(char o)
  public void writeLine()
  public void writeLine(String s)
  public void writeLine(Object o)
  public void writeLine(byte o)
  public void writeLine(short o)
  public void writeLine(int o)
  public void writeLine(long o)
  public void writeLine(boolean o)
  public void writeLine(float o)
  public void writeLine(double o)
  public void writeLine(char o)
  public void write(String o,  int width)
  public void write(Object o,  int width)
  public void write(byte o,    int width)
  public void write(short o,   int width)
  public void write(int o,     int width)
  public void write(long o,    int width)
  public void write(boolean o, int width)
  public void write(float o,   int width)
```

```
  public void write(double o,  int width)
  public void write(char o,    int width)
  public void writeLine(String o,  int width)
  public void writeLine(Object o,  int width)
  public void writeLine(byte o,    int width)
  public void writeLine(short o,   int width)
  public void writeLine(int o,     int width)
  public void writeLine(long o,    int width)
  public void writeLine(boolean o, int width)
  public void writeLine(float o,   int width)
  public void writeLine(double o,  int width)
  public void writeLine(char o,    int width)
  public void close()
} // OutFile

public class InFile {    // text file input
  public static InFile StdIn
  public InFile()
  public InFile(String fileName)
  public boolean openError()
  public int errorCount()
  public static boolean done()
  public void showErrors()
  public void hideErrors()
  public boolean eof()
  public boolean eol()
  public boolean error()
  public boolean noMoreData()
  public char readChar()
  public void readAgain()
  public void skipSpaces()
  public void readLn()
  public String readString()
  public String readString(int max)
  public String readLine()
  public String readWord()
  public int readInt()
  public int readInt(int radix)
  public long readLong()
  public int readShort()
  public float readFloat()
  public double readDouble()
  public boolean readBool()
  public void close()
} // InFile
```

Strings and Characters in Java

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

```
import java.util.*;

    char c, c1, c2;
    boolean b, b1, b2;
    String s, s1, s2;
    int i, i1, i2;

    b = Character.isLetter(c);               // true if letter
    b = Character.isDigit(c);                // true if digit
    b = Character.isLetterOrDigit(c);        // true if letter or digit
    b = Character.isWhitespace(c);           // true if white space
    b = Character.isLowerCase(c);            // true if lowercase
    b = Character.isUpperCase(c);            // true if uppercase
    c = Character.toLowerCase(c);            // equivalent lowercase
    c = Character.toUpperCase(c);            // equivalent uppercase
    s = Character.toString(c);               // convert to string
    i = s.length();                          // length of string
    b = s.equals(s1);                        // true if s == s1
    b = s.equalsIgnoreCase(s1);              // true if s == s1, case irrelevant
    i = s1.compareTo(s2);                    // i = -1, 0, 1 if s1 < = > s2
    s = s.trim();                            // remove leading/trailing whitespace
    s = s.toUpperCase();                     // equivalent uppercase string
    s = s.toLowerCase();                     // equivalent lowercase string
    char[] ca = s.toCharArray();             // create character array
    s = s1.concat(s2);                       // s1 + s2
```

```
    s = s.substring(i1);                      // substring starting at s[i1]
    s = s.substring(i1, i2);                  // substring s[i1 ... i2-1]
    s = s.replace(c1, c2);                    // replace all c1 by c2
    c = s.charAt(i);                          // extract i-th character of s
//    s[i] = c;                               // not allowed
    i = s.indexOf(c);                         // position of c in s[0 ...
    i = s.indexOf(c, i1);                     // position of c in s[i1 ...
    i = s.indexOf(s1);                        // position of s1 in s[0 ...
    i = s.indexOf(s1, i1);                    // position of s1 in s[i1 ...
    i = s.lastIndexOf(c);                     // last position of c in s
    i = s.lastIndexOf(c, i1);                 // last position of c in s, <= i1
    i = s.lastIndexOf(s1);                    // last position of s1 in s
    i = s.lastIndexOf(s1, i1);                // last position of s1 in s, <= i1
    i = Integer.parseInt(s);                  // convert string to integer
    i = Integer.parseInt(s, i1);              // convert string to integer, base i1
    s = Integer.toString(i);                  // convert integer to string

