RHODES UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

EXAMINATIONS : NOVEMBER 2017

COMPUTER SCIENCE 301 - PAPER 2 - COMPILERS

Examiners:
Internal : Prof P.D. Terry
External : Prof M. Kuttel

Duration: 4 hours
Marks: 180
Pages: 21 (please check!)

The Concise Oxford English Dictionary may be used during this examination.

There are fifteen (15) questions. Answer ALL questions. Answers may be written in any medium except red ink, and are preferably written in the spaces provided on the question paper. You may use pencil, and you may also answer the questions by editing the supplied electronic copies of this material.

Hand in all material at the end of the examination.

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.

Good luck!

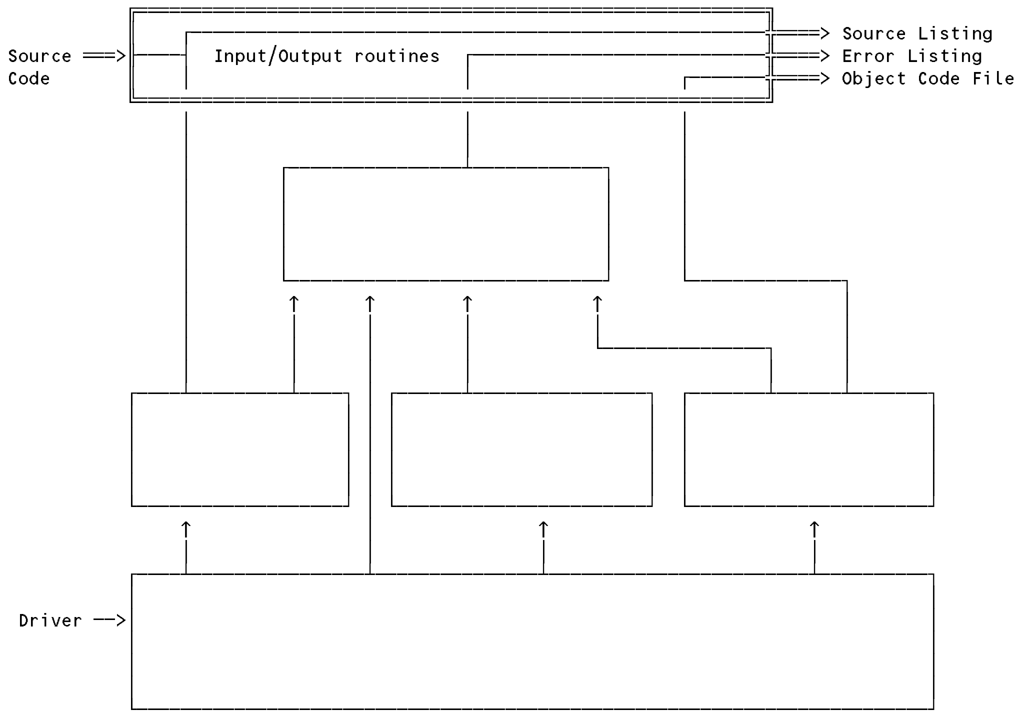
(For the benefit of future readers of this paper, various free information was made available to the students 24 hours before the formal examination. This included an augmented version of "Section C" - a request to devise a simple Boolean expression compiler targetting the Parva Virtual Machine interpreter system studied in the course. Some 16 hours before the examination a complete grammar for such a compiler and other support files for building this system were supplied to students, 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.)

DO NOT OPEN THIS PAPER UNTIL YOU ARE TOLD TO DO SO

Section A: Conventional questions [90 marks]

QUESTION A1 [8 marks]

(*Compiler structure*) A syntax-directed compiler usually incorporates various components, of which the most important are the scanner, parser, constraint analyser, error handler/reporter, code generator, symbol table handler and I/O routines. Draw a diagram indicating the dependence of these components on one another, and in particular the dependence of the central syntax analyser on the other components. [4 marks]



For each component, indicate whether it it would be considered as belonging to the front end or the back end of a compiler, and whether or not Coco/R could generate it from an attributed grammar (as opposed to generating it in some other way). [4 marks]

	Front or Back End?	Generated by Coco/R? Yes / No?
Scanner		
Parser		
Constraint Analyser		
Error Handler/Reporter		
Code Generator		
Symbol Table Handler		