    StringBuffer                              // build strings (Java 1.4)
      sb = new StringBuffer(),                //
      sb1 = new StringBuffer("original");     //
    StringBuilder                             // build strings (Jaba 1.5 and 1.6)
      sb = new StringBuilder(),               //
      sb1 = new StringBuilder("original");    //
    sb.append(c);                             // append c to end of sb
    sb.append(s);                             // append s to end of sb
    sb.insert(i, c);                          // insert c in position i
    sb.insert(i, s);                          // insert s in position i
    b = sb.equals(sb1);                       // true if sb == sb1
    i = sb.length();                          // length of sb
    i = sb.indexOf(s1);                       // position of s1 in sb
    sb.delete(i1, i2);                        // remove sb[i1 .. i2-1]
    sb.deleteCharAt(i1);                      // remove sb[i1]
    sb.replace(i1, i2, s1);                   // replace sb[i1 .. i2-1] by s1
    s = sb.toString();                        // convert sb to real string
    c = sb.charAt(i);                         // extract sb[i]
    sb.setCharAt(i, c);                       // sb[i] = c

    StringTokenizer                           // tokenize strings
      st = new StringTokenizer(s, ".,");      // delimiters are . and ,
      st = new StringTokenizer(s, ".,", true); // delimiters are also tokens
      while (st.hasMoreTokens())              // process successive tokens
        process(st.nextToken());

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

Strings and Characters in C#

The following rather meaningless code illustrates various of the string and character manipulation methods that are available in C# and which will be found to be useful in developing translators.

```
using System.Text;   // for StringBuilder
using System;        // for Char

    char c, c1, c2;
    bool b, b1, b2;
    string s, s1, s2;
    int i, i1, i2;

    b = Char.IsLetter(c);                     // true if letter
    b = Char.IsDigit(c);                      // true if digit
    b = Char.IsLetterOrDigit(c);              // true if letter or digit
    b = Char.IsWhiteSpace(c);                 // true if white space
    b = Char.IsLower(c);                      // true if lowercase
    b = Char.IsUpper(c);                      // true if uppercase
    c = Char.ToLower(c);                      // equivalent lowercase
    c = Char.ToUpper(c);                      // equivalent uppercase
    s = c.ToString();                         // convert to string
    i = s.Length;                             // length of string
    b = s.Equals(s1);                         // true if s == s1
    b = String.Equals(s1, s2);                // true if s1 == s2
    i = String.Compare(s1, s2);               // i = -1, 0, 1 if s1 < = > s2
    i = String.Compare(s1, s2, true);         // i = -1, 0, 1 if s1 < = > s2, ignoring case
    s = s.Trim();                             // remove leading/trailing whitespace
    s = s.ToUpper();                          // equivalent uppercase string
    s = s.ToLower();                          // equivalent lowercase string
```

```
    char[] ca = s.ToCharArray();              // create character array
    s = String.Concat(s1, s2);                // s1 + s2
    s = s.Substring(i1);                      // substring starting at s[i1]
    s = s.Substring(i1, i2);                  // substring s[i1 ... i1+i2-1] (i2 is length)
    s = s.Remove(i1, i2);                     // remove i2 chars from s[i1]
    s = s.Replace(c1, c2);                    // replace all c1 by c2
    s = s.Replace(s1, s2);                    // replace all s1 by s2
    c = s[i];                                 // extract i-th character of s
//    s[i] = c;                               // not allowed
    i = s.IndexOf(c);                         // position of c in s[0 ...
    i = s.IndexOf(c, i1);                     // position of c in s[i1 ...
    i = s.IndexOf(s1);                        // position of s1 in s[0 ...
    i = s.IndexOf(s1, i1);                    // position of s1 in s[i1 ...
    i = s.LastIndexOf(c);                     // last position of c in s
    i = s.LastIndexOf(c, i1);                 // last position of c in s, <= i1
    i = s.LastIndexOf(s1);                    // last position of s1 in s
    i = s.LastIndexOf(s1, i1);                // last position of s1 in s, <= i1
    i = Convert.ToInt32(s);                   // convert string to integer
    i = Convert.ToInt32(s, i1);               // convert string to integer, base i1
    s = Convert.ToString(i);                  // convert integer to string

    StringBuilder                             // build strings
      sb = new StringBuilder(),               //
      sb1 = new StringBuilder("original");    //
    sb.Append(c);                             // append c to end of sb
    sb.Append(s);                             // append s to end of sb
    sb.Insert(i, c);                          // insert c in position i
    sb.Insert(i, s);                          // insert s in position i
    b = sb.Equals(sb1);                       // true if sb == sb1
    i = sb.Length;                            // length of sb
    sb.Remove(i1, i2);                        // remove i2 chars from sb[i1]
    sb.Replace(c1, c2);                       // replace all c1 by c2
    sb.Replace(s1, s2);                       // replace all s1 by s2
    s = sb.ToString();                        // convert sb to real string
    c = sb[i];                                // extract sb[i]
    sb[i] = c;                                // sb[i] = c

    char[] delim = new char[] {'a', 'b'};
    string[]  tokens;                         // tokenize strings
    tokens = s.Split(delim);                  // delimiters are a and b
    tokens = s.Split('.' ,':', '@');          // delimiters are . : and @
    tokens = s.Split(new char[] {'+', '-'});  // delimiters are + -?
    for (int i = 0; i < tokens.Length; i++)   // process successive tokens
      Process(tokens[i]);
  }
}
```