QUESTION A2 **[8 + 6 + 3 + 4 + 2 + 5 = 28 marks]**

(Grammars) Formally, a grammar G is defined by a quadruple $\{ N, T, S, P \}$ with the four components

(a) N - a finite set of **non-terminal** symbols,
(b) T - a finite set of **terminal** symbols,
(c) S - a special **goal** or **start** or **distinguished** symbol,
(d) P - a finite set of **production rules** or, simply, **productions**.

where a production relates to a pair of strings, say α and β, specifying how one may be transformed into the other:

$$\alpha \rightarrow \beta \text{ where } \alpha \in (N \cup T)^* N (N \cup T)^* \ , \ \beta \in (N \cup T)^*$$

and we can then define the language $L(G)$ produced by the grammar G by the relation

$$L(G) = \{ \ w \mid \ S \Rightarrow^* w \ \wedge \ w \in T^* \ \}$$

(a) In terms of this style of notation, define **precisely** (*that is to say, mathematically; we do not want a long essay or English description*) what you understand by [2 marks each]

(1) FIRST(σ) where $\sigma \in (N \cup T)^+$

(2) FOLLOW(A) where $A \in N$

(3) A context-free grammar

(4) A reduced grammar

(b) A student, asked for a concise statement of the rules that must be satisfied by the productions of a context-free grammar in order for it to be classified as an LL(1) grammar, came up with the answer on the next page:

For each non-terminal $A_i \in N$ that admits alternatives, two rules must be obeyed. If

$$A_i \rightarrow \xi_{i1} \mid \xi_{i2} \mid \xi_{i3} \mid \ldots \xi_{in}$$

Rule 1 :

$$\text{FIRST}(\xi_{i1}) \cap \text{FIRST}(\xi_{i2}) \cap \text{FIRST}(\xi_{i3}) \cap \ldots\ldots \cap \text{FIRST}(\xi_{in}) = \varnothing$$

Rule 2 :

$$\text{FIRST}(A_i) \cap \text{FOLLOW}(A_i) = \varnothing$$

While this is admirably concise, it may not be quite correct. Suggest how it might be improved and/or corrected (quite simply). [6 marks]

(c) Describe the language generated by the following grammar, using English or simple mathematics. [3 marks]

$$S \rightarrow A\ B$$
$$A \rightarrow a\ A \mid a$$
$$B \rightarrow b\ B\ c \mid bc$$

(d) Is the grammar in (c) an LL(1) grammar? If not, why not, and can you find an equivalent grammar that *is* LL(1)? [3 marks]

(e) This simple grammar describes strings comprised of an equal number of the characters *a* and *b*, terminated by a period, such as *aababbba*. Is this an LL(1) grammar? Explain. [2 marks]

$$S \rightarrow B\ .$$
$$B \rightarrow a\ B\ b\ B \mid b\ B\ a\ B \mid \varepsilon$$

(f) "Keep it as simple as you can, but no simpler" said Einstein. Strings that might be members of the language of (e) can surely be accepted or rejected by a very simple algorithm, without recourse to the direct use of a grammar. Suggest such an algorithm, using a high-level notation. [5 marks]

QUESTION A3 **[24 marks]**

(Recursive descent parsers) An index to a departmental guide might have entries exemplified by

```
abstraction, data               165, Appendix 1, 300-312
aegrotat examinations           -- see unethical doctors
aggregate pass, chances of      0
class attendance, intolerable   12, 745
class members                   38
deadlines, compiler course      -- see sunrise
lectures missed                 1, 3, 5-9, 12, 14-19, 21-25, 28
loss of DP certificate          2017
probable exclusion from Rhodes  2018
senility, onset of              21-24, 72
subminimum for aggregation      40
```

The following Cocol grammar describes the form of such an index:

```
COMPILER Index $CN
/* Grammar describing very simple index in a departmental guide */

CHARACTERS
  digit       =   "0123456789" .
  letter      =   "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  eol         =   CHR(10) .

TOKENS
  word        =   letter { letter } .
  number      =   digit { digit } .
  EOL         =   eol .

IGNORE CHR(0)  .. CHR(9) + CHR(11) .. CHR(31)

PRODUCTIONS
  Index       =   { Entry } EOF .
  Entry       =   Key References EOL .
  Key         =   word { "," word | word } .
  References  =   DirectRefs | CrossRef .
  DirectRefs  =   PageRefs { "," PageRefs  } .
  PageRefs    =   [ "Appendix" number "," ] number [ "-" number ] .
  CrossRef    =   "--" "see" Key .
END Index .
```

(a) Assume that you have available a suitable scanner method called `GetSym` that can recognize the terminals of *Index* and classify them appropriately as members of the following set

```
{ EOFSym, noSym, EOLSym, wordSym, numberSym, appendixSym,
  commaSym, dashSym, dashDashSym, seeSym }
```

Compute the FIRST and FOLLOW sets of each of the following non-terminals in this grammar (the first one has been done for you). [10 marks]

FIRST(Entry) { wordSym }

FOLLOW(Entry)

FIRST(References)

FOLLOW(References)

FIRST(PageRefs)

FOLLOW(PageRefs)

(b) Develop a hand-crafted recursive descent parser for recognizing the index of this guide based on the grammar above. *(Your parser can take drastic action if an error is detected. Simply call methods like* `Accept` *and* `Abort`*, familiar from your practical course, to produce appropriate error messages and then terminate parsing. You are not required to write any code to implement the* `GetSym`, `Accept` *or* `Abort` *methods.)* [14 marks]

```
static void Index() {
// Index =  { Entry } EOF .



} // Index

static void Entry() {
// Entry = Key References EOL .




} // Entry

static void Key() {
// Key = word { "," word | word } .









} // Key
```

```
static void References() {
// References = DirectRefs | CrossRef .
```

```
} // References
```

```
static void DirectRefs() {
// DirectRefs = PageRefs { "," PageRefs  } .
```

```
} // DirectRefs
```

```
static void PageRefs() {
// PageRefs = [ "Appendix" number "," ] number [ "-" number ] .
```

```
} // PageRefs
```

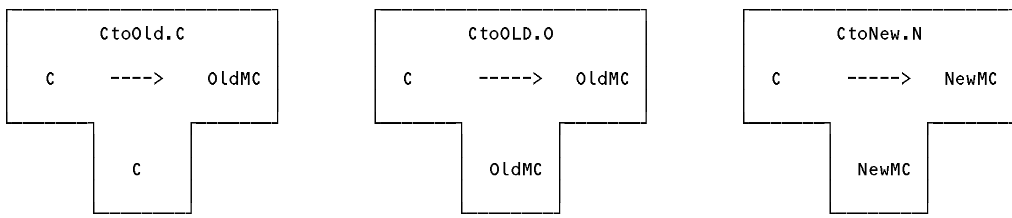
```
static void CrossRef() {
// CrossRef = "--" "see" Key .
```

```
} // CrossRef
```

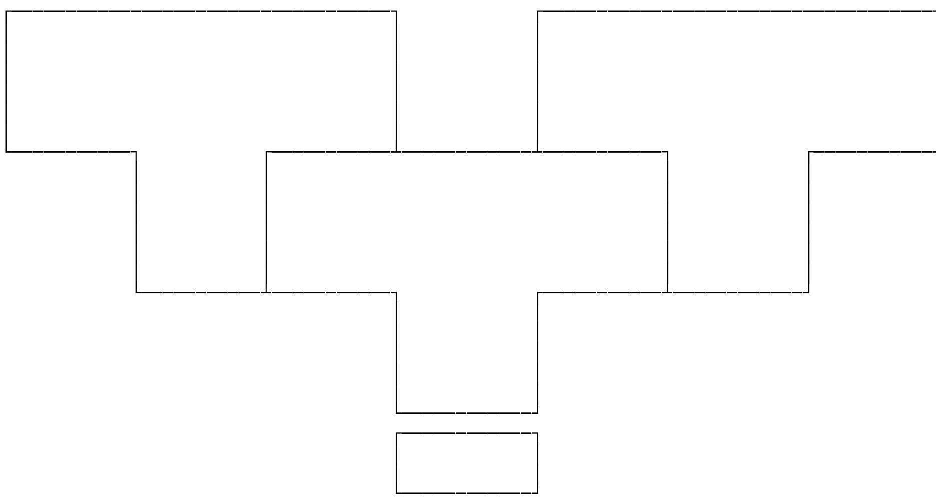
QUESTION A4 **[4 + 4 + 3 + 3 = 14 marks]**

(T diagrams) The process of "porting" a compiler for an established language to a new computer incorporates a *retargetting* phase (modifying the compiler to produce target code for the new machine) and a *rehosting* phase (modifying the compiler to run on the new machine). Enlarge on aspects of the process for porting a C compiler, by drawing a set of T diagrams. Assume that you have available the compilers (a) and (b) below and wish to produce compiler (c).