Simple list handling in Java

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

```
import java.util.*;

class ArrayList
// Class for constructing a list of elements of type E

  public ArrayList<E>()
  // Empty list constructor

  public void add(E element)
  // Appends element to end of list

  public void add(int index, E element)
  // Inserts element at position index

  public E get(int index)
  // Retrieves an element from position index

  public E set(int index, E element)
  // Stores an element at position index

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

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

  public boolean isEmpty()
  // Returns true if list is empty
```

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

  public int indexOf(E element)
  // Returns position of element in the list

  public E remove(int index)
  // Removes the element at position index

} // ArrayList
```

Simple list handling in C#

The following is the specification of useful members of a C# (2.0/3.0) list handling class.

```
using System.Collections.Generic;

class List
// Class for constructing a list of elements of type E

  public List<E> ()
  // Empty list constructor

  public int Add(E element)
  // Appends element to end of list

  public element this [int index] {set; get; }
  // Inserts or retrieves an element in position index
  // list[index] = element;  element = list[index]

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

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

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

  public int IndexOf(E element)
  // Returns position of element in the list

  public void Remove(E element)
  // Removes element from list

  public void RemoveAt(int index)
  // Removes the element at position index

} // List
```

ADDENDUM 1 (Question A5)

Student number | | | | | | | | eg | 6 | 3 | T | 0 | 8 | 4 | 4 |

```
void main () {
  int[] list = new int[4];
  int i, j, k;                  // compilation point 1