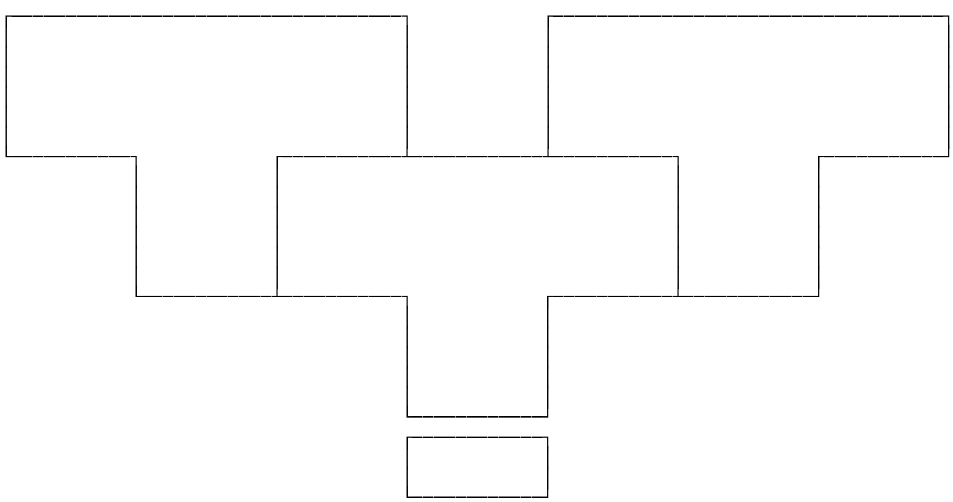
(a) Old Compiler Source (b) Old Compiler Executable (c) New Compiler Executable



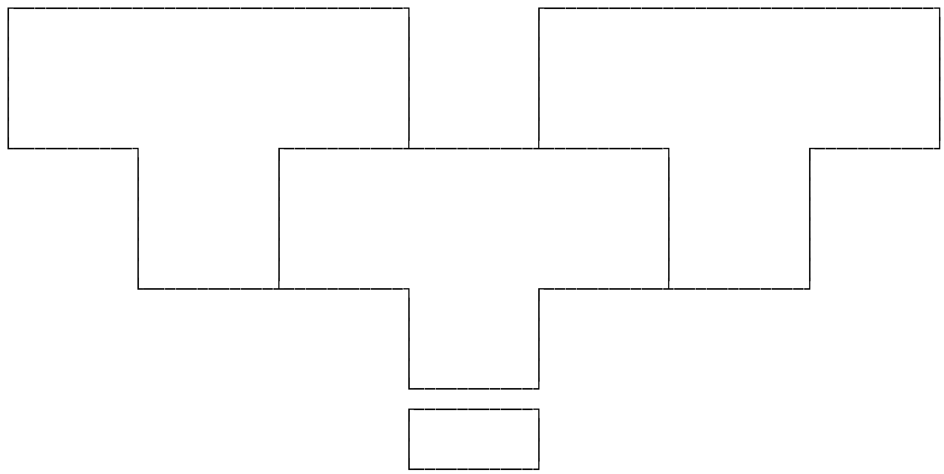
(a) Retarget the compiler. [4 marks]



(b) Rehost the compiler. [4 marks]



(c) Check a claim that the old compiler is a self-compiling compiler. [3 marks]



(d) Is the new compiler a self-compiling compiler? Justify your answer. [3 marks]

QUESTION A5 **[16 marks]**

(Attributed Grammars in Cocol) XML (eXtensible Markup Language) is a powerful notation for marking up data documents in a portable way. XML code looks rather like HTML code, but has no predefined tags. Instead, a user can create customized markup tags, similar to those shown in the following extract.

```
<!-- comment - a sample extract from an XML file -->

<personnel>
  <entry>
    <name>John Smith</name>
  </entry>
  <entry_2>
    <name>Joan Smith</name>
    <address/>
    <gender>female</gender>
  </entry_2>
</personnel>
```

An *element* within the document is introduced by an *opening tag* (like `<personnel>`) and terminated by a *closing tag* (like `</personnel>`), where the obvious correspondence in spelling is required. The name within a tag must start immediately with a letter or lowline character (_), and may then incorporate letters, lowlines, digits, periods or hyphens before it is terminated with a > character. Between the opening and closing tags may appear a sequence of free format *text* (like `John Smith`) and further *elements* in arbitrary order. The free format *text* may not contain a < character - this is reserved for the beginning of a tag. An *empty element* - one that has no internal members - may be terminated by a *closing tag*, or may be denoted by an *empty tag* - an opening tag that is terminated by `/>` (as in `<address/>` in the above example). Comments may be introduced and terminated by the sequences `<!--` and `-->` respectively, but may not contain the pair of characters `--` internally (as exemplified above).

Develop a Cocol specification, incorporating suitable CHARACTER sets and TOKEN definitions for

(a) opening tags,
(b) closing tags,
(c) empty tags,
(d) free format text

and give PRODUCTIONS that can analyse complete documents like the one illustrated .

Tags must be properly matched. A document like the following must be rejected

```
<bad.Tag>
   This is valid internal text
   <okayTag>
     More internal stuff
   </okayTag>
</badTag>  <!-- badTag should have been written as bad.Tag -->
```

Show how your grammar should be attributed to perform such checks. [16 marks]

Incidentally, it should be noted that the full XML specification defines far more features than those considered here!

```
using Library;

COMPILER XML $CN
/* Parse a set of simple XML elements */

CHARACTERS
  letter    = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz" .
  incomment = ANY - "-" .

TOKENS
  opentag  =

  closetag =

  emptytag =

  text     =

PRAGMAS /* We cannot use the comment feature of Cocol which only allows
           two character delimiters */

  comment = "<!--" { incomment | '-' incomment } "-->" .

IGNORE  CHR(0) .. CHR(31)
```

```
PRODUCTIONS
   XML   =




END XML.
```

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.

Your answers to the following questions should, whenever possible, include actual code, and not simply become a vague discussion.

QUESTION B6 **[4 marks]**

After studying the `LogicCom` grammar that has been provided, you should realize that the compiler derived from it takes a very casual approach to ensuring that the static semantics of the language are obeyed. What, in general, do you understand by the concept of *static semantics* and what is the difference between *static semantics* and *syntax*?

QUESTION B7 **[3 + 3 = 6 marks]**

(a) Can you use Boolean constants *true, false* as arguments in a function call - for example

```
Fun(x, y) returns x or y;
write(Fun(true, false));
```

Justify your answer! If you think it *is* possible, might there be any constants that you could not use in this way? Explain. [3 marks]

(b) Discuss (giving reasons) whether or not the above definition of `Fun(x,y)` can be followed by calls like [3 marks]

```
a = Fun(y, x);      // parameters seem to have been inverted
```

or

```
a = Fun(x, x);      // the same parameter has been used twice
```

QUESTION B8 **[8 + 3 = 11 marks]**

(a) The system as supplied makes no attempt to verify that the number of arguments supplied in a function call is the same as the number of parameters specified in the function definition. Remedy this deficiency. [8 marks]

(b) What might be the run-time effect of omitting this compile-time check if, for example, you compiled and then ran the program [3 marks]

```
Fun(x, y) returns x or y;      // two parameters

a = Fun(x) + Fun(x, y, z);    // one and then three arguments
```

QUESTION B9 **[2 + 10 = 12 marks]**

(a) Should it be regarded as an error if a function appears not to refer to, or to use, some of its parameters - for example

```
Fun(x, y, z)  returns x;
```

Justify your answer. [2 marks]

(b) If you wished to warn a user of this situation, give the changes to the code that would be needed to do so. [10 marks]

QUESTION B10 **[8 marks]**

Incorporate code that allows the system to detect and report erroneous function definitions like

```
Fun(x, y)  returns x or y or z;
```

and

```
Fun(x, x)  returns x and x;
```

[8 marks]