  if (i > 0) {
    int a, b, c;                // compilation point 2

  } else {
    int c, a, d;                // compilation point 3

  }
  int[] b = new int[3], last;   // compilation point 4
}
```

Point 1

Name	list	i	j	k							
Offset	0	1	2	3							

Point 2

Name	list	i	j	k							
Offset	0	1	2	3							

Point 3

Name	list	i	j	k							
Offset	0	1	2	3							

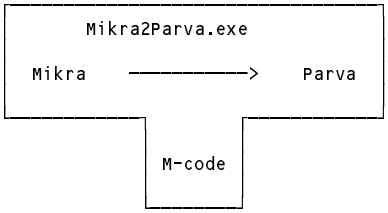
Point 4

Name	list	i	j	k							
Offset	0	1	2	3							

ADDENDUM 2 (Question B15)

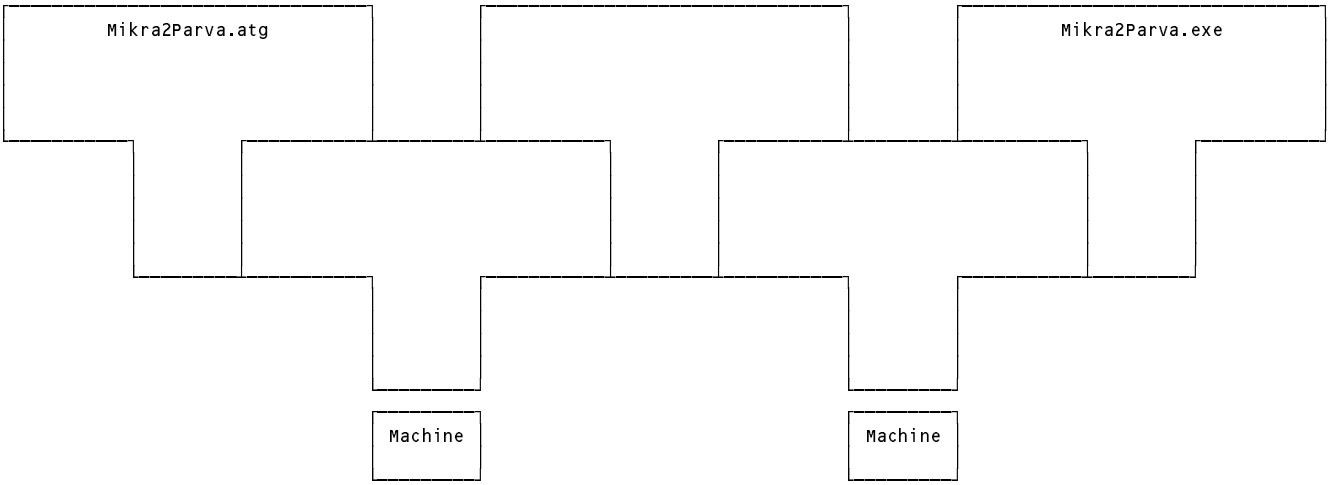
Student number | | | | | | | | eg | 6 | 3 | T | 0 | 8 | 4 | 4 |

The Mikra2Parva executable you produce might be represented

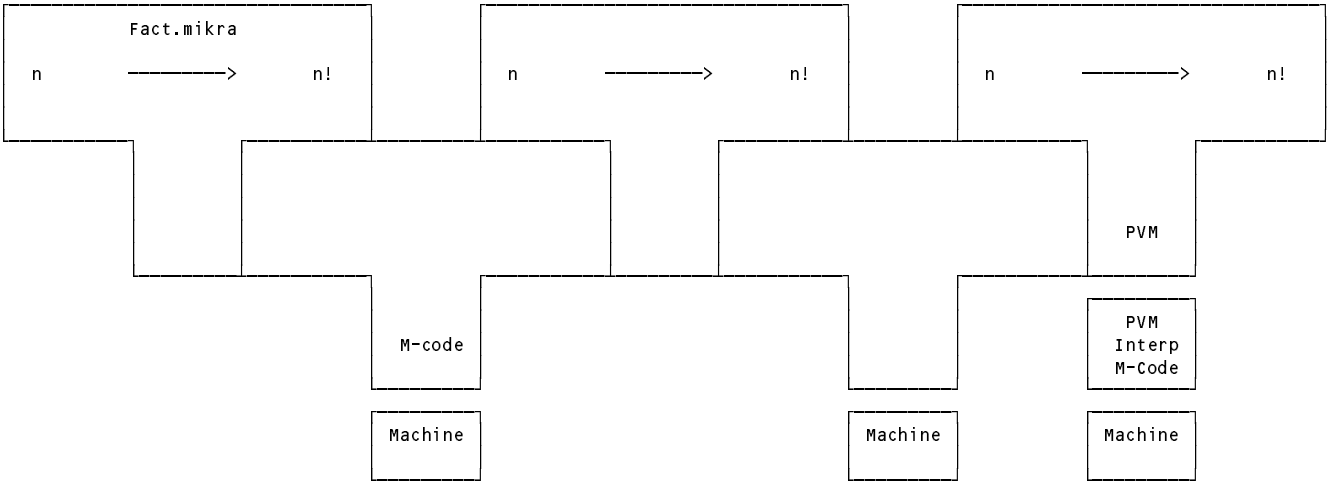


Developing the Mikra2Parva system using Coco and (either) Java (or) C#

Assume that any "real machine code" (M-code) that you produce can run directly on the "Machine"



Using the Mikra2Parva system



Examples for testing the Mikra to Parva translator

(These correspond approximately to ones suitable for questions B10 - B13).

```
program One;
(* 1.mik:  Arithmetic and boolean operators *)
begin
  write(17 mod 5, 23 > 4, 7 = 7, 8 <> 85, (3 > 4) and (4 < 6) or (7 < 12))
  (*  2 TRUE TRUE TRUE TRUE *)
end One.


program Two;
(* 2.mik:  read and write statements *)
begin
  readLn("prompt");
  write(17 mod 5, 23 > 4, 7 = 7, 8 <> 85, (3 > 4) and (4 < 6) or (7 < 12));
  writeLn;
  (*  2 TRUE TRUE TRUE TRUE *)
end Two.


program Three;
(* 3.mik:  repeat loops *)
const
  CowsComeHome = true;
begin
  repeat
    write("hello ");
    write("world")
  until 1 < 4;

  repeat
    write("hello ");
    write("world");
    repeat
      write("HELLO WORLD")
    until CowsComeHome
  until 1 < 4;
end Three.


program Four;
(* 4.mik:  loop - exit - end loops *)
const
  CowsComeHome = true;
begin
  loop
    write("hello ");
    writeLn("world");
    exit;
    write("can't get here!");
    loop (* just as well *) end
  end;
  write("escaped at last");
end Four.


program Five;
(* 5.mik:  casting operations *)
const
  AsciiA = 65, AsciiB = 66, AsciiM = 77;
begin
  writeLn( char(65), int('A') );
  writeLn( char(65 + 12), int('A' + int('B')) )
end Five.


program Six;
(* 5.mik:  variable declarations *)
var
  a, b, c : int;
  sieve : array of bool;
begin
  a := 12;
  b := true;
  sieve := new bool[a];
  sieve[a mod 3] := a > 4;
end Six.
```