QUESTION B11 **[2 + 3 + 5 = 10 marks]**

(a) Under what conditions can one function call another? For example, can one write code like [2 marks]

```
AND3(x, y, z)     returns x and y and z;
AND4(a, b, c, d)  returns a and AND3(b, c, d);
```

(b) Given that there may be conditions in which this is possible, what would be the effect of using the supplied system with the following code? [3 marks]

```
Silly(x)   returns true and Silly(x);
write ( Silly(x) );
```

(c) What modification to the supplied system will prevent that `Silly` sort of behaviour? Explain. [5 marks]

QUESTION B12 **[8 marks]**

Code generation in the system supplied to you treats the *and* and *or* operators as binary infix operators. Change the code generation to make use of "short circuit" semantics (hint: suitable opcodes are already to be found in the supplied PVM). [8 marks]

QUESTION B13 **[10 + 5 = 15 marks]**

(a) The Logic Lecturer is bound to make another appearance. This time his request is to be able to use the traditional operators `.` `+` and `'` as well as the words *and or* and *not*, to be able to use 0 and 1 as representations of *false* and *true*, and to be able to leave the *and* operator out altogether, so that the following would be equivalent Boolean expressions [10 marks]

```
w and x and y or not z          w.x y + z'
```

(b) Implement a simple pragma $N so that one can write the value of an expression using either the words *false* and *true*, or the digits 0 and 1. A truth table for *x* and *y* in each style could then be obtained with the alternative code shown below. [5 marks]

```
writeLine(" x       y       x.y");
loop x {
  loop y { $N+ // use 0 and 1
    writeLine(x,   y,   x.y);
  }
}

      x       y      x.y

      0       0       0
      0       1       0
      1       0       0
      1       1       1
```

```
writeLine(" x       y       x.y");
loop x {
  loop y { $N- // default: use false and true
    writeLine(x,   y,   x.y);
  }
}

      x         y         x.y

      false     false     false
      false     true      false
      true      false     false
      true      true      true
```

QUESTION B14 **[8 marks]**

A danger in using the *LoopStatement* is that code in the loop might attempt to change the value of the loop variable (this is known as "threatening" the control variable). For example, code like this could prove troublesome:

```
loop x {
   read(x);
   x = not x;
}
```

Implement a system for detecting such threatening code at compile-time (and preventing such code from being executed). [8 marks]

QUESTION B15 **[8 marks]**

Examination of the way in which the *LoopStatement* has been implemented might suggest that the compiler writer has treated

```
loop x {
  ...                  // code for the body of the loop goes here
}
```

as equivalent to

```
x = false;
repeat
  ...
  x = not x;
until (x == false); // again
```

and that the code generated follows that template, leading to

```
        LDC   0
        STL   x       ; x = false;
start   .....         ; code for the body of the loop goes here
        LDL   x
        NOT
        STL   x       ; x = not x;
        LDL   x
        LDC   0       ; false
        CEQ           ; x == false?
        BZE   start   ; no - loop again
exit                  ; rest of code after loop
```

Show that this can easily be done more elegantly, and modify the code for the *LoopStatement* parser accordingly. [8 marks]

END OF THE EXAMINATION QUESTIONS

Section C

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

Candidates were provided with the basic ideas, and were invited to devise a simple compiler based on a supplied grammar to generate PVM code for evaluating Boolean expressions, ando then to extend this to allow a user to define simple "one-liner" functions that could be incorporated into the evaluation of such expressions.

It was pointed out that the PVM supplied to them incorporated the codes needed to support function calls, on the lines discussed in chapter 14 of the text book. A skeleton symbol table handler was provided, as was a code generator virtually identical to the one they had seen previously.

They were provided with an exam kit for C#, containing the Coco/R system, along with a suite of simple, suggestive test programs. They were told that later in the day some further ideas and hints would be provided.

Section D

(Summary of free information made available to the students 16 hours before the formal examination.)

A complete grammar for a rudimentary solution to the exercise posed earlier in the day was supplied to candidates in a later version of the examination kit. They were encouraged to study it in depth and warned that questions in the formal exam would probe this understanding; few hints were given as to what to expect, other than that they might be called on to comment on the solution, and perhaps to make some modifications and extensions. They were also encouraged to spend some time thinking how any other ideas they had during the earlier part of the day would need modification to fit in with the solution kit presented to them.

END OF THE EXAMINATION PAPER